UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Venture

propusă de

Radu Șolcă

Sesiunea: iulie, 2017

Coordonator științific

Drd. Florin Olariu

Conf. Dr. Adrian Iftene

UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Venture

Radu Șolcă

Sesiunea: iulie, 2017

Coordonator științific

Drd. Florin Olariu

Conf. Dr. Adrian Iftene

DECLARAȚIE PRIVIND ORIGINALITATE ȘI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licență cu titlul „Venture” este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară 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 și în traducere proprie din altă limbă, sunt scrise între ghilimele și dețin referința precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alți autori deține referința precisă;
- codul sursă, imaginile etc. preluate din proiecte open-source sau alte surse sunt utilizate cu respectarea drepturilor de autor și dețin referințe precise;
- rezumarea ideilor altor autori precizează referința precisă la textul original.

Iași, 27.06.2017

Absolvent Radu Șolcă

_____

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „Venture”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

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

Iași, 27.06.2017

Absolvent Radu Șolcă

_____

ACORD PRIVIND PROPRIETATEA DREPTULUI DE AUTOR

Facultatea de Informatică este de acord ca drepturile de autor asupra programelor-calculator, în format executabil și sursă, să aparțină autorului prezentei lucrări, Radu Șolcă.

Încheierea acestui acord este necesară din următoarele motive:

Doresc drepturile de autor asupra lucrării de față, pentru a continua dezvoltarea și aprofundarea acesteia, în vederea utilizării sale drept baza pentru o viitoare lucrare de disertație.

Iași, 27.06.2017

Decan Conf. Dr. Adrian Iftene

_____

Absolvent Radu Șolcă

_____

Cuprins

Introducere

Lucrarea de față își propune să prezinte principalele aspecte teoretice și practice ale arhitecturii bazate pe microservicii.

O arhitectura bazata pe microservicii este o forma „lightweight” a SOA (Service Oriented Architecture), unde serviciile sunt focusate pe a face o singura treaba și pe a o face bine. Acest stil arhitectural creste rapid în popularitate pentru dezvoltarea și mentenanța sistemelor complexe server-side. (Gammelgaard, 2017)

Arhitectura clasică, monolitica vine cu avantajele și dezavantajele ei. Spre exemplu, dezvoltarea inițiala este mai rapida, adăugarea de componente noi este relativ ușoara – mai ales când aplicația este încă de mici dimensiuni iar componentele se integrează ușor între ele. Pe de altă parte, pe măsură ce aplicația crește în dimensiune, devine din ce în ce mai greu de modificat, costurile de mentenanță cresc exponențial și devine mai greu de înțeles pentru un programator nou intrat pe proiect. (Gooen)

Prin comparație, o arhitectura bazata pe microservicii vine cu provocările ei, dar rezolva multe din probleme mai sus menționate. Fiind vorba de un sistem distribuit, costurile inițiale de dezvoltare sunt mai ridicate decât în cazul arhitecturii monolitice și testarea poate fi mai greoaie. Dar, aplicația rămâne ușor scalabila și ușor de întreținut pe tot parcursul dezvoltării acesteia iar unui programator nou intrat pe proiect îi va veni mai ușor să înțeleagă oricare serviciu individual din sistemul distribuit. În plus, un sistem distribuit este mai rezilient decât unul monolitic – o eroare într-un serviciu nu se propaga neapărat în restul sistemului. De asemeni, elimină angajamentul pe termen lung fața de o anumita tehnologie – spre exemplu: servicii diferite pot fi scrise în limbaje de programare diferite – și permite un timp scurt de la începerea dezvoltării pana la lansarea în producție. (Badola, 2015)

Primul capitol al acestei lucrări, intitulat „Contribuții”, va prezenta elementele inedite pe care le aduce această lucrare, în contextul stilului arhitectural bazat pe microservicii.

Al doilea capitol, intitulat „Aspecte teoretice”, va explica fundamentele teoretice ale stilului arhitectural bazat pe microservicii, precum și alte concepte strâns legate de acesta.

Al treilea capitol, intitulat „Aspecte practice”, va exemplifica noțiunile prezentate în capitolul precedent prin aplicația „Venture”. Aceasta vine în ajutorul oricărei persoane ce dorește să realizeze un proiect (informatic, de voluntariat etc.), dar fie nu poseda o imagine clara de ansamblu, fie nu are cunoștințele necesare pentru finalizarea lui.

