UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

**FACULTATEA DE INFORMATICĂ**



LUCRARE DE LICENȚĂ

**Rolul și avantajele paternului Event Sourcing în sistemele distribuite folosind NoSQL**

**propusă de**

***Gabriel-Angelo Panțiru***

**Sesiunea:** *iulie, 2018*

**Coordonator Științific**

**Drd. Colab. Florin Olariu**

**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IASI**

**FACULTATEA DE INFORMATICA**

**Rolul și avantajele paternului Event Sourcing în sistemele distribuite folosind NoSQL**

***Gabriel-Angelo Panțiru***

**Sesiunea:** *iulie, 2018*

**Coordonator Științific**

***Drd. Colab. Florin Olariu***

DECLARAŢIE PRIVIND ORIGINALITATE ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licenţă cu titlul „*Rolul si avantajele paternului Event Sourcing în sistemele distribuite folosind NoSQL*” este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau din străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

* toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
* reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
* codul sursă, imaginile etc. preluate din proiecte *open*-*source* sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
* rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iaşi,

Absolvent *Gabriel-Angelo Panțiru*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(semnătura în original)

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Rolul si avantajele Event Sourcing in sistemele distribuite folosind NoSQL*”, codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date detest etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi,

Absolvent *Gabriel-Angelo Panțiru*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(semnătura în original)

Contents

[Introducere 6](#_Toc517646024)

[Contribuție personala 8](#_Toc517646025)

[Capitolul I. Prezentarea aplicației 9](#_Toc517646026)

[Capitolul II. Tehnologii folosite 14](#_Toc517646027)

[Capitolul II. Detalii teoretice 15](#_Toc517646028)

[Capitolul IV. Implementarea aplicației 19](#_Toc517646029)

[Proiectul Web Application 20](#_Toc517646030)

[Proiectul Services 27](#_Toc517646031)

[Proiectul Events Consummer 33](#_Toc517646032)

[Proiectul Data Layer 35](#_Toc517646033)

[Proiectul Domain 35](#_Toc517646034)

[Concluzii 38](#_Toc517646035)

[Bibliografie 39](#_Toc517646036)

[Anexa 1 40](#_Toc517646037)

[Anexa 2 42](#_Toc517646038)

[Anexa 3 44](#_Toc517646039)

[Anexa 4 46](#_Toc517646040)

[Anexa 5 49](#_Toc517646041)

[Anexa 6 50](#_Toc517646042)

# Introducere

Event Sourcing este un pattern care poate oferi foarte multe beneficii. În cele mai multe aplicații este suficient să putem interoga baza de date și să aflăm ultima stare a ei. Sunt însă multe cazuri unde ar fi avantajos să salvam toate stările prin care a trecut aplicația deoarece acest lucru poate oferi posibilități de analiza a modului cum se schimba starea aplicației ceea ce ar putea aduce beneficii aplicațiilor din anumite domenii cum ar fi cel economic. Pentru a ilustra aceste avantaje am ales sa implementez o aplicație folosind acest pattern oferind in acest fel explicații sa le leg de scenarii reale de folosire.

“Event Sourcing este o modalitate de a persista starea aplicației tale folosind, stocând o istorie din care se poate determina starea curenta a aplicatiei “[[6](https://martinfowler.com/eaaDev/EventSourcing.html)] .

Paternul de event sourcing este folosit frecvent împreună cu CQRS și acesta poate fi benefic deoarece permite separarea modelului de citire și modelului de scriere, oferind în acest mod posibilitatea de a lucra la eficiența operațiilor de citire și a operațiilor de scriere in mod separat, lucru care este util in aplicațiile în care se fac foarte multe operații de citire dar puține de scriere sau invers și este necesară eficientizarea doar a unui subset de operații care sunt mai importate pentru business.

Evenimentele pot fi procesate oricând de consumere si folosite in scopuri precum: reconstituirea ultimei stări a aplicației, analiza modului în care stările aplicației se schimbă, restaurarea aplicației la o stare din trecut sau pentru obținerea informațiilor valoroase despre business.

Am ales această temă de licență deoarece paternul de event sourcing mi s-a părut interesant iar prima oară când am auzit de el surprins deoarece nu puteam să îmi imaginez altă modalitate de a stoca datele decât cea tradițională în care fiecare entitate are salvată ultima ei stare într-o înregistrare din una sau mai multe tabele.

Deși partea inițiala cât am studiat acest patern a fost mai grea, cu cât aflam mai multe despre el, cu atât căpăta mai mult sens și începeam să descopăr tot mai multe beneficii și moduri in care ar putea fi folosit. Din aceste motive m-am hotărât sa îl studiez mai bine în lucrarea de licență.

Aplicația in ace am ales să implementez această arhitectură va fi una cu scop didactic, menită să ofere studenților un mediu virtual care să-i ajute să se organizeze mai bine și să faciliteze schimbul de informații.

În implementarea aplicației m-am concentrat pe dezvoltarea unei arhitecturi de back-end cât mai solide, peste care se poate adăuga funcționalitate ușor și care să poată fi integrată cu alte aplicații similare cu scop didactic.

Unul dintre dezavantajele acestui patern este complexitatea și faptul că are o curbă de învățare mai mare iar dezvoltarea unei aplicații cu această arhitectură necesită mai mult timp și resurse, lucru care pentru companii înseamnă un singur lucru: mai mulți bani. Cu toate acestea multe companii aleg să folosească această arhitectura în aplicațiile lor, principalul motiv fiind scalabilitatea.

În continuare voi prezenta un mic studiu de caz despre o aplicație internă a unei bănci olandeze care implementează paternul de event sourcing pentru a ilustra că arhitectura este de actualitate și că companiile aleg să investească în asemenea aplicații, chiar dacă sunt mai costisitoare. Sursa acestor informații este team-lead-ul proiectului Eugen Forașcu.

Această aplicație internă a băncii olandeze, este o aplicație web, cu back-end-ul implementat în .net, iar scopul ei este de a urmări anumite metrici ale clienților ei (alte companii care aplică sau au deja împrumuturi la această bancă), pe baza cărora angajații băncii decid dacă le pot oferi sau nu credite pentru diverse proiecte.

Această aplicație face parte dintr-un sistem distribuit, iar la înregistrarea inițiala a unui client în aplicație, interoghează celelalte componente pentru a prelua informații despre acel client. La randul lor, celelalte componente din sistem preiau informații din evenimentele aplicației implementate cu event sourcing.

Un prim motiv pentru care această aplicație a fost implementată cu această arhitectură este că datele din ea trebuiau sa fie folosite si in alte aplicații interne ale lor, iar event sourcing structurează date astfel încât să se poate face interpretări cât mai complexe asupra lor.

Un al doilea motiv pentru care această aplicație a fost implementată cu această arhitectură a fost că cliențtii vroiau o aplicație peste care să se integreze usor funcționalități noi și care să fie flexibilă la schimbări ale funcționalitaților existente.

Un alt motiv pentru folosirea acestei arhitecturi este că în domeniul bancar orice informație despre client este un factor în luarea deciziei de a-l susține sau nu în proiectele sale oferind-ui finantare. Faptul că event sourcing oferă un istoric a etapelor prin care a trecut clientul este un avantaj foarte mare deoarece în deciziile de genul acesta trecutul unui client poate cântări la fel, dacă nu mai mult, decât starea prezentă a lui.

## Contribuție personala

În această lucrare de licența am urmărit să analizez paternul event sourcing în sistemele distribuite folosind o bază de date NoSQL și beneficiile acestei arhitecturi, cum ar fi modul în care aceste sisteme pot eficientiza procesul dezvoltării software. Pentru a îndeplini acest lucru am implementat o aplicație folosind această arhitectura și o voi folosi ca studiu de caz pentru a găsi avantajele dar și avantajele utilizării acestei arhitecturi.

Aplicația pe care am implementat-o are ca scop imbunatațirea productivității personale a studenților. În implementarea aplicației am căutat să adaug tipul de funcționalităti cu care această arhitectura se descurcă cel mai bine, cum ar fi de exemplu statisticile.

În implementarea aplicației m-am folosit de o librărie CQRSlite care ofera o serie de interfete de care am legat implementarea arhitecturii CQRS combinat cu Event Sourcing.

Aplicația construită are o arhitectură de back-end solida și scalabilă, ceea ce înseamnă ca se pot adaugă ușor funcționalități noi și poate fi integrată cu alte sisteme, deoarece fiind un sistem distribuit, componentele sale sunt construite astfel încât să poată comunica cu alte componente.

De asemenea am tras ca concluzii din acestă implementare, ce tipuri de aplicații sunt cele mai potrivite pentru care această arhitectură.

# Capitolul I. Prezentarea aplicației

În ultima vreme au apărut tot mai multe aplicații care ne facilitează dezvoltarea într-un anumit domeniu oferind un mediu virtual care ne ajută să ne rezolvăm mai ușor problemele sau ne oferă o nouă perspectivă asupra domeniului în felul acesta îmbunătățindu-ne înțelegerea lui. Pornind de la ideea asta m-am gândit să fac o aplicație care oferă un mediu virtual care să ajute studenții cu studiile la facultate. Aplicația își îndeplinește scopul de a ajuta studenții să studieze mai bine prin trei moduri:

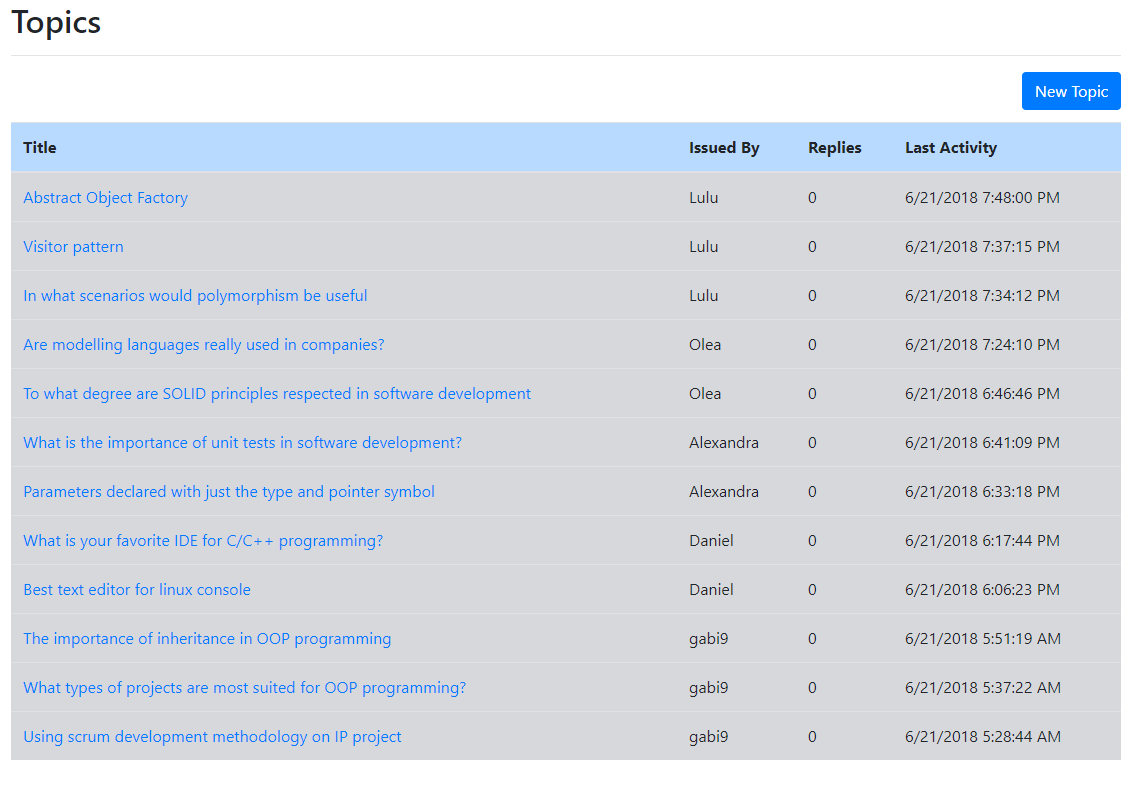
În primul mod facilitând schimbul de informații între studenți oferind un forum, care poate fi vazut în figura 1, unde pot avea loc discuții despre materii și pune întrebări cu neclaritățile. Aici un argument împotriva acestei idei ar putea fi că există grupuri pe Facebook care fac deja acest lucru, însă pe lângă aceste grupuri utilizatorul este bombardat cu informații din toate părțile: notificări de la prieteni, mesagerie sau reclamele personalizate, iar din cauză asta studenții nu se pot concentra foarte mult timp asupra a ce se postează în grup, fiind astfel nu tocmai cea mai potrivită soluție pentru a facilita schimbul de informații legat de facultate. Pe pagina inițiala a forumului, se poate găsi o lista de topic-uri, fiecare element din această listă conținând titlul topicului, numărul de răspunsuri asociate lui, data la care ultima modificare a fost făcută (aici incluzând modificările la topic-uri, răspunsuri existente sau adăugarea de răspunsuri noi) și cine a făcut ultima modificare. Tot de la aceasta pagina un utilizator are posibilitatea de a intra în pagina propriu zisă a topic-urilor sau de a intra în pagina de editare a topic-urilor create de el. În pagina de detalii a unui topic se află titlul acestuia, conținutul, data la care topicul a fost creat sau modificat, cine a creat acel topic, un buton pentru a adăuga un nou răspuns (vizibil la toți utilizatorii), buton pentru a edita acel topic (vizibil doar utilizatorului care a creat acel topic), o listă de răspunsuri fiecare conținând data la care a fost adăugat sau modificat, conținutul răspunsului, cine a scris răspunsul si un buton de editare vizibil doar celui ce a postat răspunsul.