Al patrulea și ultimul capitol al lucrării, intitulat „Posibile îmbunătățiri”, va veni cu idei referitoare la viitoarea dezvoltare a proiectului.

1.Contribuții

În prima parte a lucrării de față (capitolul 2., „Aspecte Teoretice”) au fost adunate și conspectate informații despre concepte și pattern-uri relevante subiectului, din diverse surse, prezentate în bibliografia de la finalul tezei.

Adevărata contribuție a lucrării de față este, de fapt, în a doua parte a acesteia (capitolul 3., „Aspecte Practice”), unde s-au explorat conceptele prezentate în prima parte, pe parcursul dezvoltării unui sistem implementat pe baza microserviciilor. În această parte s-a vizitat utilizarea practicii Domain-Driven Design (DDD) pentru alegerea corectă a dimensiunii microserviciilor și implementarea pattern-urilor CQRS și Event Sourcing cu ajutorul acestora.

2.Aspecte teoretice

În acest capitol se vor prezenta aspectele teoretice de bază ale stilului arhitectural bazat pe microservicii, câteva pattern-uri arhitecturale ce pot facilita existența microserviciilor sau care pot beneficia de aceasta, și unele consecințe ale implementării unui sistem distribuit. Ideile prezentate în acest capitol sunt preluate în principal din cartea „Microservices în .NET Core", scrisă de Christian Horsdal Gemmelgaard, un arhitect/developer expert în materie de .NET. Alte idei sunt preluate din diferite articole publicate pe internet, sau din concluzii proprii.

2.1. Microservicii – concepte generale

Un microserviciu este un serviciu web dedicat și optimizat pentru expunerea unei singure capabilități către întregul sistem. Un sistem bazat pe microservicii, atunci când este scris bine, este maleabil, scalabil, rezilient și permite un timp scurt de implementare pentru serviciile individuale. Aceste trăsături sunt deseori greu de obținut în sistemele echivalente monolitice.

Christian Horsdal Gemmelgaard remarca, în cartea sa, intitulată „Microservices în .NET Core", ca nu exista o definiție acceptată pentru ce reprezintă un microserviciu. El conturează mai departe 6 trăsături pe care le consideră definitorii pentru acesta:

- Un microserviciu este responsabil pentru o singura capabilitate.
- Un microserviciu poate fi lansat în producție în mod individual.
- Un microserviciu consista dintr-unul sau mai multe procese.
- Un microserviciu poseda propria lui baza de date.
- Un microserviciu poate fi întreținut de o echipa de mici dimensiuni.
- Un microserviciu este ușor de înlocuit. (Gammelgaard, 2017)

Un microserviciu este responsabil pentru o singura capabilitate.

O capabilitate poate fi de doua feluri: o capabilitate de business – ce reprezintă capabilități de baza ale sistemului, sau o capabilitate tehnica – care reprezintă capabilități ajutătoare pentru alte microservicii.

A spune ca un microserviciu este responsabil pentru o singura capabilitate, nu este decât o extensie a „Single Responsibility Principle”, din cadrul principiilor SOLID, peste serviciile din cadrul sistemului distribuit.

Delimitarea capabilităților se poate face utilizând DDD („Domain Driven Design”).

(Gammelgaard, 2017)

Un microserviciu poate fi lansat în producție în mod individual.

Când un microserviciu se schimba, el trebuie să poată fi relansat în mod individual, fără a relansa orice alta parte a sistemului, iar celelalte servicii ale sistemului trebuie să poată continua funcționarea pe durata lansării în mediul de producție a serviciului modificat și după aceasta.

Aceasta caracteristica este importanta pentru un sistem bazat pe microservicii datorita faptului ca într-un astfel de sistem exista de regula un număr mare de servicii și fiecare în parte poate comunica cu multe altele. Daca aceasta proprietate nu s-ar respecta și lansarea serviciilor ar trebui făcută în grupuri sau într-o anumită ordine, atunci procesul ar deveni greoi și riscant. Acest fenomen ar trebui evitat, în favoarea lansărilor dese și de dimensiuni mici, care aduc la rândul lor riscuri mici.