Fig 1: Topics page

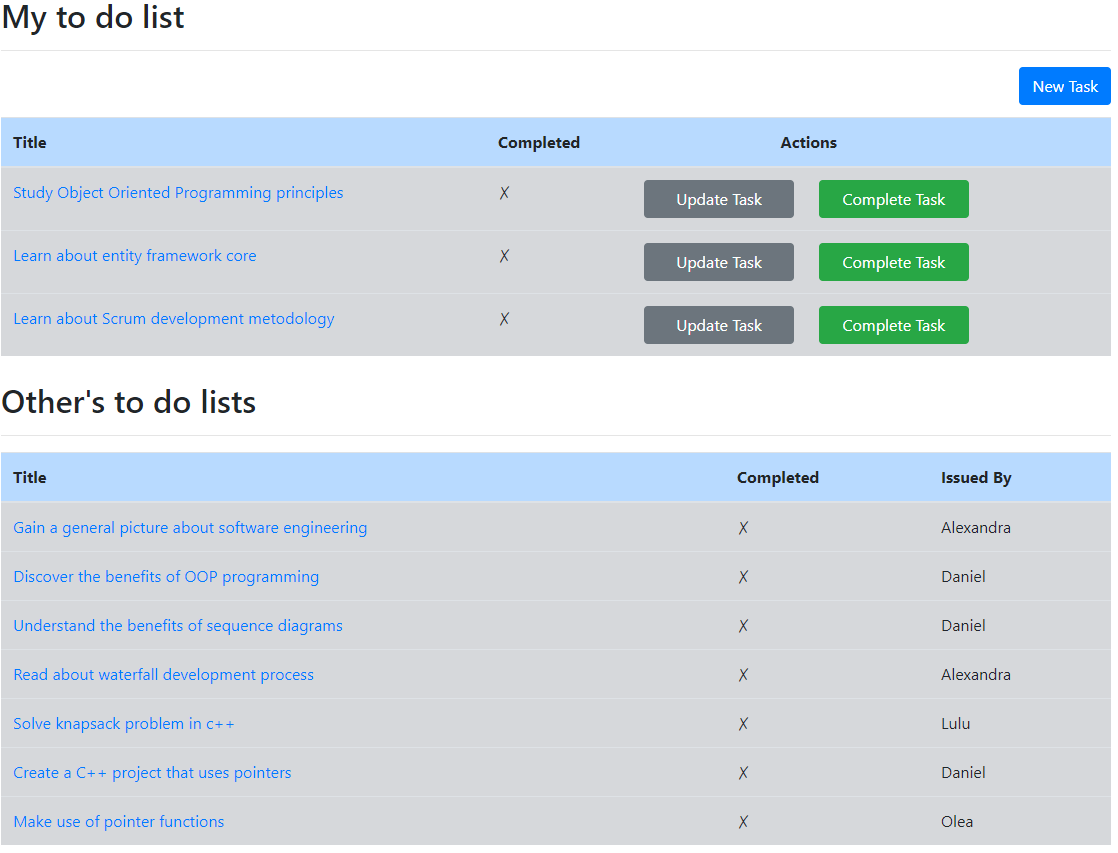
În al doilea mod îi ajută să se organizeze mai bine, oferind funcționalitate de to do list, care poate fi vazută in figura 2, unde studenții își pot crea sarcini, ofere o descriere a acestora și să ține evidența a câte ore au lucrat la acea sarcină. De asemenea tot de la ideea de a ajuta studenții să se organizeze mai bine, în aplicație studenții pot vedea și ce sarcini și-au plănuit colegii lor pentru a se inspira unii de la alții. Această secțiune de to do list este de asemenea pagină de pornire și conține prima dată o listă cu sarcinile create de utilizatorul autentificat, pentru fiecare element din listă putând fi accesate pagina cu detalii despre acel task, pagina unde se poate edita taskul și da click fie pe un buton pentru a marca sarcina ca terminată fie pentru a redeschide-o, în cazul în care a fost marcată că terminată anterior. Mai jos utilizatorul mai are acces la o lista cu sarcinile pe care și le-au creat ceilalți studenți, de unde se pot inspira, având astfel posibilitatea de a modela comportamentul colegilor a căror rezultate vor să le obțină. Deoarece aplicația este în engleză, o sarcină are denumirea de “task” și conține un titlu, câmp ce apare în pagină cu lista de sarcini, o descriere, unde studentul poate dezvolta ideea din titlu sau planifica cum anume vrea să îndeplinească sarcina și un număr de ore estimativ.

Fig 2: To do list

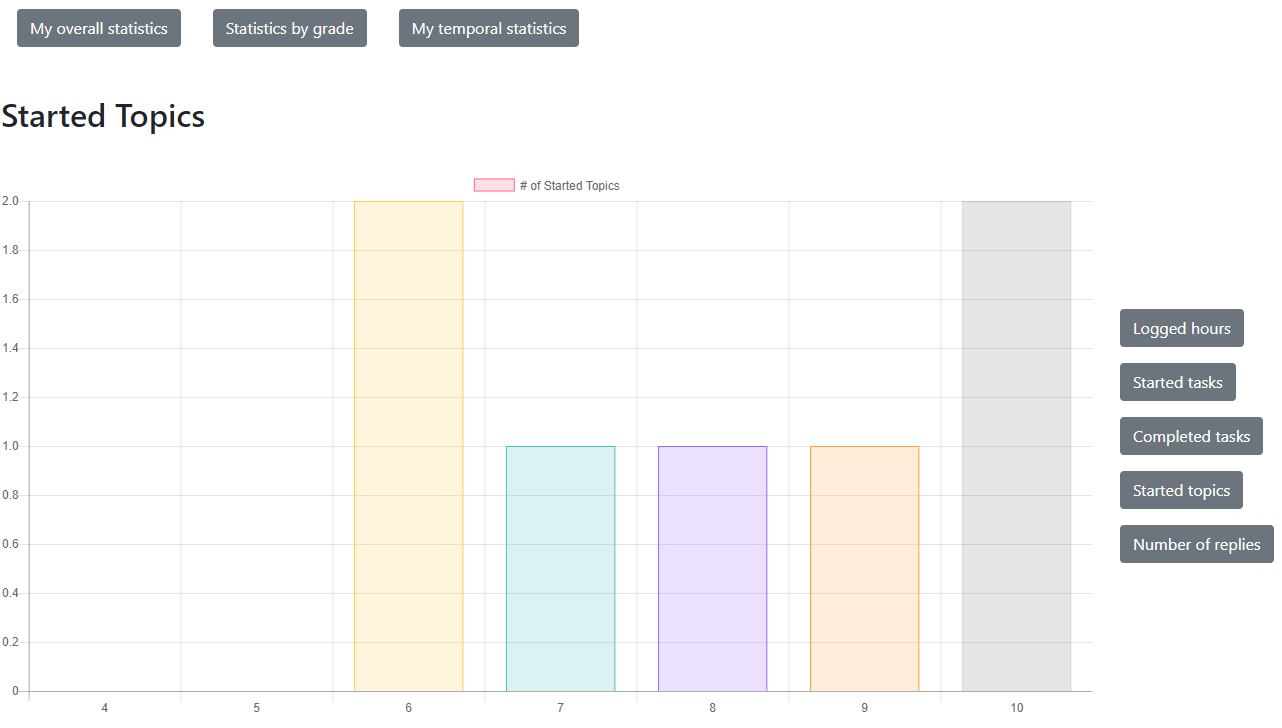
În al treilea mod le oferă o viziune generală asupra comportamentului lor și a felului cum acesta corelează cu notele de la facultate, oferind statistici cu activitatea în timp a studentului și statistici legate de activitatea studenților în aplicație grupate în funcție de notele pe care aceștia le au la facultate. Această pagină poate fi vazută in fig 3. Desigur, neavând date reale, pentru scop demonstrativ, voi crea niște conturi false, le voi asigna câte o notă și voi face acțiuni aleatorii în aplicație. Rezultatul final nu va arata nici o corelație adevărata între cât de mult un utilizator a folosit aplicația și rezultatele de la facultate, scopul acestei parți din lucrare fiind de a arata că se poate analiza relația dintre gradul de folosire al aplicației și rezultatele utilizatorilor, iar dacă aplicația va fi folosită în o situație reală, atunci acest modul ar putea arăta dacă studenții care folosesc mai mult aplicația obțin rezultate mai bune la facultate.

Fig 3: Statistics page

Desigur pe lângă aceste trei moduri prin care aplicația își propune să faciliteze procesele de studiere ale studenților, ar mai putea exista nenumărate alte soluții și de aceea în lucrarea de licența mi-am propus să mă focusez asupra unei arhitecturi de back-end care să poată integra cu ușurința oricâte din aceste soluții și care să ofere un potențial foarte ridicat pentru ce se poate construi peste ea.

Din punct de vedere arhitectural aplicația este un sistem distribuit ce implementează paternul de event sourcing, care la baza reprezintă salvarea stării aplicației ca o serie de modificări, în acest patern fiind numite evenimente, iar pentru a accesa ultima stare a ei, este necesar iterarea prin toată colecția de evenimente de la lansarea aplicației și construirea ultimei stări a aplicației din ele.

Unul dintre avantajele salvării stării aplicației ca o serie de modificări este că aceste modificări pot fi văzute ca un history log, lucru care ne oferă posibilitatea să generăm noi modele din aceste evenimente, la orice moment, mai ales că ele corelează cu acțiunile domeniului său. Acest lucru poate fi util în special pentru generarea de statistici, deoarece history log-ul ne permite să facem analize a modului în care a fost folosită aplicația. Pentru a ilustra mai bine acest punct de vedere să luăm ca exemplu următoarea situație: avem un magazin online care rulează de 2 ani si ne-am gândit ca ar fi valoros pentru creșterea business-ului dacă am ști de câte ori a fost adăugat fiecare produs la “Favorite” de către utilizatori. Într-o aplicație care nu are un history log acest lucru nu va putea fi implementat decât de la acel moment înainte, însă într-o aplicație ce implementează event sourcing, avem acces la history log și nu numai că putem implementa acea funcționalitate din acel moment înainte dar putem face asta încă de la lansarea aplicației, lucru care ar putea aduce foarte multă valoare business-ului, deoarece ar putea analiza în mod similar din mai multe perspective tot ce sa întâmplat încă de la debutul aplicației.

Exista soluții deja implementate pentru a avea un history log în aplicație, una din ele fiind ApexSQL Log, care oferă posibilitatea de a vedea și a face replay la toate tranzacțiile și de a vedea un istoric complet cu rândurile schimbate, însă comparativ cu event sourcing, această soluție este mult mai greu de folosit deoarece istoricul care îl oferă, nu este legat de domeniul aplicației.

În această situație exemplul de mai sus cu magazinul online ar putea să lămurească și mai bine de ce paternul de event sourcing, deși mai greu de implementat inițial, pe termen lung oferă posibilități de dezvoltare mult mai mari decât soluțiile generale deja implementate care nu au avantajul de a fi legate de domeniul aplicației. Revenind la magazinul online, dacă am avea un history log de la o soluție gata implementata, cum ar fi ApexSQl Log, pentru a afla de câte ori a fost marcat un produs ca “favorit”, v-a trebui să ne uităm la istoricul tabelei unde sunt menținute relațiile dintre utilizatori și produse, care, putem presupune cu multă siguranță că are multe alte câmpuri, să ne uităm la istoricul rândurilor și să număram de câte ori un produs a fost marcat ca “favorit”, lucru care necesită multă procesare a datelor și o complexitate destul de mare, mai ales când scopul analizei este de a afla relații între mai multe acțiuni din domeniu. În cazul în care aplicația ar fi fost implementată prin event sourcing, sarcina de mai sus revine la simplul fapt de a număra câte evenimente de tipul “MarkProductAsFavourite” sunt pentru acel produs.

Aplicația implementată în această lucrare de licența este formată din doua componente, ceea ce o face sistem distribuit. O componentă inserează evenimentele în o baza de date non relaționala, atunci când se face update, iar la query această interoghează o bază de date relaționala unde este ținuta ultima stare a aplicației, care a fost construită de a doua componentă a aplicației care citește evenimentele din baza de date non relaționala și construiește ultima stare a aplicației în tabele temporare din baza de date relaționala. Paternul de event sourcing oferă niște avantaje sistemelor distribuite, unul dintre acestea fiind util în cazul în care una sau mai multe componente sunt down și anume acestea pot procesa evenimentele generate în timpul în care au fost down și să își reconstruiască starea aplicației ca și cum nimic nu s-ar fi întâmplat.

În implementarea acestei aplicații am ales să memorez evenimentele într-o bază de date non-relaționala, deoarece acestea au o structură flat și stocarea lor într-o structură care nu e bazată pe relații face ca query-urile să fie mai rapide. Baza de date non relaționala pe care am folosit-o se numește Raven Db și am ales-o pentru că este populara în aplicațiile dot Net.

# Capitolul II. Tehnologii folosite

Baze de date non relationale

O bază de date non relațională oferă un mecanism de stocare si interogare a datelor care nu este modelat în stilul clasic al relațiilor dintre tabele. Aceste baze de date au design-ul mai simplu, sunt mai ușor de scalat și oferă o disponibilitate mai mare. În general motivul pentru care oferă o disponibilitate mai mare este pentru că este compromit consistența datelor pentru această

disponibilitate. Mai multe detalii despre aceste tipuri de baze de date se poate găsi in Anexa 1.

RavenDb

După o perioadă de cercetări în care am făcut research ca să îmi dau seama care ar fi cea mai potrivită bază de date non-relațională, am ajuns la concluzia că RavenDb este cea mai potrivită pentru proiectul meu deoarece oferă mai multe funcționalități care sunt oferite și de bazele de date tradiționale cum ar fi de exemplu tranzacțiile. De asemenea deși nu era cea mai populara bază de date (aceasta fiind MongoDb) și nu avea cel mai mare suport, ea poate fi folosită ușor pe platforma de dezvoltare .net, având pe site-ul lor o documentație foarte bine dezvoltată pentru utilizarea acesteia in .Net. Pentru mai multe detalii despre RavenDb și modul acesteia de funcționare consultați Anexa 1.

Fig 4. RavenDb Icon

.Net core and Asp.Net

.Net Core este o platforma open source pentru dezvoltarea aplicatiilor desktop, web sau mobile. .Net Core este o versiune îmbunătățita la .net, rescrisă de la zero și cross platform. Comparativ cu .Net, acesta oferă dezvoltatorilor o mulțime mai restrânsa de pachete care sunt incluse in proiect by default, lăsând la alegerea lui sa includă pachetele de care are nevoie, în felul acesta dimensiunea proiectul final fiind mai mica și incluzând doar assembly-urile de care aplicația are nevoie. Asp.Net este un framework construit peste .Net Core folosit pentru dezvoltarea aplicațiilor web oferind funcționalitățile de baza a unui server, posibilitatea de a integra codul C# cu cel Html și oferind mecanisme de model binding ce ușurează munca programatorului. In anexa 2 se găsește o descriere mai pe larg a funcționalităților oferite de acestea.

# Capitolul III. Detalii teoretice

Sisteme distribuite

Sistemele distribuite de software reprezintă o mulțime de componente software care comunică între ele pentru a atinge un scop comun. Beneficiile folosirii sistemelor distribuite constau în capacitatea de specializare a componentelor individuale asupra tipului de task pe care trebuie să îl îndeplinească. În sistemele distribuite aplicațiile sunt împărțite în componente mai mici lucru care încurajează reutilizarea componentelor deja existente și îmbunătățește în acest mod procesul de dezvoltare al softwarelui.

Event Sourcing în sisteme distribuite

Având în vedere că sistemele distribuite se bazează pe comunicarea între componentele din care sunt formate, event sourcing poate fi un pattern foarte potrivit pentru a implementa în aceste sisteme. Când o componentă din un sistem distribuit trebuie să îi transmită informații unei alte componente, acest lucru se poate face prin publicarea și consumarea de evenimente. Dacă evenimentele sunt salvate în o bază de date ar putea constitui un avantaj deoarece în acest mod dacă o componentă nu funcționează pentru o perioadă de timp, aceasta poate parcurge evenimentele pierdute și ajunge la starea la care trebuia să fie dacă ar fi funcționat tot timpul.

Analiza datelor din evenimente

Dacă aceste evenimente sunt persistate într-o bază de date atunci pot fi folosite pentru a analiza și crea diferite statistici referitoare la modul în care aplicația e folosită sau asupra datelor care sunt introduse în ea. Aceste evenimente din baza de date pot fi văzute ca o istorie completă a tuturor modificărilor asupra datelor din aplicație, nu doar ultima stare a datelor, ca la majoritatea aplicațiilor, iar acest lucru permite crearea unor statistici complexe care să ofere o viziune mai buna asupra domeniului aplicatiei, cum e folosita si cum ar putea fi imbunatatita pentru a maximiza serviciile pe care aceasta le ofera.

RDBMS vs. NoSQL: pentru ce tip de date sunt potrivite

Nu se poate spune ca baza de date non-relațională este mai bună decât cea relațională, ci doar că fiecare tip de bază de date este mai potrivită pentru anumite probleme. Bazele de date non-relaționale aduc beneficiul că modelul datelor este nestructurat ceea ce poate oferi o scalabilitate mare aplicațiilor mai ales în faza inițiala când schema datelor nu este bine stabilită sau urmează să fie supusă unor schimbări. În general bazele de date NoSql sunt mai potrivite pentru datele care în o structura relațională ar fi salvate în mai multe tabele și care ar necesita operații de join la interogare, care sunt costisitoare la timp. O analiză mai buna asupra diferenței dintre bazele de date relaționale și cele NoSql se poate găsi in Anexa 3 de la finalul documentației.

Tipuri de aplicații potrivite pentru această arhitectură

Deoarece această arhitectură poate aduce în unele situații avantaje iar în altele dezavantaje, ea este folosită pentru tipurile de aplicații în care avantajele întrec dezavantajele. Dintre aplicațiile cărora această arhitectură aduce beneficii fac parte: cele care au nevoie de scalabilitate mare, cele care au nevoie de un istoric complet al modificărilor aici întrând aplicațiile de tip economic ca cele de contabilitate sau sistem de gestionare al băncilor unde fiecare tranzacție trebuie să fie stocată din motive de securitate sau sistemele de contabilitate.

Salvarea fiecărei modificări sub forma de eveniment este o arhitectură greu de implementat și întreținut, de aceea această arhitectură este folosită doar în aplicațiile cu nevoi speciale.

Tipuri de aplicații nepotrivite pentru această arhitectură

Această arhitectură odată implementată vine cu o anumită serie de beneficii, însă sunt aplicații în care aceste beneficii nu își au rostul sau ar fi prea mici ca să justifice efortul implementării arhitecturii distribuite. În această categorie intră aplicațiile de dimensiuni mici, care nu au nevoie de multe resurse și nu sunt proiectate pentru a fi scalabile sau aplicații care au o structură si un flow simplu, pentru care introducerea componentelor intermediare nu ar face decât să îngreuneze timpul de procesare.

Sistemele distribuite

Sistemele distribuite nu sunt o soluție perfectă pentru fiecare problemă care există. Pe lângă avantaje, această arhitectură vine și cu o serie de dezavantaje, care în anumite situații ar putea întrece avantajele oferite. De exemplu dacă vorbim de o aplicație cu structură simplă, atunci implementarea sa ca un sistem distribuit nu ar fi chiar potrivită deoarece procesul inițial de dezvoltare necesita mai multe resurse, iar comunicare între componente ar putea mări nenecesar timpul de procesare a sarcinilor.

Pe de altă parte, atunci când structura aplicației este complexă, implementarea acesteia ca un sistem distribuit ar putea aduce multe avantaje. În primul rând, cel mai mare beneficiu ar fi scalabilitatea: componentele existente în sistem sunt deja făcute să comunice cu alte sisteme pentru a-și face treaba iar adăugarea unei noi componente se poate face folosind interfețele de comunicare deja existente cu celelalte componente ale sistemului. Apoi pe lângă scalabilitate, sistemele distribuite vin cu o serie de beneficii, care în anumite situații critice, pot face o diferență foarte mare:

* posibilitatea de a scala doar componenta cea mai slaba sau cea mai folosită
* integrarea mai ușoară în sisteme distribuite deja existente
* posibilitatea utilizării de componente deja existente
* rularea fiecărei componente pe servere specific adaptate pentru acel tip de procesare
* posibilitatea utilizării unui limbaj de programare specializat pentru diferitele componente (de ex c++ pentru partea de procesare și python pentru un server web)

Sistemele distribuite au continuat să evolueze și să fie folosite pentru rezolvarea a cât mai multor probleme astfel încât acestea au început sa fie clasificate în anumite tipuri. O explicare detaliate a acestor tipuri poate fi găsită în anexa 4.

Event sourcing

Martin Fowler a răspândit această arhitectură sub numele de event sourcing. În esența event sourcing înseamnă salvarea stării aplicației ca o serie de schimbări iar pentru a accesa ultima stare a datelor trebuie sa parcurgem aceste schimbări si să o construim. Salvarea sub formă de schimbări a stării aplicației aduce de la sine un istoric cu tot ce sa întâmplat în aplicație lucru care este mai apropriat de modul de gândire al oamenilor, deoarece trecutul ne oferă informații importante, iar când luăm o decizie, nu o luăm doar din prisma prezentului, ci și a trecutului.

Modul acesta de salvare a datelor este o reprezentare mult mai exactă a realității, iar din acest motiv schimbările naturale din domeniul aplicației nu vor veni ca un șoc asupra implementării și nu vor fi necesare decât schimbări minore pentru a recalibra aplicația cu domeniul ei.

O descriere a lui Martin Fowler a event sourcing-ului poate fi găsită in anexa 5.

Paternul CQRS forțează separarea modelelor pentru modificări de cele pentru citire a datelor. Structurarea proiectului în acest fel respecta Single Responsability Principle care spune ca fiecare clasă ar trebui să aibă o singură responsabilitate, acest lucru fiind folositor deoarece atunci când trebuie să schimbăm ceva la aplicație, știm exact unde trebuie sa ne uităm.

Motivul pentru care am ales sa folosesc acest patern în implementarea aplicației este că se potrivește foarte bine cu Event Sourcing, modelul pentru modificări din CQRS fiind comanda din event sourcing din care se generează evenimentul. Acest patern mai are avantajul că în modele vor exista strict datele de care este nevoie, eficientizând în acest fel transmiterea datelor din o parte a aplicației in alta. În figura 5 de mai jos se poate vedea o diagrama a paternului CQRS.

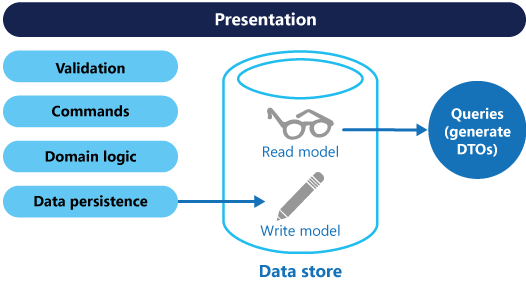
 O descriere mai pe larg a acestui patern se găsește in anexa 6.

Fig 5: CQRS diagram.

[Sursă imagine](https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs)

# Capitolul IV. Implementarea aplicației

Framework-ul CQRSlite

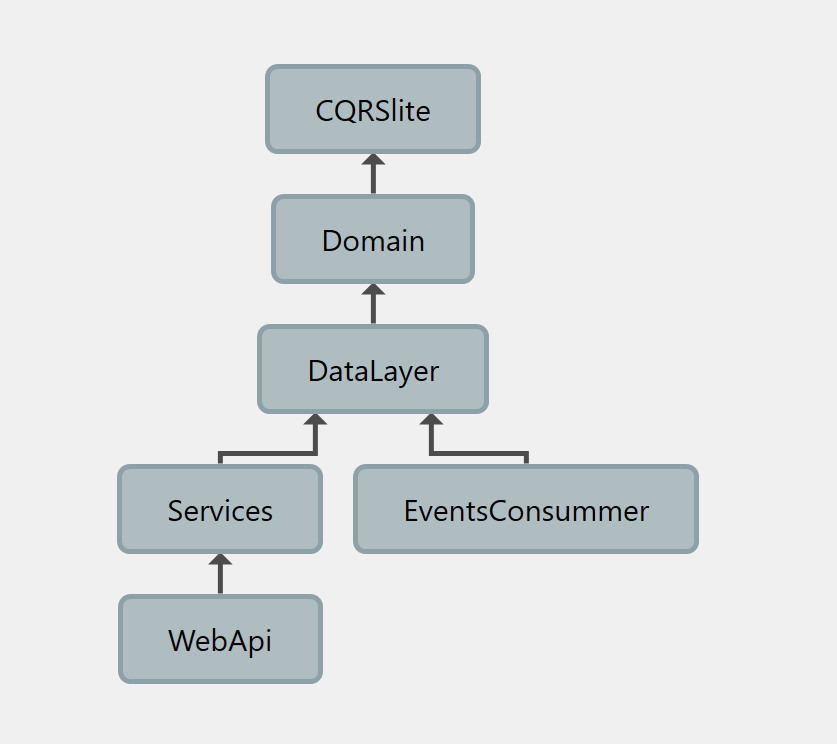
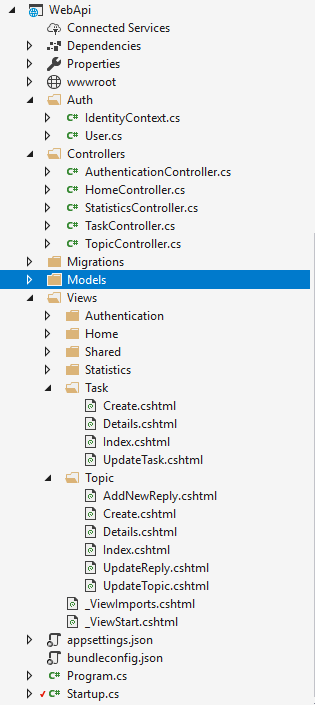
În implementarea aplicației am folosit framework-ul CQRSlite care oferă un set de interfețe și funcționalități comune pentru a ușura munca implementării unei aplicații prin acest pattern. Acest framework pune la dispoziție utilizatorului funcționalități de trimitere comenzi și publicare de evenimente, posibilitatea de a lucra cu snapshoturi și rezolva problema concurentei adăugând în mod automat un time stamp și o versiune fiecărui eveniment. În paternul de event sourcing evenimentele sunt procesate pentru a ajunge la ultima stare a aplicației, lucru care ar putea dura destul de mult când sunt multe evenimente de procesat iar pentru a rezolva problemă din aceasta situație intervine conceptul de snapshotting. Prin snapshoting se salvează o anumită stare intermediara a aplicației iar evenimentele se procesează doar de la acel snapshot înainte, mărind în acest fel viteza aplicației atunci când sunt foarte multe evenimente de procesat.

Fig 6: Diagrama proiectelor

În figura 6 se poate vedea diagrama proiectelor, iar în continuare urmează o descriere sumară a fiecăruia și rolul pe care îl are joacă în sistemul distribuit.

## Proiectul Web Application



Proiectul WebApplication este de tipul ASP.NET CORE MVC. În el sunt controllerele și viewurile. Deși modelele folosite pentru comunicarea între viewuri și controllere (ViewModels) sunt în mod normal în acest proiect, în cazul de fata sunt în proiectul de Servicii deoarece arhitectura CQRS permite ca serviciile să întoarcă doar strictul necesar de care au nevoie viewr-uile, nemaifiind nevoie de modele în proiectul WebApi care să filtreze informațiile primite de la servicii. Structura acestui proiect poate fi văzută in figura 7 alăturată.

De asemenea în acest proiect sunt înregistrate și configurate majoritatea funcționalităților de bază din .net core sau pachetele nuget ca librăria Asp.Net Identity folosită la autentificare. Acest proiect, referențiază direct sau indirect, toate celelalte proiecte din aplicația web și tot aici este configurat și sistemul de Dependecy Injection care este folosit și in celelalte proiecte. Dependecy Injection este un mecanism de instanțiile automată a claselor de către framework, atunci când e nevoie de ele, și odată configurat, știe automat ce parametri trebuie să transmită constructorilor pentru a instanția acea clasa. In proiectul Asp.Net Core Web Application este folosit pentru a instanția clasele de controllere atunci când vine un request de la client pentru o anumită pagina web.

Fig 7: Structura proiectului Web Application

Instanțierea claselor putându-se face în mod recursiv, de exemplu dacă controller-ul are nevoie de un repository pentru a comunica cu baza de date, iar la rândul său repository-ul are nevoie de un obiect de tipul ApplicationContext, mecanismul de Dependency Injection stie să creeze prima oara un obiect de tipul ApplicationContext, apoi să apeleze constructorul de la repository trimițându-i obiectul de tipul ApplicationContext, iar după ce a creat repository-ul instanțiala controller-ul , care a fost obiectul cerut inițial și îi pasează acestuia obiectul de care are nevoie. Atâta timp cât aceste dependințe între obiecte sunt înregistrate în fișierul Startup.cs mecanismul știe automat să le instanțieze recursiv.

Mecanismul de Dependency Injection nu este folosit doar în contextul aplicațiilor web, unde este necesară instantaneea obiectelor în mod dinamic atunci când un request vine la server. Acest mecanism poate fi creat și în aplicațiile consola când exista obiecte ce au nevoie de un numar de parametri foarte mare sau care au nevoie de alte obiecte care au un lanț foarte lung de dependințe. De asemenea mecanismul mai este util și în proiectele de testare, în special în porțiunile unde sunt testele de integrare, pentru a simula cât mai exact condițiile reale în care va funcționa sistemul sau pentru a ușura munca creării obiectelor asupra cărora se fac teste, acestea fiind necesare în fiecare funcție de testare și care ar crea foarte mult cod duplicat de instanțiile a obiectelor cu același parametri, sau cu mici diferențe în parametrii de plecare, lucru care ar încălca principiul DRY.