Respectarea acestei caracteristici aduce anumite constrângeri; Spre exemplu orice modificare asupra interfeței unui microserviciu trebuie să fie „backward-compatible”, astfel încât serviciile care comunică cu acesta să nu trebuiască modificate la rândul lor. În cazul unui web API, aceasta se poate realiza prin versionare.

(Gammelgaard, 2017)

Un microserviciu consista dintr-unul sau mai multe procese.

Pentru a fi independent fata de restul sistemului, un microserviciu trebuie să fie încapsulat în unul sau mai multe procese separate.

Daca acest principiu nu ar fi respectat și am avea doua sau mai multe microservicii care rulează în cadrul aceluiași proces, atunci o eroare ce apare într-unul din ele se poate propaga și în celelalte, ceea ce duce la o reziliență scăzută și un cuplaj crescut. De asemenea, ar face foarte dificila, sau chiar aproape imposibilă, implementarea proprietății de a putea lansa în mod individual un microserviciu.

(Gammelgaard, 2017)

Un microserviciu posedă propria lui baza de date.

Aproape orice capabilitate de business necesită persistarea datelor într-o anumită forma. Un serviciu ce implementează o astfel de capabilitate, pentru a rămâne independent fața de restul sistemului, va trebui să posede propria lui baza de date.

Această proprietate permite unui serviciu să folosească baza de date cea mai potrivită pentru capabilitate implementată, ceea ce poate duce la beneficii în ceea ce privește costurile de dezvoltare, performanță și scalabilitate.

Persistarea datelor este de asemenea un detaliu de implementare și ca atare, ar trebui ascuns față de restul sistemului.

(Gammelgaard, 2017)

Un microserviciu poate fi întreținut de o echipa de mici dimensiuni.

Pentru ca aceasta să fie posibila, serviciul trebuie să fie de mici dimensiuni. Un serviciu care creste prea mult, devine greu de întreținut și modificat – similar unei aplicații monolitice; un astfel de serviciu ar trebui împărțit la rândul lui în altele mai mici.

(Gammelgaard, 2017)

Un microserviciu este ușor de înlocuit.

Echipa care întreține un microserviciu trebuie să fie capabilă să îl rescrie de la zero într-un interval de timp acceptabil. Similar cu principiul anterior, pentru a putea fi rescris cu ușurință, un microserviciu trebuie să fie de mici dimensiuni.

Același autor subliniază, în cartea lui, un număr de beneficii ale microserviciilor care aplica caracteristicile prezentate mai sus:

- Permit practica „continous delivery”.
- Permit un workflow eficient pentru programator.
- Sunt robuste prin design.
- Pot fi scalate independent unul față de celălalt.

(Gammelgaard, 2017)

Permit practica „continous delivery”.

„Continous delivery” este abilitatea de a lansa schimbări de orice tip – incluzând noi facilitați, reparații de bug-uri etc. – în mediul de producție într-un mod rapid, sigur și sustenabil.

Țelul acestei practici este de a transforma procesul de lansare, fie el al unui sistem la scară mare, cu un mediu de producție complex, într-unul predictibil, de rutină, ce poate fi executat la cerere.

Pentru a atinge acest țel, trebuie să ne asigurăm că codul este întotdeauna într-o stare din care poate fi lansat cu ușurință, chiar și în cazul unde avem un număr mare de programatori ce fac schimbări în fiecare zi. Astfel este eliminată nevoia pentru fazele de integrare, testare și întărire, cât și nevoia de „code freezes”. (What is Continuous Delivery?, fără an)

Stilul arhitectural bazat pe microservicii și conceptul de „continous delivery” sunt complementare. Adoptarea acestui concept este ușurata în cadrul sistemelor bazate pe această arhitectură prin orientarea către servicii care pot fi dezvoltate și modificate cu ușurința și rapiditate, pot fi testate utilizând teste automate, pot fi lansate în mod independent și pot fi operate în mod

eficient. Deși implementarea „continous delivery” este ușurată de aceste proprietăți, ea nu reiese direct din ele.

Beneficiile adoptării „continous delivery” includ agilitatea sporită pe planul business-ului, lansări sigure, reducerea riscurilor și calitate sporită a produsului.