Tot în acest proiect se realizează și procesul de autentificare care e implementat folosind pachetul Microsoft.AspNetCore.Identity, o soluție ce oferă implementări pentru majoritatea funcționalităților de autentificare și autorizare și care poate fi folosit doar prin adăugarea unor setări de configurare în fișierul Startup.cs. Acest pachet folosește un context diferit pentru a se conecta la baza de date, și creează automat niște tabele care le va folosi pentru a memora datele despre utilizatori, cum ar fi numele, emailul sau rolurile acestora în aplicație.

În momentul când un utilizator este autentificat, serverul generează un cookie ce conține un token de autentificare și un token anti forgery care este folosit pentru a verifica că tokenul de autentificare nu a fost falsificat (previne atacurile de tipul cross site scripting). Acest cookie este salvat atât pe client, cât și pe server, iar de fiecare dată când acel utilizator face un request către server, este trimis și cooki-ul de autentificare și este comparat cu cel de pe server pentru a verifica dacă requestul provine într-adevăr de la un utilizator logat în aplicație.

Modele folosite pentru comunicarea între controllere și view-uri respectă paternul de CQRS, modelele pentru post/update/delete derivând din clasa ICommand iar modelele de rezultat pentru query derivând din clasa IQueryResult. Paternul CQRS are avantajul că permite simplificarea obiectelor prin care se transferă date între controller și view-uri folosind câte un obiect separat pentru fiecare request, în acest fel fiecare putând avea strict datele necesare requestului, rețeaua nefiind încărcata cu proprietăți care ar fi fost utile altor requesturi, însă care, în cazul de fata nu folosesc la nimic.

Pe partea de citire, când se face un request pentru o pagină, pentru obținerea datelor, este creat un obiect ce deriva din Interfata IQuery, și este pasat unui obiect de tipul query handler localizată în proiectul Services, care știe să aducă datele necesare. După ce datele au fost procurate controller-ul returnează view-ul corespunzător paginii cerute pasându-i modelul cu datele cerute. Din nou, datorită arhitecturii CQRS toate modelele pasate la view-uri derivă din aceeasi interfața IQueryResult, nemai fiind nevoie de obiecte ViewModel, deoarece rezultatele întoarse de query handler conțin întotdeauna doar datele necesare pentru a fi afișate pe acea pagina.

Partea de scriere, este similară cu partea de citire, în sensul că de fiecare data la controller este pasata o clasă ce derivă din interfața ICommand, și aceasta este pasată mai departe unui command handler, similaritatea dintre partea de citire și cea de scriere fiind ca întotdeauna este trimis către proiectul de servicii un obiect ce deriva ori din interfața ICommand ori din IQuery către un query, respectiv command handler care are responsabilitatea ori de a întoarce datele cerute, ori de a aduce modificări datelor existente. După ce obiectul de tip command handler și-a terminat treaba, acesta redirectează la o pagină unde se poate vedea rezultatul acțiunii făcute. Spre exemplu, după ce un utilizator a creat un task, acesta este redirectat către pagina cu lista de task-uri, unde poate vedea noul său task.

În viewrile razor am folosit frameworkul Bootstrap pentru stilizare și cod c# combinat cu HTML și JavaScript pentru redacția lor. Un avantaj care îl oferă Razor view-urile este că poate oferi model binding automat, prin niște tag-uri HTML speciale și specifice razorului definite de către cei de la Microsoft “Tag Helpers”, care se folosesc de cod C# pentru a randa elemente HTML in view-ruile razor în momentul când acestea sunt transformate în fișier .html și trimis către clientul care a făcut cererea. Exista multe Tag Helpere pe care asp.net core le oferă, pentru funcționalități comune, cum ar fi crearea formularelor, link-uri sau încărcarea resurselor.

View-ruile mi-au simplificat dezvoltarea parții de front-end a aplicației oferind funcționalități ajutătoare după modelul celor folosite în framework-urile mai populare pentru dezvoltarea parții de ușer interfaze, cum ar fi angular.

Cel mai ajutător lucru mi s-a părut funcționalitatea de model binding, care știe sa mapeze automat datele din pagina la elementele HTML, atât la primirea paginii, cat si la postarea formularelor sau accesarea link-urilor cu anumiți parametri din model. Tot ce trebuie sa faci este sa folosești Tag Helpere speciale în care să specifici ce elemente HTML să afișeze pentru field-urile primite în model.

Mai jos în figura 8 se poate vedea un exemplu cu formularul folosit pentru updatarea unui task. Atât pentru preluarea datelor inițiale pentru a se completa în formularul de editare cat și pentru trimiterea lor către serverul de back end, nu am fost nevoit sa scriu javascript deloc, toate aceste lucruri făcând-se automat prin transmiterea modelului din controller către view-ul pentru pagina de updatare a task-ului și prin tag helperele de forma “input asp-for=’Title’”. In momentul când clientul face request la server către o pagină, serverul transformă Tag Helperele și tot codul c# in cod HTML și JavaScript static. Pe client Tag helper-ul “form asp-action=’UpdateTask’”, va fi transformat în elemente HTML simple și cod JavaScript pentru a trimite înapoi la back-end datele formularului prin metoda POST, după ce au fost modificate.

|  |
| --- |
| <div class="row">  <div class="col-md-4">  <**form** **asp-action**="UpdateTask">  <**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>  <**input** **type**="hidden" **asp-for**="AggregateId" />  <div class="form-group">  <**label** **asp-for**="Title" class="control-label"></**label**>  <**input** **asp-for**="Title" class="form-control" />  <**span** **asp-validation-for**="Title" class="text-danger"></**span**>  </div>  <div class="form-group">  <**label** **asp-for**="Content" class="control-label"></**label**>  <**input** **asp-for**="Content" class="form-control" />  <**span** **asp-validation-for**="Content" class="text-danger"></**span**>  </div>  <div class="form-group">  <**label** **asp-for**="Hours" class="control-label"></**label**>  <**input** **asp-for**="Hours" class="form-control" />  <**span** **asp-validation-for**="Hours" class="text-danger"></**span**>  </div>  <div class="form-group">  <input type="submit" value="Save" class="btn btn-primary" />  </div>  </**form**>  </div>  </div> |

Fig 8: Formular update task

În folderul Controllers, găsim următoarele 3 Controllere, toate de tipul MVC:

* AuthenticationController ce conține paginile de Login și Register. Metodele din acest controller sunt decorate cu verbe HTTP de tipul GET, POST, PUT sau DELETE
* TaskController ce conține metode ce întorc view-uri corespunzătoare fiecărei pagini legate de secțiunea Task, ca cea cu lista de task-uri, pentru detalii, crearea unui task nou sau updatarea acestuia. La fel ca la AuthenticationController, și metodele de aici sunt decorate cu verbe HTTP
* TopicController ce conține metode ce întorc view-uri corespunzătoare fiecărei pagini legate de secțiunea Topics, cum ar fi lista de topic-uri, pagina detaliată a topicului cu toate răspunsurile sale sau paginile de creare, updatare și adăugare de noi răspunsuri. Pentru a respecta standardele de dezvoltare a aplicațiilor Asp.Net Mvc și aceste metode sunt decorate cu verbe HTTP
* StatisticsController responsabil cu toate paginile din Tab-ul de Statistici ale aplicației. Acest controller este puțin diferit de celelalte trei în sensul că conține doar metode decorate cu verbul GET, ceea ce înseamnă că această secțiune a aplicației este doar informaționala și utilizatorii nu pot modifica nimic din ea, în mod direct, ci doar indirect folosind aplicația, făcând cât mai multe acțiuni, pentru a putea fi analizate și afișat raportul în această secțiune.

Înafara de cel de autentificare, fiecare controller este decorat cu atributul Authorize, ca în figura 9, care spune pachetului Identity că pe paginile gestionate de controllerul respectiv pot avea acces doar utilizatorii care au fost logați.

|  |
| --- |
| [Authorize(AuthenticationSchemes = CookieAuthenticationDefaults.AuthenticationScheme)]  public class TaskController : Controller  {  private ISession \_session { get; }  private IViewSincronizor \_viewSincronizor { get; } |

Fig 9: Autorizare in controllere

După cum am menționat și mai sus, când un utilizator face o acțiune în interfața care trebuie să modifice datele, partea de client trimite la back-end un POST, ce conține în body un obiect de tipul comanda ce conține modificările pe care le-a lansat utilizatorul din interfața. Aceasta comanda este mai apoi trimisă către un command handler care va crea un eveniment din ea, iar la crearea evenimentului acesta are nevoie să știe ce utilizator a făcut aceasta acțiune.

Pentru a lega fiecare comandă de un utilizator, în controllere, fiecare metoda care primește o comanda, completează în aceasta numele utilizatorului pe care îl obține din proprietatea “User.Identity.Name”. Această proprietate este dată în mod automat de componenta Identity, care știe să lege utilizatorul logat pe front end de fiecare request făcut la back-end.

Secțiunea de statistici a aplicației conține foarte multe grafice. Acestea sunt realizate cu ajutorul librăriei Chart.js. Aceasta este o librărie puternică folosită pentru vizualizarea datelor care m-a ajutat să obțin grafice care să arate foarte bine, cu putină munca. Pentru a crea un grafic, trebuie doar să te uiți pe site-ul lor, la exemplele de grafice, alegi modelul potrivit și copii codul pentru respectivul grafic în pagina de html. În figura 10 de mai jos se poate vedea un exemplu de cod JavaScript pe care l-am folosit în paginile HTML pentru unul din graficele din secțiunea de statistici:

|  |
| --- |
| <script>  var ctx = document.getElementById("myChart");  var myChart = new Chart(ctx, {  type: 'bar',  data: {  labels: @Html.Raw(Json.Serialize(grades)),  datasets: [{  label: '# of Logged Hours',  data: @Html.Raw(Json.Serialize(nrOfLoggedHours)),  backgroundColor: [  'rgba(255, 99, 132, 0.2)',  'rgba(54, 162, 235, 0.2)',  'rgba(255, 206, 86, 0.2)',  'rgba(75, 192, 192, 0.2)',  'rgba(153, 102, 255, 0.2)',  'rgba(255, 159, 64, 0.2)'  ],  borderColor: [  'rgba(255,99,132,1)',  'rgba(54, 162, 235, 1)',  'rgba(255, 206, 86, 1)',  'rgba(75, 192, 192, 1)',  'rgba(153, 102, 255, 1)',  'rgba(255, 159, 64, 1)'  ],  borderWidth: 1  }]  },  options: {  scales: {  yAxes: [{  ticks: {  beginAtZero: true  }  }]  }  }  }); |

Fig 10: Exemplu de cod Javascript pentru afisarea graficelor

Folosind această librărie, creare unui grafic este redusă la alegerea unui tip, cum ar fi “bar chart”, trimiterea modelului de date pe care graficul îl va modela, setarea culorilor folosite și opțional coordonatele de la care să înceapă desenarea graficului și dimensiunile acestuia.

Un alt lucru interesant de povestit despre acest proiect este felul în care sunt structurate view-urile Razor. Atunci când creezi un proiect Asp.Net MVC acesta are gata create niște exemple de pagini care sunt create după un standard ce ușurează munca proiectării și dezvoltării paginilor web.

Aceste pagini generate prin mecanismul automat de scaffolding respectă principiul DRY, în sensul că porțiunile comune ale paginilor web cum ar fi secțiunile de header, ce conțin meniul principal, sau secțiunea de footer, sunt salvate în fișiere de sine stătătoare și nu trebuie rescrise în fiecare pagină a aplicației. Acest lucru ușurează munca programatorilor deoarece nu mai trebuie să scrie așa de mult cod, dar și eficientizează procesul de dezvoltare deoarece oferă o structură mai buna de baza a fișierelor cshtml, pe care se poate dezvolta mult mai ușor.

Un ultim lucru de menționat legat de proiectul WebApplication este că în el se află un fișier de configurare “appsettings.json” în care sunt trecute de obicei setările pentru mediul în care proiectul să ruleze. Avantajul de a trece aceste setări în acest fișier, este că “appsettings.json”, nu este compilat și transformat într-un fișier dll, ci este copiat în folderul de output al compilarii fișierelor sursă, ca fișier cu extensia json și poate fi modificat în timp real, pentru a schimba setările de configurare ale aplicației, fără a mai fi necesară recompilarea surselor.

În figura 11 de mai jos poate fi văzut fișierul de configurare din proiectul meu, unde am stocat connection string-ul la baza de date sql și datele de conectare pentru baza de date RavenDB.

|  |
| --- |
| {  "Raven": {  "Url": "http://localhost:8080",  "Database": "Licenta2018"  },  "Logging": {  "IncludeScopes": false,  "LogLevel": {  "Default": "Warning"  }  },  "ConnectionStrings": {  "DefaultConnection": "Data Source=DESKTOP-P6BH1QB\\SQLEXPRESS;Initial Catalog=Licenta2018;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailover=False"  }  } |

Fig 11: Fișierul appsettings.json

## Proiectul Services

Proiectul de services este un proiect crucial în aplicație în special pentru partea de write din CQRS. Aici se găsește implementarea pentru una din cele mai importante parți ale aplicației și anume primirea comenzii de la controllere și generarea, din datele oferite de aceasta, un eveniment legat de aggregatul pe care sa făcut modificarea, care să fie inserat în baza de date.

Tot aici se face separarea între comenzi și query-uri, acest proiect fiind împărțit în două subfoldere „Commands” și „Queries”. Folderul Commands este împărțit la rândul sau în subfolderele, Task și Topic care reprezintă entitățile de baza cu care lucrează aplicația, pentru care se pot primi comenzi și genera evenimente prin command handler (spre deosebire de entitățile de la secțiunea de statistici, care sunt read-only și se găsesc doar în folderul de „Queries”).

Atât componentele de tasks cât și topics au fost implementate folosind o structura similara, în sensul ca aceste foldere conțin comenzile ce corelează cu acțiunile pe care le poate face un utilizator legat de acea entitate în interfața și un command handler care știe să prelucreze acea comanda și să genereze un eveniment din ea, care cu ajutorul framework-ului CQRSlite, îl leagă de un anumit aggregat și îl inserează în baza de date. În folderul Task putem găsi comenzile și handlerele ce prelucrează aceste comenzi:

* CompleteTaskCommand - comanda pentru marcarea unui task ca fiind completat
* ReopenTaskCommand - comanda folosită pentru redeschiderea unui task
* CreateTaskCommand – comanda ce conține datele inițiale despre un anumit task
* UpdateTaskCommand – comanda ce conține versiunea actualizată a descrierii, titlului și a numărului de ore estimat pentru terminarea acelui task
* LogTaskHoursCommand – comanda trimisă când utilizatorul logheaza ore pe un task
* TaskCommandHander – clasa principala care se ocupă cu procesare comenzilor de mai sus, având cate o metodă „Handle” pentru fiecare comandă legată de entitatea Task

Comenzile derivează din interfața marker „ICommand” care nu conține proprietăți sau metode iar TaskCommandHandler derivă din mai multe interfețe „ICommandHandller< ICommand>”, obligând clasa să implementeze metoda „Handle(ICommand command)”. Clasele ce derivă din interfața „ICommand” sunt clase a căror scop este doar de a face transferul de date între front-end și back-end, ele neavând nici o metodă înafara de constructori.

Clasele ce derivă din interfața „ICommandHandller<ICommand>” sunt la polul opus al comenzilor, în sensul că acestea nu conțin deloc proprietăți cu date ce urmează a fi inserate în baza de date, ele conținând doar metodele handle, pentru fiecare tip de comandă, și niște obiecte folosite pentru a insera evenimentul generat în baza de date.

Înainte de a trece mai departe la explicațiile despre cum am implementat în servicii, procesul de generare a unui eveniment dintr-o comanda, trebuie să explic puțin obiectul numit sesiune în framework-ul CQRSlite, cum leagă el evenimentele de un agregat și etapele prin care trece când adaugă un eveniment în baza de date pentru un agregat existent.

În event sourcing evenimentele legate de o entitate, reprezintă toate seria de modificări prin care a trecut acea entitate. Termenul folosit pentru a identifica acea entitate în event sourcing este de agregat, din acest motiv, toate evenimentele trebuie să aibă o proprietate numita AggregateId, care este folosit ca modalitate de a lega modificările din eveniment de entitatea respectivă.

Pentru comunicarea cu baza de date am folosit obiectul „Session”, descris mai sus, din framework-ul CQRSlite. Pe lângă proprietatea AggregateId din eveniment, pentru a putea fi inserat în baza de date, acesta trebuie să fie legat și în mod abstract de acel agregat. Din această cauză sesiunea lucrează cu obiecte ce derivă din clasa abstractă AggregateRoot. Iar modul cum a fost gândit framework-ul este că pentru fiecare acțiune care o poate face utilizatorul în interfața pentru acea entitate, în clasa ce implementează AggregateRoot, pentru acea entitate, trebuie să existe o metodă, care să aibă ca nume descrierea acțiunii, iar în interiorul ei trebuie să se creeze evenimentul cu datele primite din comanda de la controller.

În felul acesta framework-ul CQRSlite forțează developerul să lege evenimentele atât prin proprietatea AggregateId cât și prin generarea acestuia care se face în clasa AggregateRoot, de agregatul de care aparține. În mod similar framework-ul CQRSlite forțează dezvoltatorii în mai multe circumstanțe să respecte paternul de Event Sourcing combinat cu CQRS cat mai bine.

Acum că am trecut în revistă câteva detalii despre clasa „Session”, modul de implementare a acesteia și motivul pentru care a fost aleasă aceasta abordare putem trece mai departe la a explica procesul prin care o comandă este transformată în un eveniment și diferitele scenarii care pot avea loc.

În metodele handle din command handler este folosit framework-ul CQRSlite. Aceste metode pot aborda, în general, doua tipuri de implementări, în funcție de ce fel de comanda a primit:

* Dacă comanda primită este pentru a actualiza o entitate deja existentă în baza de date, avem nevoie să creăm un obiect ce derivă din AggregateRoot care identifică în mod unic agregatul de care trebuie să legăm comanda. Pentru a face acest lucru trebuie să interogam sesiunea trimițând-ui ca parametru un AggregateId, iar această acțiune ne va returna instanța clasei ce identifică în mod unic acel aggregat și care conține câte o metodă pentru fiecare comandă pe care o poate face utilizatorul pentru acel tip de entitate. Acum că avem obiectul, singurul lucru ce mai trebuie făcut este să apelam metoda corespunzătoare comenzii respective, trimițând ca parametri datele din acea comandă. În metodă se va crea un eveniment din datele primite ca parametru și se va apela metoda generică ApplyEvent pentru evenimentul nou creat, care îl va insera în baza de date.
* Dacă comanda primită este pentru a crea o nouă entitate, atunci lucrurile stau mai simplu. În acest caz nu mai trebuie să interogam sesiunea pentru acel obiect care identifică unic acel agregat, deoarece agregatul este inexistent și trebuie să îl creăm. Pentru a-l crea singurul lucru care trebuie să îl facem este să instanțiem clasa ce deriva din AggregateRoot pentru acel tip de entitate și să îi trimitem ca parametri în constructor, datele din comandă. În constructor se va apela din nou metoda generică ApplyEvent care primește ca parametru un eveniment creat din datele primite de la command handler, însă fiind în constructor, va fi apelat doar cu evenimente pentru crearea unor noi entități. După ce am apelat noul constructor, trebuie să adăugam evenimentul în sesiune apelând o metoda de genul “await \_session.Add(task);”, iar apoi facem commit : “await \_session.Commit()”

În momentul în care am dat commit în sesiune, framework-ul în spate se va folosi de obiectul RavenEventStore pentru a insera evenimentul în baza de date. Clasa RavenEventStore derivă din interfața IEventStore care se găsește în framework-ul CQRSlite și este o interfață ce trebuie implementată de fiecare developer ce dorește să utilizeze acest framework, deoarece implementarea acesteia, depinde foarte mult de tipul de bazei de date folosită. În cazul meu am folosit RavenDB și a trebuit să implementez metode pentru a citi și scrie evenimente în baza de date, după documentația de pe site-ul bazei de date RavenDB.

O altă funcționalitate interesantă pe care o are framework-ul este aceea de a face reiterarea prin toate evenimentele în timp real când se inserează un nou eveniment. Motivul pentru care cei care au implementat framework-ul au procedat în acest fel este pentru a putea oferi developărilor funcționalitatea de a verifica că anumite reguli de business sunt îndeplinite înainte să insereze noul eveniment, fără a fi nevoiți să interogheze starea curenta a aplicației, care în mod normal e recomandat să fie ținuta separat de locul în care sunt salvate evenimentele.

Deoarece explicațiile de mai sus sunt puțin abstracte și greu de înțeles fără a vedea cod-ul efectiv, am adăugat mai jos în figura 12 niște exemple de cod din command handler (în partea stânga a figurii), respectiv din două metode ale clasei ce implementează AggregateRoot pentru entitatea de tip task (în partea dreaptă a figurii).

|  |  |
| --- | --- |
| public async System.Threading.Tasks.Task Handle(CreateTaskCommand command)  {  \_semaphoreSlim.Wait();  try  {  var task = new Domain.Task(Guid.NewGuid(), command.IssuedBy, command.Title, command.Content, command.Tags, command.Hours);  await \_session.Add(task);  await \_session.Commit();  }  finally  {  \_semaphoreSlim.Release();  }  }  public async System.Threading.Tasks.Task Handle(UpdateTaskCommand command)  {  await \_semaphoreSlim.WaitAsync();  try  {  var task = await \_session.Get<Domain.Task>(command.AggregateId);  task.UpdateTaskDetails(command.AggregateId, command.IssuedBy, command.Title, command.Content, command.Tags, command.Hours, command.LoggedHours, command.CompletedStatus);  await \_session.Commit();  }  finally  {  \_semaphoreSlim.Release();  }  } | public Task(Guid aggregateId, string issuedBy, string title, string content, List<Tag> tags, double hours)  {  Id = aggregateId;  ApplyChange(new TaskCreatedEvent(aggregateId, GetType(), issuedBy, title, content, hours,tags));  Completed = false;  }  public void UpdateTaskDetails(Guid aggregateId, string issuedBy, string title, string description, List<Tag> tags, double hours, double loggedHours,bool completedStatus)  {  ApplyChange(new TaskUpdatedEvent(aggregateId, GetType(), issuedBy, title, description, hours,loggedHours,completedStatus ));  } |

Fig 12: Exemplu cod partea de scriere

În figura 13 de mai jos poate fi văzut cum este declarată clasa TaskCommandHandler și din ce interfețe derivează:

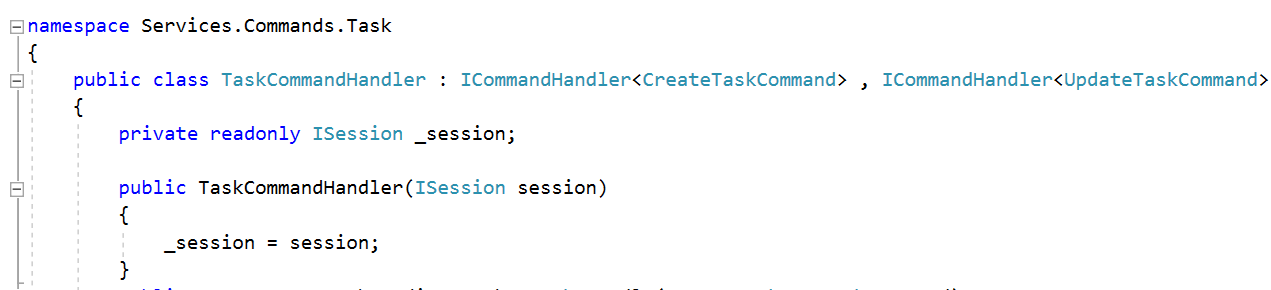


Fig 13: Declararea clasei TaskCommandHandler

În subfolderul Topic de la Commands găsim următoarele comenzi și handlere:

CreateTopicCommand - folosită la crearea unui nou topic; conține titlul și conținutul acestuia

UpdateTopicCommand – conține date similare cu comanda pentru crearea unui nou topic, însă se poate aplica numai daca topicul nu conține deja răspunsuri

AddNewReplyCommand – conține răspunsul, cine l-a oferit și data când a făcut-o

UpdateReplyCommand – similară cu comanda de adăugare a unui răspuns dar trebuie să fie verificat dacă acel reply există deja înainte de a se crea evenimentul de ReplyUpdated

TopicCommandHandler – la fel ca la command handler-ul pentru taskuri, acesta are metode „Handle” care pot primi ca parametri comenzi legate de entitatea topic

Tot în proiectul Services avem folderul queries care este organizat în subfoldere pe următoarele viewuri :

TaskListView – acest view conține informațiile afișate în pagina cu lista de taskuri cum ar fi Id-ul, titlul sau dacă taskul e terminat sau nu

TaskView – conține toate detaliile legate de un task și este folosit pe pagina de task details

TopicListView – conține informațiile necesare pentru pagina listei de topicuri

TopicView – conține toate informațiile unui topic : titlu, descriere și răspunsuri

Datele din aceste viewuri sunt completate de către aplicația ViewProcessorConsumer pe măsura ce procesează noile evenimente. În fiecare dintre subfulderele descrise mai sus se găsește o clasa care conține informațiile pe baza carora se face query-ul (de obicei id), o clasă care reprezintă modelul returnat de query și clasa handler care face cererea pentru respectivele date la repository. Cele 3 clase descrise mai sus derivă din IQuery, IQueryHandler și IQueryResult. Figura 14 exemplifică acest lucru mai bine.

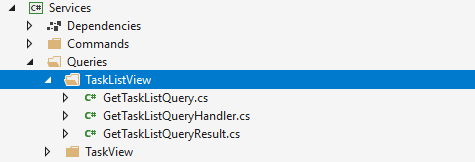


Fig 14: Structura modelelor de citire

## Proiectul Events Consummer

După cum știm, aplicația ce am implementat-o în această lucrare este un sistem distribuit. În acest sistem proiectul EventsConsummer este o aplicație de tip consola, de sine stătătoare, care rulează în background și procesează fiecare eveniment inserat în baza de date RavenDB, pentru a updata view-urile materializate.

Această aplicație consola joacă un rol crucial în sistemul distribuit, deoarece fără ea, nu am avea decât o mulțime de evenimente din care nu s-ar înțelege prea multe.

În acest proiect găsim clase numite Handlere, pentru fiecare view din baza de date sql, și care au rolul de a procesa fiecare eveniment și actualiza view-ul de care este răspunzător, pentru a fi sincronizat cu ultima stare a aplicației.

În această secțiune a aplicației, când spunem view, ne referim la un tip diferit de view decât cele din proiectul WebApi unde au rolul de a descrie browser-ului prin cshtml cum să arate pagina. În acest proiect un view reprezintă o anumită tabelă în baza de date sql, pe care sunt făcute interogări folosind modelele de query, de la arhitectura CQRS, pentru a putea afișa datele pe interfața utilizatorului.

Folosirea view-urilor de acest tip aduce un avantaj destul de important în aplicație: din cauză că tabelele din baza de date sql conțin exact datele care trebuie să se folosească pe pagina, query-urile vor fi mai rapide decât dacă ar fi trebuit să selectez doar anumite coloane din o tabela generică care este folosită în toate paginile sau dacă aș fi încărcat toată tabela, aducând în memorie multe date care nu doar ca nu vor fi folosite, ci vor și îngreuna procesarea și transmiterea lor din o etapa a aplicației în alta.

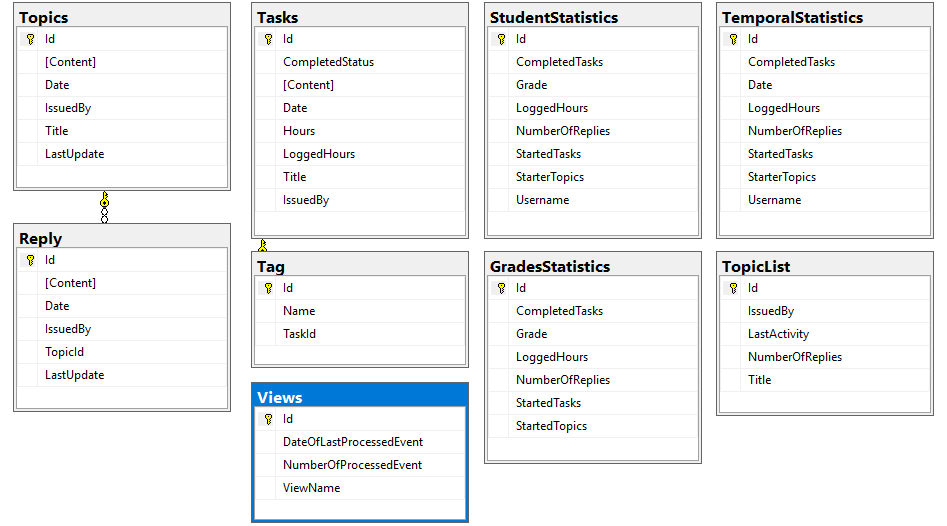
Pentru a ilustra punctul de mai sus putem lua ca exemplu pagina unde se afișează lista de taskuri. Această pagină are nevoie din baza de date doar de titlul taskului, id-ul și dacă este completat sau nu. Dar legat de entitatea task mai avem pagina unde se găsesc toate detaliile task-ului, cum ar fi descrierea, numărul de ore estimate pentru a termina acel task sau numărul de ore logate. În mod tradițional toate datele legate de task-uri ar fi fost salvate într-o singura tabela (sau mai multe legate prin chei străine, lucru ce ar avea același efect) care să aibă ca coloane toate detaliile legate de acel task folosite în aplicație. Deși este buna pentru aplicațiile de dimensiuni mai mici, aceasta modalitate de stocare ar avea ca efect că atât query-urile cât și procesarea datelor vor fi mai încete decât dacă ar fi stocate fiecare în locații diferite în baza de date, în funcție de cum sunt grupate în paginile web sau în funcție strict de coloanele care sunt necesare la fiecare query.

Aplicația consolă interoghează constant baza de date RavenDB pentru a vedea dacă sunt evenimente noi inserate, iar când găsește unul, îl transmite tuturor obiectelor handlere, care își actualizează view-urile de care sunt responsabili.

Un handler tratează orice tip de eveniment în funcție de modul în care acel eveniment afectează acel view; de exemplu handlerul pentru view-ul TopicList își va update conținutul pentru evenimente de tipul TopicUpdated însă nu se va modifica pentru evenimentul TaskUpdated deoarece acel eveniment nu schimbă cu nimic conținutul datelor din viewul TopicList.

Baza de date SQL

Fig 15: Diagrama bazei de date SQL



Pentru a ilustra mai bine conceptul de sisteme distribuite în aplicația mea evenimentele le voi salva în o baza de date iar viewurile le voi materializa în o baza de date separată. Această distribuție poate aduce avantaje foarte interesante deoarece în acest mod baza de date unde se inserează evenimente poate fi una optimizată pentru inserări, de exemplu non relaționala, iar baza de data ce conține tabelele de viewuri poate fi una optimizată pentru citiri, cum ar fi cea de tip relaționala.

Dar această arhitectură de folosire a două baze de date pentru a eficientiza partea de read și partea de write ar putea aduce și dezavantaje deoarece acest lucru ar însemna că aplicația trebuie să ia informații din prima baza de date și sa le folosească în a doua, lucru care ar putea fi destul de costisitor din punct de vedere al timpului. Diagrama bazei de date SQL poate fi văzută în figura 15.

## Proiectul Data Layer

Proiectul DataLayer conține clasele ce interacționează cu bazele de date. Baza de date pentru viewuri a fost generată folosind modul de lucru Code First din Entity Framework și în acest proiect găsim Contextul acestei baze de date, repository-ul și migrările care au fost folosite la updatarea tabelelor. Tot în acest proiect avem și EventStore, care este o clasă ce derivă din interfața IEventStore găsita în framework-ul CQRSlite, care se ocupa cu inserarea sau citirea evenimentelor în baza de date. Tabelele în care sunt salvate informațiile despre evenimente sunt:

Aggregates, unde se salvează Id-ul agregatului, versiunea acestuia (reprezentând numărul de evenimente pentru acel agregat) și tipul acestuia, care în aplicația implementată de mine poate fi Task sau Topic.

Events: în această tabelă se salvează toate detaliile legate de un eveniment precum AggregateId, TimeStamp, IssuedBy, Type, Version și Data unde se salvează toate informațiile specifice acelui eveniment.

Când se adaugă un nou eveniment în baza de date această operație necesită actualizarea atât a tabelului Aggregates cât și Events ceea ce necesită ca această operație de adăugare a unui nou eveniment să fie tranzacțională. Pentru procesul de inserare a noilor evenimente clasa EventStore este punctul de lucru cu baza de date. De asemenea cu clasa EventStore interacționează alte clase din framework-ul CQRSlite, cum ar fi Sesiunea (obiect in totalitate implementat de către framework) sau obiectele ce deriva din clasa abstractă AggregateRoot.

## Proiectul Domain

În proiectul Domain găsim definite obiectele de baza ale aplicației, care sunt folosite în toate celelalte proiecte. Aceste obiecte sunt organizate în 3 subfoldere: Aggregates, Events și Views. În Aggregates sunt obiectele ce derivă din clasa abstractă AggregateRoot din framework-ul CQRSlite. Clasa AggregateRoot este în primul rând folosită ca marker, alte clase din framework lucrând cu ea, iar în al doilea rând conține funcția ApplyEvent care primește ca parametru un eveniment legat de acel agregat și îl inserează în baza de date folosindu-se de EventStore. În implementarea proiectului meu clasele Task și Topic deriva din ea. În folderul Events se găsesc toate evenimentele folosite în aplicație, atât pentru topic cât și pentru task. În Views se găsesc toate modelele care se folosesc în ApplicationContext și după care este creata baza de date. Deși clasele Task și Topic se găsesc și în Aggregates și în Views, cele din Views nu deriva din clasa abstracta AggregateRoot, și sunt folosite pentru funcționalități diferite.

Flow-ul aplicației la partea de write

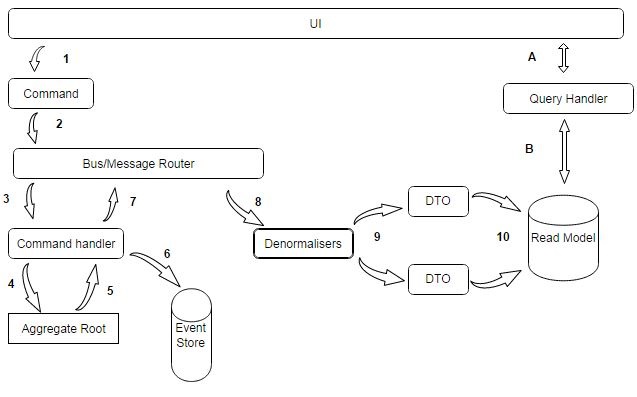


Fig 16: Model arhitectural CQRS + Event sourcing

[Sursă imagine](http://danielwhittaker.me/2014/10/02/cqrs-step-step-guide-flow-typical-application/)

Din UI se trmite o comandă către back-end când un utilizator ia o acțiune să modifice starea aplicației. Această ajunge în controller și de acolo este pasată unui obiect de tipul message router care știe ce fel de command handler să instanțiile care să se ocupe de prelucrarea acelei comenzi. Ajunsă în command handler, asupra comenzii se fac verificări care să determine dacă se poate genera un eveniment din ea. Dacă condițiile se îndeplinesc atunci, în cazul în care comanda este legată de o entitate deja existentă în baza de date, informațiile din comanda sunt trimise către clasa ce implementează AggregateRoot pentru acea entitate și acolo este generat evenimentul sau dacă comanda conține informații despre crearea unei noi entități, evenimentul este generat direct în command handler și introdus în Event Store prin sesiune. Atât flow-ul la citire cât si cel la scriere este descris foarte bine în figura 16 de mai sus.

Flow-ul aplicației la partea de read

Odată inserat în tabela de evenimente, acesta e gata să fie procesat de aplicația EventsConsummer care interoghează constant baza de date cu evenimente și le procesează pe cele noi. Aplicația EventsPrcessor știe care evenimente au fost procesate și care nu, pentru fiecare view, prin tabela "Views" unde memorează numele view-ului, numărul evenimentelor procesate pentru acel view și o data pentru când a fost procesat ultima oară un eveniment pentru acel view. Pentru a verifica dacă trebuie să actualizeze informațiile în view-uri, aplicația numără numărul de evenimente din tabela "Events" și compară cu numărul de evenimente procesate din tabela "Views",a cărui conținut poate fi văzut in fig 17, iar dacă găsește o diferența atunci încarcă în memorie ultimele evenimente neprocesate și apelează handler-ul potrivit pentru a extrage informațiile din eveniment și actualiza tabela respectivă. Când un utilizator vrea să acceseze niște date din view-uri, din front-end se trimite un query către controllere de unde este pasat la un query bus care știe să instanțieze query handler-ul potrivit pentru a se ocupa de cerere. În Query handler este interogat repository-ul iar datele sunt pasate înapoi către controller pentru a putea fi afișate utilizatorului.

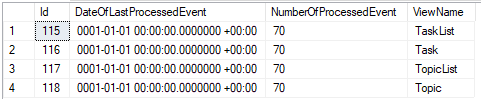


Fig 17: Tabelul views

# Concluzii

În final consider ca investiția pe care am facut-o în această lucrare de licență a dat roade deoarece am descoperit multe lucruri interesante. Lucrul la această lucrare de licență a fost o provocare pentru ca complexitatea acestei arhitecturi a fost la un nivel cu care nu m-am mai întâlnit pana acum, însă acest lucru a fost benefic deoarece m-a făcut să îmi forțez limitele.

Pe măsură ce am progresat cu studiul acestei arhitecturi și implementarea aplicației, mi-am confirmat multe dintre părerile pe care le aveam inițial despre această arhitectură, cum ar fi că este scalabilă sau că oferă posibilitati foarte mari de dezvoltare datorită modului în care datele sunt salvate .

Însă de asemenea am fost surprins de unele aspecte, cum ar fi cât de greu este să pui la punct arhitectura de baza astfel încât evenimentele să fie generate și procesate corect pentru ca utilizatorul sa poată avea vedea in timp real ultima stare a aplicației. În fraza anterioara fac referire la o problemă comună sistemelor distribuite ce folosesc event sourcing, numită problema de consistenșă eventuala, care constă în faptul că există un deplay între momentul în care se generează un eveniment și momentul în care acesta se procesează iar din aceasta cauză utilizatorii nu vad în aplicație ultima versiune a datelor, ci versiunea cu 1-2 secunde în trecut.

Deși a fost mai greu să pun la punct arhitectura de bază care să funcționeze corect, odată ce am facut-o și-a dovedit avantajul scalabilitații, prin faptul ca am reușit să generez statistici pentru 11 grafice mult mai ușor decât aș fi facut-o în o aplicatie ce nu avea această arhitectură. Deoarece statisticile sunt bazate întotdeauna pe acțiunile ce pot fi facute în domeniu, tot ce trebuie să faci pentru a le genera este să aplici formulele dorite pe numarul evenimentelor care prezintă interes.

În concluzie cred că această arhitectură are avantajul că își stochează datele într-un format mult mai consistent cu realitatea, deoarece și în viața de zi cu zi nu ne folosim doar de informațiile din prezent, ci avem în vedere și trecutul, ceea ce înseamnă că o schimbare în domeniul aplicației va veni ca un șoc mai mic pentru această arhitectură și va putea fi încorporată fară schimbări majore.

Datorită acestor motive cred că aplicația dezvolatată pentru această lucrare are potențial, înspecial arhitectura de back-end pentru generare și procesarea evenimentelor. Pe langă continuarea ideii de aplicație pentru productivitate studenților, aceasta poate fi integrată cu alte aplicații cu scop didactic, pentru a crea un learning management sistem căt mai dezvoltat.

# 

# Bibliografie

1. O descriere a bazelor de date NoSQL <https://www.mongodb.com/scale/what-is-a-non-relational-database>
2. Carte despre cum funcționează RavenDb <https://github.com/ravendb/book/releases>
3. Tutorial funcționalități .Net Core: <https://www.tutorialspoint.com/dotnet_core/index.html>
4. Tutorial sintaxă razor <https://www.w3schools.com/asp/razor_syntax.asp>
5. Introducere în event sourcing <https://msdn.microsoft.com/en-us/library/jj591559.aspx>
6. Viziune generală a event sourcing-ului prezentată de Martin Fowler:

<https://martinfowler.com/eaaDev/EventSourcing.html>

1. Exemplu implementare CQRS folosind NoSQL în .Net <https://exceptionnotfound.net/real-world-cqrs-es-with-asp-net-and-redis-part-1-overview/>
2. Un alt exemplu de implementare CQRS împreună cu event sourcing și introducerea în framework-ul CQRSlite <https://www.codeproject.com/articles/991648/cqrs-a-cross-examination-of-how-it-works>
3. Lucrare de cercetare asupra sistemelor distribuite [https://link.springer.com/article/10.1007/s00607-016-0508-7](https://link.springer.com/article/10.1007/s00607-016-0508-7%20)
4. Prezentare generală a CQRS-ului din documentația microsoft cu exemple de cod <https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs>
5. Framework-ul CQRSlite folosit și în implementarea aplicației <https://github.com/gautema/CQRSlite>
6. Sursă foarte bună de FAQs legat de CQRS și Event Sourcing <http://cqrs.nu/>
7. Documentație folosire RavenDB in .Net <https://ravendb.net/docs/article-page/4.0/csharp>
8. Prezentare generală event sourcing cu avantaje și exemple de aplicații potrivite pentru arhitectura <https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing>
9. Libraria folosită la randarea graficelor din aplicație <https://www.chartjs.org/>
10. SQL vs NoSQL <https://www.zdnet.com/article/rdbms-vs-nosql-how-do-you-pick/>
11. Fowler conferință event sourcing <https://www.youtube.com/watch?v=aweV9FLTZkU>
12. Conferință CQRS și event sourcing <https://www.youtube.com/watch?v=JHGkaShoyNs>

# Anexa 1

Baze de date non-relațională [[1](https://www.mongodb.com/scale/what-is-a-non-relational-database)]

O bază de date non-relațională este orice bază de date care nu respectă modelul tradițional al sistemelor de management al bazelor de date. Această categorie de baze de date mai este numită NoSQL, a început să fie din ce în ce mai adoptată odată cu apariția aplicațiilor Big Data. Deși tehnologiile NoSQL variază foarte mult, aceste baze de date sunt de obicei mai flexibile și scalabile decât cele relaționale. Câteva caracteristici ale bazelor de date non-relaționale:

* Modelele de date: spre deosebire de modelele relaționale care necesită o schemă predefinită bazele de date NoSql oferă design flexibil al schemei, lucru care le face mult mai ușor să fie actualizate atunci când se schimbă specificațiile sistemului
* Structura datelor: bazele de date non-relaționale sunt proiectate astfel încât să poată stoca date nestructurate care nu se potrivesc modelului de rând și coloană. Acest lucru este un beneficiu deoarece o mare parte din date din ziua de azi sunt nestructurate
* Scalabilitatea: bazele de date non-relaționale pot fi scalate ușor folosind servere care nu necesită foarte multă performantă
* Gratuite: Bazele de date NoSQL sunt în mare parte gratuite, ceea ce înseamnă că nu trebuie să plătești taxe pentru licența

RavenDB [[2](https://github.com/ravendb/book/releases)]

RavenDB este o bază de date NoSQL bazată pe documente, distribuita și cu performantă ridicată. O bază de date bazate pe documente stochează “documente”, însemnând informație structurată în format self-contained (date de sine stătătoare/independente). Un document este de obicei în format JSON sau XML. RavenDB este, esențial, o bază de date pentru stocarea și lucrarea cu date în format JSON. In această bază de date se pot stoca și date binare și alte câteva tipuri de date, însă este folosită în principal pentru documente JSON.

RavenDb poate rula pe un singur nod (potrivită pentru dezvoltare sau pentru aplicații de dimensiuni reduse) sau pe un cluster de noduri (lucru care oferă disponibilitate ridicată, load balancing, geo distribuire a datelor sau a procesării lor).

Spre deosebire de alte baze de date NoSQL, RavenDB respectă principiile ACID. ACID vine de la atomic, consistent, izolat și durabil ceea ce înseamnă că are tranzacții în adevăratul sens al cuvântului.

Când o instanța de RavenDB primește un query acesta este trimis la query optimizer pentru analiză. Rolul query optimizer-ului este de a determina ce indecși ar trebui să fie folosiți la acel query. În RavenDB sunt două tipuri de query-uri: query-urile dinamice de tipul “from Orders where …”, care oferă query optimizer-ului libertatea completă să aleagă ce indecși să folosească și query-urile care specifica ce tip de index să fie folosit, cum ar fi “from index Orders/ByCompany where …” care spun explicit bazei de date sa folosească indexul Orders/ByCompany. Dacă componenta de query optimizer nu găsește un index pentru execuția unui query atunci aceasta construiește unul în timp real.

Indexarea în RavenDB este o operație ce se execută în background ceea ce înseamnă că noul query va aștepta terminarea acelei indexări sau va da timeout. În același timp RavenDB poate lansa operația de indexare în spate și să folosească indecșii deja existenți, dar posibil neactualizați, care ar putea să nu ofere rezultate care să conțină ultimele modificări, iar când noul index a fost creat, RavenDB șterge fostul index care acum nu v-a mai fi utilizat. Cu timpul componenta query optimizer va analiza cele mai utilizate tipuri de interogări executate și va genera un set optim de indecși care să răspundă la acele query-uri. Schimbări asupra tipurilor de query-uri vor declanșa și schimbări asupra indecșilor existenți, pentru că se va ajusta noilor cerințe.

Query-urile pe RavenDB folosesc întotdeauna un index și de aceea sunt foarte rapide, nefiind necesare scanări ale întregii baze de date.

# Anexa 2

.Net Core [[3](https://www.tutorialspoint.com/dotnet_core/index.htm)]

.Net core este cea mai nouă platformă de dezvoltare deținută de Microsoft. Funcționează pe mai multe platforme și a fost reproiectată astfel încât să facă .Net rapid, flexibil și modern. .Net Core este o contribuție majoră adusă de Microsoft în domeniul dezvoltării aplicațiilor. Prin acest framework open-source, dezvoltatorii pot construi aplicații Android, iOS, Linux, Mac și Windows.

Asp.Net Core este un nou framework web de la Microsoft. A fost reproiectat de la bază să fie rapid, flexibil, modern și funcționeze pe platforme multiple și poate fi folosit pentru dezvoltarea aplicațiilor web cu .Net. Asp.Net a fost folosit de mulți ani pentru dezvoltarea aplicațiilor web. De atunci, framework-ul a trecut printr-o evoluție constantă și în prezent a ajuns la versiunea cea mai recentă Asp.Net Core 2.0.

* Asp.Net Core nu este o continuare a framework-ului Asp.Net 4.6
* Este un framework complet nou, de sine stătător, ce nu depinde de Asp.Net 4.6
* Este o rescriere a framework-ului curent Asp.Net 4.6, dar de dimensiuni mai reduse și mai modular

Asp.Net este un framework web open-source, la fel ca .Net Core, optimizat pentru cloud pentru dezvoltarea aplicațiilor web moderne pentru platformele Windows, Linux sau Mac. Include framework-ul MVC, care acum combină funcționalitățile MVC-ului și a Web API-ului într-un singur framework de programare web.

* Aplicațiile Asp.Net Core pot rula pe .Net Core sau pe framework-ul .Net complet
* Arhitectura a fost proiectată astfel încât să ofere un framework de dezvoltare optimizat pentru aplicații care sunt lansate în cloud sau rulate local
* Este format din componente modulare, fară prea multe elemente nenecesare, pentru a se păstra flexibilitatea în timp ce soluția este dezvoltată

Asp.Net Core vine cu următoarele avantaje:

* Are schimbări arhitecturale ce au rezultat în un framework mai mic și mai modular
* Nu se mai bazează pe System.Web.dll ci pe un set granular de pachete NuGet
* Permite optimizarea aplicației pentru a include doar pachetele NuGet de care este nevoie
* Din cauza faptului că este un framework mai mic decât Asp.Net, oferă o securitate mai buna, performantă îmbunătățita și costuri scăzute

Razor [[4](https://www.w3schools.com/asp/razor_syntax.asp)]

Razor este o sintaxă de programare simplă pentru integrarea codului C# în paginile web.

Sintaxa razor este bazată pe framework-ul Asp.Net, care face parte din framework-ul Microsoft.Net, proiectat specific pentru crearea aplicațiilor web.

Paginile web razor pot fi descrise ca pagini HTML cu două tipuri de conținut: conținut HTML si cod Razor.

Când serverul citește pagina, mai întâi execută codul Razor, înainte ca să trimită pagina HTML la browser. Codul executat pe server conține acțiuni ce nu pot fi executate în browser, de exemplu accesarea bazei de date. Codul de pe server generează conținut HTML dinamic în timp real, înainte să o trimită la browser. Văzut din punct de vedere al browser-ului, conținutul HTML-ului generat de codul de pe server nu este diferit de conținutul HTML static.

Paginile web Asp.Net cu sintaxă Razor au un tip special de extensie la fișier: “.cshtml” (Razor care folosește C#) sau “vbhtml” (Razor care folosește VB).

# Anexa 3

RDBMS vs NoSQL [[16](https://www.zdnet.com/article/rdbms-vs-nosql-how-do-you-pick/)]

Primul lucru care trebuie luat in considerare când alegem o bază de date este caracteristica datelor pe care le vei manipula. Dacă datele au o structură simplă tabulară, ca un spreadsheet de contabilitate, atunci un model relațional ar putea fi adecvat.

Date precum cele geo-spațiale, date inginerești sau modele moleculare tind să fie foarte complexe. Pot avea niveluri multiple de nesting și modelul final al datelor pot fi complicate. Asemenea tipuri de date, în trecut, au fost modelate în tabele relaționale, însă ele nu se potrivesc în mod natural în aceste structuri bidimensionale de rânduri și coloane.

În cazuri ca acestea, ar trebui luate în considerare bazele de date non-relaționale. Datele cu nivele multiple de nesting și ierarhii sunt foarte ușor de reprezentate în formatul JavaScript Object Notation (JSON) folosit de unele baze de date NoSql.

Următorul lucru care trebuie luat în considerare este volatilitatea datelor. Este posibil ca modelul datelor să se schimbe și să evolueze sau este mai probabil că va rămâne același? În general, nu se pot cunoaște toate detaliile despre cum va arata modelul datelor la momentul proiectării bazei de data așa ca putină flexibilitate ar fi utilă. Problema volatilității datelor aduce foarte multe probleme utilizatorilor sistemelor de management al bazelor de date relaționale (RDBMS).

O plângere frecvent legată de bazele de date non-relaționale este că cedează consistența în favoarea disponibilității ridicate. Totuși acest lucru nu poate fi spus despre toate bazele de date NoSQL. În general, o baza de date relaționala ar trebui considerată în aplicațiile care necesită tranzacții de mai multe rânduri și join-uri complexe. În o baza de date NoSQL ca MongoDb, de exemplu, un document (obiect complex) poate fi echivalentul unor join-uri pe mai multe tabele și consistenta este garantată pentru acel obiect.

Bazele de date NoSQL, în general, evită funcțiile bazelor de date relaționale, cum ar fi join-urile între tabele care pot încetini citirea datelor.

Bazele de date relaționale, pe de altă parte, sunt potrivite pentru aplicațiile ce necesită query-uri complexe și analiză. Uneori datele Hadoop sunt încărcate în o bază de date relaționala pentru a putea fi analizate mai bine și forma rapoarte cât mai complexe. Deci o bază de date non- relaționala este potrivită când nevoile pentru query-uri complexe sunt foarte critice.

Analizele în timp real pentru date operaționale sunt mai bine potrivite pentru un mediu non relațional. Mai mult, în cazuri în care datele sunt aduse din mai multe sisteme, pentru a construi o aplicație, o bază de date NoSQL poate aduce și mai multe beneficii.

Bazele de date relaționale și NoSQL pot coexista în unele scenarii pentru a lua cele mai bune avantaje de la fiecare. IBM a anunțat implementarea unui API pentru MongoDB care oferă ficționalități de limbaj query și wire protocol.

# Anexa 4

Tipuri de sisteme distribuite [[5](https://link.springer.com/article/10.1007/s00607-016-0508-7)]

1. Sisteme distribuite pentru performantă înaltă

O clasă importantă de sisteme distribuite este cea folosită pentru procesările ce necesită putere computaționala foarte mare. Procesările de performantă înaltă a apărut odată cu introducerea calculatoarelor cu multiprocesoare. În aceste sisteme, mai multe procesoare sunt organizate astfel încât toate au acces la aceeași memorie fizică, după cum este arătat în figura X, unde se face o comparație între arhitectura bazată pe multiprocesoare și cea bazată pe mai multe calculatoare. Această arhitectură bazată pe mai multe procesoare care împart aceeași memorie s-a dovedit a fi foarte eficientă pentru îmbunătățirea performantelor programelor și era relativ simplă de pus in practică.

Esența multiprocesoarelor care împart aceeași memorie este că mai multe thread-uri execută instrucțiuni în același timp, toate având acces la memoria comună. Din păcate, acest model nu se poate scala ușor: pană acum s-au dezvoltat calculatoare care au doar un număr relativ mic de procesoare care să aibă acces în mod eficient la memoria comună. Pentru a depăși limitările sistemelor cu memorie comună, procesarea de înaltă performantă s-a mutat către sisteme distribuite. Această schimbare a însemnat, de asemenea, că multe programe au început să își transmită mesaje, în loc să modifice memoria comună pentru a comunica și sincroniza execuția thread-urilor. Inițial modelele de comunicare prin mesaje s-au dovedit a fi mult mai dificile iar erorile apăreau mai des comparativ cu modelele de programare bazate pe memoria comună.

Din acest motiv s-au făcut cercetări semnificative în încercarea de a construi sisteme formate din mai multe calculatoare distincte dar care imitau modelul de memorie comună, numite sisteme DSM.

În esența sistemele DSM permiteau unui procesor sa acceseze o locație de memorie al unui alt calculator ca si cum ar fi memorie locală. Acest obiectiv a putut fi atins folosind tehnicile deja existente in sistemele de operare, de exemplu, mapând toate paginile de memorie principală ale diferitelor procesoare într-o singură adresă virtuală. Când procesorul A vrea să acceseze o pagină localizată la un alt procesor B, este ridicată o excepție de tipul Page Fault în A care îi permite sistemului de operare de la A să aducă conținutul din pagina cerută de la B în același mod cum l-ar aduce disk-ul local. În același timp, procesorul B va fi informat că acea pagină nu este momentan disponibilă.

Această idee elegantă de a mimica sistemele cu memorie comună folosind mai multe calculatoare a fost eventual abandonată din cauză că performanta pe care o oferea nu a fost suficient de mare cu așteptările programatorilor, care în final au început să se bazeze mai mult pe modelele de comunicare prin transmiterea mesajelor, care deși erau mai complexe erau mai predictibile.

* 1. Cluster computing

Sistemele Cluster-computing au devenit populare când prețul calculatoarelor personale și a stațiilor de lucru au scăzut. Cluster computing este folosit pentru programarea în paralel unde un program, care are nevoie de putere de procesare foarte mare, rulează în paralel pe mai multe mașini. Un cluster este format din o colecție de noduri care sunt controlate de către un singur nod master. Nodul master se ocupă cu alocare de noduri anumitor programe, menține o coadă de sarcini care trebuie executate și oferă o interfața pentru utilizatorii sistemului. Cu timpul clusterele moderne au început să aibă și unele noduri dedicate anumitor sarcini precum cea de procesare a datelor, de management al nodurilor sau noduri speciale care sunt optimizate pentru stocarea de date pentru ca fiecare tip de sarcină să poată fi executat în cel mai eficient mod posibil.

* 1. Grid computing

O funcționalitate caracteristică a sistemelor tradiționale de cluster computing este omogenitatea. În cele mai multe cazuri calculatoarele din un cluster sunt aproximativ la fel, cu același sistem de operare și sunt conectate prin aceeași rețea. Totuși, pe măsura ce sistemele distribuite au evoluat acestea au început să aibă arhitecturi mai hibride în care nodurile sunt special configurate pentru anumite task-uri. Această diversitate este și mai răspândita în sistemele de Grid computing, unde nu se pleacă de la ideea ca componentele sunt similare din punct de vedere al hardware-ului, sistemelor de operare sau a politicilor de securitate.

Caracteristica principala în sistemele de grid computing este că resurse din diferite organizații sunt aduse împreuna pentru a permite colaborarea unor oameni din diferite instituții. Aceasta colaborare este realizată sub forma unei organizații digitale. Procesele care aparțin aceleiași organizații virtuale au drepturi de acces la resursele pe care acea organizație le oferă. Aceste resurse pot fi servere de procesare (computere normale sau super computere care pot fi împlementate la rândul lor ca sisteme cluster), computere specializate în stocarea fișierelor și baze de date.

Cea mai mare parte din software-ul pentru realizarea sistemelor de grid computing este dedicată oferirii de acces la resursele din diferite sisteme administrative, și doar acelor utilizatori care aparțin unei anumite organizații virtuale. Din acest motiv în dezvoltarea acestor sisteme concentrarea se face asupra problemelor arhitecturale.

* 1. Cloud computing

Cloud computing reprezintă o colecție de resurse virtualizate ușoar de folosit și accesibile. Clientul își poate configura alocarea de resurse în mod dinamic, lucru care oferă o scalabilitate foarte mare: dacă aplicația are nevoie de mai multă putere de calcul aceasta se poate aloca în mod automat, iar dacă necesitățile computaționale ale aplicației scad, resursele se pot dealoca, iar în acest mod clientul plătește doar pentru ce consuma.

În practică cloud computing poate fi împărțit în patru categorii ca în figura de mai jos:

Hardware:

Nivelul cel mai jos oferă posibilitatea de a aloca resursele hardware de care clientul are nevoie: procesoare, routere, până la putere și sisteme de răcire. Acest nivel este localizat în general la centrele de date și conține resursele care sunt invizibile clienților.

II) Infrastructura

Acesta este un strat important care formează baza pentru majoritatea platformelor de cloud computing și se folosește de tehnici de virtualizare pentru a oferi clienților o infrastructura ce consta din spațiu de stocare virtual și resurse de calcul. În cloud computing totul se bazează pe alocare de dispozitive de stocare sau servere virtuale.

Platforma:

Platforma oferă unui client de cloud computing ceea ce un sistem de operare oferă unui dezvoltator de aplicații: mijloace de a dezvolta și lansa aplicațiile ce au nevoie sa ruleze în cloud. Providerii de PaaS oferă un mediu optimizat în care utilizatorii își pot instala aplicații, seturi de date lucra la dezvoltarea sau testarea funcționalităților. Cele mai multe servicii de tipul PaaS sunt destinate dezvoltării de software. Aceste platforme oferă putere computaționala, spațiu de stocare, ide pentru scrierea codului, sistem de versionare, compilatoare și servicii de testare.

Nivelul aplicație:

Acest nivel oferă clienților aplicații gata implementate și customizabile. Câteva exemple de tipuri de aplicații mai cunoscute din aceasta categorie sunt procesoare de text, aplicații spreadsheet, de prezentare sau calendare.

Toate aceste servicii oferite de cloud se bazează pe tehnici de virtualizare, iar ceea ce vede utilizatorul reprezintă ceea ce este cu adevărat în spate. În cele mai multe cazuri resursele oferite utilizatorului pot fi obținute prin combinarea puterii computaționale a mai multor servere.

# Anexa 5

Event sourcing [[6](https://martinfowler.com/eaaDev/EventSourcing.html)]

Putem interoga starea unei aplicații pentru a afla stagiul în care se afla și acest lucru poate răspunde la multe întrebări. Însă de multe ori vrem să știm nu doar starea curentă ci și seria de schimbări care ne-a adus în acel punct.

Event Sourcing asigură că toate schimbările din aplicație sunt stocate ca o secvența de evenimente. Nu doar că putem interoga aceste evenimente, dar putem de asemenea să folosim șirul de evenimente pentru a reconstrui stări din trecut.

Ideea fundamentală din Event Sourcing este aceea de a asigura că fiecare schimbare la starea unei aplicații este capturată într-un obiect de tip eveniment, și că aceste evenimente sunt stocate în ordinea în care au fost aplicate începând de la începutul aplicației.

Cel mai evident avantaj al event sourcing-ului este că avem acces la un audit log ce conține toate schimbările, însă acest lucru ar putea fi obținut prin modalități mai ușoare (am putea de exemplu să memoram într-un fișier starea obiectului de fiecare data când aceasta se schimbă sau să folosim funcționalitatea deja implementată de unele baze de date care pot oferi istoricul tuturor modificărilor doar prin niște setări de configurare).

Cheia in Event Sourcing este că ne garantează că toate schimbările în obiectele de domeniu sunt inițializate de evenimente, lucru care ne duce la un număr de avantaje care pot fi extrase din logul de evenimente:

Reconstruire completă: putem șterge complet starea aplicației și să o reconstruim aplicând evenimentele din event log pe o aplicație goală

Interogări temporale: putem determina starea aplicației la orice punct din timp

Adăugarea evenimentelor noi: daca un eveniment din trecut a fost incorect putem să inserăm un alt eveniment care să corecteze greșelile din primul. De asemenea am putea simula stări alternative ale aplicației inserând anumite evenimente în trecut, experimentând în acest mod ce s-ar fi întâmplat daca anumite acțiuni ar fi fost luate.

Un exemplu comun de aplicații care folosesc event sourcing sunt sistemele de versionare a codului sursă.

Ultima stare a aplicației poate fi stocată pe disk sau în memorie deoarece datorită evoluării resurselor calculatoarelor acest lucru nu mai e o problemă.

# Anexa 6

CQRS [[12](https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs)]

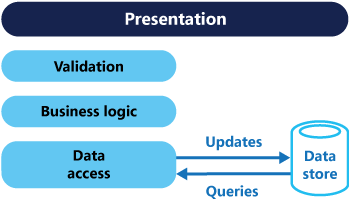
Cqrs separă operațiile de citire a datelor, de cele pentru editarea datelor folosind interfețe separate. Acest lucru poate sa maximizeze performanta, călibilitatea și securitatea. Acesta ajută evoluția sistemului în timp oferind flexibilitate ridicată.

Fig 18: Modelul tradițional CRUD

[Sursă imagine](https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs)

În sistemele tradiționale de management (Fig 18), atât comenzile (modificările aduse datelor) cât și interogările (cererile pentru date) sunt făcute la același set de entități din repository-ul de date. Aceste entități pot reprezenta subseturi de rânduri din unul sau mai multe tabele în baze de date relaționale. În mod tipic În aceste sisteme toate operațiile de citire, creare, actualizare și ștergere sunt aplicate asupra aceleași entități. De exemplu, un obiect folosit pentru transferul datelor (DTO) este luat din baza de date și afișat pe ecran. Un utilizator modifică câteva field-uri ale acestui obiect (DTO) și apoi este salvat din nou în baza de date. În acest caz același obiect este folosit atât pentru operațiile de citire cât și pentru cele de editare. Figura din dreapta ilustrează o arhitectura CRUD tradiționala.

Aceste sistem funcționează bine când aplicația nu are foarte multă logica de business. Tool-urile de scaffolding pot genera foarte ușor cod pentru a accesa datele în acest mod, care poate fi apoi customizat după necesitați.

Totuși, abordarea CRUD tradiționala are unele dezavantaje:

* Deseori există o diferența între datele de care are nevoie o operație de citire fată de una de editare și de aceea în aceste sisteme unele coloane sau proprietăți sunt editate, deși ele nu sunt necesare în acea operație.
* Poate avea un impact negativ asupra performantei datorită suprasolicitării stratului de acces al bazei de date și complexității interogărilor necesare pentru a extrage informațiile.
* Pot face sistemele de management al securității și permisiunilor mai complexe deoarece fiecare entitate este folosită atât pentru operațiile de citire cât și pentru cele de scriere, lucru care poate expune date în contextul greșit.

Command and Query Responsibility Segregation (CQRS) este un patern care separă operațiile care citesc datele (interogările) de operațiile care editează datele (numite comenzi), folosind interfețe diferite. Asta înseamnă că modelele de date folosite pentru interogări și cele care modifică datele sunt diferite. Modelele pot fi apoi izolate, după cum e arătat în următoare figură de mai sus, deși nu este absolut necesar.

Comparativ modelul singular de entități folosit în sistemele CRUD, folosirea modelelor separate pentru interogările și modificările datelor în sistemele bazate pe CQRS simplifică design-ul și implementarea. Totuși, un dezavantaj este ca spre deosebire de arhitecturile CRUD, codul pentru arhitectura CQRS nu este generat automat de mecanisme de scaffold-ing.

Modelul pentru interogări folosit pentru citirea datelor și modelul de update, pentru scrierea datelor, pot accesa același data store, putând folosi și view-uri sql sau generând proiecții în timp real. Totuși, în general este recomandat să separi datele în data store-uri diferite pentru a maximiza performanta, scalabilitatea și securitatea, după cum este arătat în figura 19:

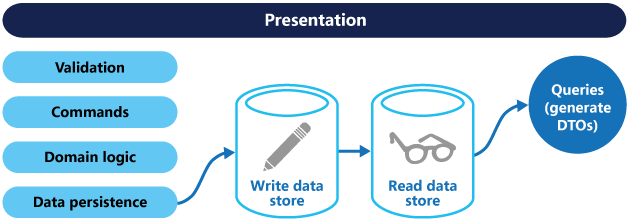


Fig. 19: Folosirea bazelor de date diferite pentru citire si scriere

[Sursă imagine](https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs)

Read store-ul poate fi doar o replică read-only a write store-ului, sau pot avea o structură total diferită. Folosirea mai multor replici a read store-ului poate îmbunătăți foarte mult performanta interogărilor și responsivitatea UI a aplicației, în special în scenarii distribuite unde replici read-only sunt puse cât mai aproape de instanțele aplicațiilor.

Separarea store-urilor pentru citire și scriere de asemenea permite fiecăruia să fie scalat suficient de mult comparativ cu nivelul cu care este folosit. De exemplu în general operațiile de citire sunt mult mai dese decât cele de modificare și CQRS ar putea permite optimizarea lor mai mult decât a celor de scriere care nu au un load foarte ridicat.

Pentru a decide daca în implementarea aplicației ar fi mai potrivit sa folosim CQRS decât CRUD putem lua în considerare următoarele aspecte:

* Împărțirea datelor în store-uri diferite pentru citire și scriere poate îmbunătăți performanta și securitatea unui sistem, dar poate adauga complexitate din punct de vedere al rezilienței și consistentei eventuale. Store-ul pentru modelul de read trebuie să fie actualizat să reflecte schimbările la modelul de write și poate fi dificil de detectat când un utilizator a făcut o interogare pentru niște date care se bazează pe un model de read store neactualizat, ceea ce ar însemna că datele care vor fi returnate nu vor include și cele mai recente modificări.
* O abordare bună de luat în considerare ar fi sa aplicam CQRS la secțiuni limitate din sistem, unde va aduce și cea mai multă valoare.
* O abordare comună folosită pentru a rezolva problema eventual consistency este folosirea paternului event sourcing împreuna cu CQRS astfel încât modelul de scriere este bazat doar pe operații de adăugare (append-only) evenimente create de executarea comenzilor. Aceste evenimente sunt folosite pentru a actualiza view-urile materializate care au rolul de model de citire.

Paternul CQRS este util de folosit în următoarele situații:

* Domenii colaborative unde operații multiple sunt executate în paralel asupra acelorași date. CQRS permite să definim comenzi care să nu aibă conflicte cu celelalte comenzi, chiar dacă operam asupra aceluiași tip de date.
* Aplicații ce conțin interfețe de utilizator în care utilizatorii sunt ghidați prin un proces cu o serie de pași sau cu modele de domeniu complexe. Modelul de scriere are asociate servicii pentru procesarea fiecărei comenzi, validarea inputului și validare a business-ului pentru a se asigura că totul este consistent pentru fiecare agregat în modelul de scriere. Scenarii unde performanta operațiilor de citire a datelor trebuie să fie optimizate separat de performantele operațiilor de scriere a datelor, în special unde rația de operații de citire/scriere este foarte mare, și unde scalarea orizontala este necesara. De exemplu, în multe sisteme numărul de operații de citire este mai mare decât numărul de operații de scriere. O abordare potrivită în acest caz ar fi scalarea modelului de read pe mai multe instanțe, dar folosirea modelului de scriere doar pe câteva instanțe. Un număr mic de instanțe pe modelul de scriere, ajută de asemenea cu minimizarea aparițiilor conflictelor de merge.
* Scenarii în care o echipa de dezvoltatori se pot concentra asupra modelului de domeniu de scriere în timp ce alta se poate ocupa cu modelul de citire și interfețele de utilizatori.
* Scenarii în care sistemul este așteptat să evolueze în timp și ar putea să conțină mai multe versiuni ale modelului de read sau în care regulile de business se schimbă cu regularitate.
* Integrarea cu alte sisteme, în special în combinație cu event sourcing, unde o defecțiune temporara a unui subsitem nu ar trebui să afecteze disponibilitatea celorlalte sisteme.

Acest pattern nu este recomandat în următoarele situații:

* Situații în care domeniul și regulile de business sunt simple.
* Situații în care aplicațiile au stilul interfeței de utilizator bazat pe modelul CRUD și operațiile asupra datelor corespund acestui pattern sunt suficiente.
* Pentru implementări ce afectează întregul sistem. Sunt componente specifice ale unui scenariu de management al datelor unde CQRS poate fi folositor, dar poate adaugă o complexitate destul de mare care să nu fie neapărat necesară.

Paternul CQRS este folosit deseori împreuna cu Event Sourcing. Sistemele bazate pe CQRS folosesc modele separate pentru citire și scriere, fiecare customizate funcționalităților relevante din sistem. Când CQRS este folosit împreuna cu Event Sourcing, componenta de event store este modelul de scriere și reprezintă sursa de adevăr a aplicației. Modelul de citire a unui sistem bazat pe CQRS oferă view-urile materializate a datelor, în mod tipic sub forma de view-uri denormalizate. Aceste view-uri sunt customizate pentru interfețele și cerințele de display ale aplicației, lucru care ajută maximizarea atât a performanței interfeței de utilizator a aplicației cât și a vitezei de interogare a datelor.

Folosind fluxul de evenimente ca store-ul pentru scriere, spre deosebire de starea aplicației la un anumit moment din timp, se evita conflictele la modificări asupra aceluiași agregat și se maximizează performata și scalabilitatea. Aceste evenimente pot fi folosite pentru a genera în mod asincron view-uri materializate a datelor care sunt folosite pentru a popula store-urile de citire.

Un avantaj al faptului că event store-ul este sursa de adevăr a aplicației, este ca putem șterge view-urile materializate și să facem replay la toate evenimentele din trecut pentru a crea noua reprezentare a stării curente când sistemul evoluează sau când modelul de citire trebuie să se schimbe. View-urile materializate sunt efectiv un cache read-only al datelor.

Pentru folosirea paternului de CQRS cu Event Sourcing trebuie să luam în considerare următoarele aspecte:

* Ca în orice sistem unde store-urile pentru citire și scriere sunt separate, sistemele bazate pe acest pattern sunt eventual consistente. În sistemele care oferă consistență eventuală există un delay între generarea și stocarea evenimentului în baza de date și actualizarea modelului de citire.
* Acest pattern adaugă complexitatea deoarece codul trebuie creat ca să genereze și să gestioneze evenimentele, să materializeze și să actualizeze view-urile sau obiectele de care au nevoie interogările sau modelul de citire. Complexitatea paternului CQRS când este folosit împreuna cu Event Sourcing poate face o implementare de succes să fie mai dificilă și necesită o abordare diferita la proiectarea sistemelor. Totuși, event sourcing poate ușura modelarea domeniului și reconstruirea sau creare de noi view-uri deoarece intenția schimbării este salvată în eveniment.
* Generarea view-urilor materializate care urmează să fie folosite în modelul de citire sau în proiecțiile datelor făcând replay la evenimente și reprocesându-le poate necesita timp de procesare și resurse semnificative. Acest lucru este adevărat în special dacă este necesară combinarea sau analiza valorilor pe perioade lungi de timp, deoarece toate evenimentele din trecut trebuie să fie reexaminate și reprocesate dacă este cazul. Această problema poate însă fi rezolvată prin implementarea snapshot-urilor a stării aplicații la anumite intervale programate, cum ar fi odată la un anumit număr de evenimente generate, sau în funcție de starea curentă a entității.