Fără abilitatea de a lansa microserviciile în mod independent, rapid, și având costuri minime, implementarea unui sistem bazat pe microservicii va deveni foarte costisitoare foarte repede. Daca lansarea microserviciilor nu este automatizată, atunci munca manuala necesară pentru lansarea unui întreg sistem va fi copleșitoare.

(Gammelgaard, 2017)

Permit un workflow eficient pentru programator.

Acest beneficiu este dat de faptul ca microserviciile, atunci când sunt implementate bine, permit o foarte ușoara mentenanța. Din perspectiva programatorului, un microserviciu, posedând propria baza de date, având dimensiuni relativ mici și implementând o singură capabilitate, poate fi înțeles în întregime cu ușurință. Din perspectiva DevOps, un microserviciu este ușor mentenabil datorită faptului ca poate fi întreținut de o echipa restrânsă și poate fi lansat în mod individual.

În consecință, problemele din mediul de producție pot fi descoperite și adresate în mod rapid și cu riscuri minime, prin scalarea microserviciului cu pricina, sau prin lansarea unei noi versiuni ale acestuia.

(Gammelgaard, 2017)

Sunt robuste prin design și pot fi scalate independent.

O arhitectura distribuită bazată pe microservicii permite scalara componentelor individuale ale sistemului, în funcție de locul în care se produce fenomenul de „bottleneck”. În plus, microserviciile sunt orientate spre o forma asincronă de comunicare și spre toleranța la eșec

în cazurile în care comunicarea sincrona este necesară. Aceste proprietăți produc sisteme disponibile și ușor scalabile.

Pe lângă avantajele descrise mai sus, implementarea unui sistem bazat pe microservicii are și provocările ei. Costurile asociate dezvoltării unui sistem distribuit sunt bine cunoscute. Deși individual, fiecare serviciu este ușor de înțeles de către programator chiar de la prima vedere, sistemul, în întregimea sa, poate fi mai greu de înțeles decât un sistem monolitic echivalent. De asemenea, costurile inițiale de dezvoltare sunt mai ridicate decât în cazul arhitecturii monolitice și testarea poate fi mai greoaie.

Fiind compuse dintr-o multitudine de servicii, fiecare trebuind dezvoltat, lansat și întreținut în producție, costurile de DevOps pot fi mai ridicate decât în cazul unui sistem monolitic echivalent.

De asemenea, mutarea codului dintr-un serviciu în altul poate fi dificila; de aceea este necesară investiția de timp în alegerea corectă a scope-ului fiecărui microserviciu.

(Gammelgaard, 2017)

2.2. Domain-Driven Design (DDD)

„Domain-driven design este o abordare a proiectării sistemelor software bazată pe modelarea domeniului de business. Un pas important este identificarea limbajului folosit de experții în domeniu pentru a vorbi despre acesta. Totuși, se observă că limbajul folosit de experți nu este consecvent în toate cazurile.

În parți diferite ale domeniului, focusul se află pe diferite obiecte, deci un anumit cuvânt, cum ar fi *client* poate avea semnificații diferite în parți diferite ale domeniului. Spre exemplu, pentru o companie ce vinde fotocopiatoare, un *client,* în contextul departamentului de vânzări, poate fi o companie ce cumpără un număr de fotocopiatoare și poate fi reprezentat de printr-un agent de aprovizionare. În departamentul de servicii pentru clienți, un *client* poate fi un *end user*

care are probleme cu un fotocopiator. În timp ce modelăm domeniul companiei producătoare de fotocopiatoare, cuvântul *client* are semnificații diferite în contexte diferite.

Un *bounded context,* în cadrul DDD, este o parte dintr-un domeniu mai mare, în care cuvintele au aceeași semnificație. *Bounded contexts* sunt legate, dar în același timp diferite de conceptul de capabilitate de business. Un bounded context definește o zona în care limbajul este consecvent. Capabilitățile de business, pe de alta parte, reprezintă ceea ce este necesar ca business-ul să facă. În interiorul unui *bounded context,* ar putea fi necesar ca business-ul să facă mai multe lucruri. Fiecare din aceste lucruri se poate, probabil, defini ca o capabilitate de business.” (Gammelgaard, 2017)

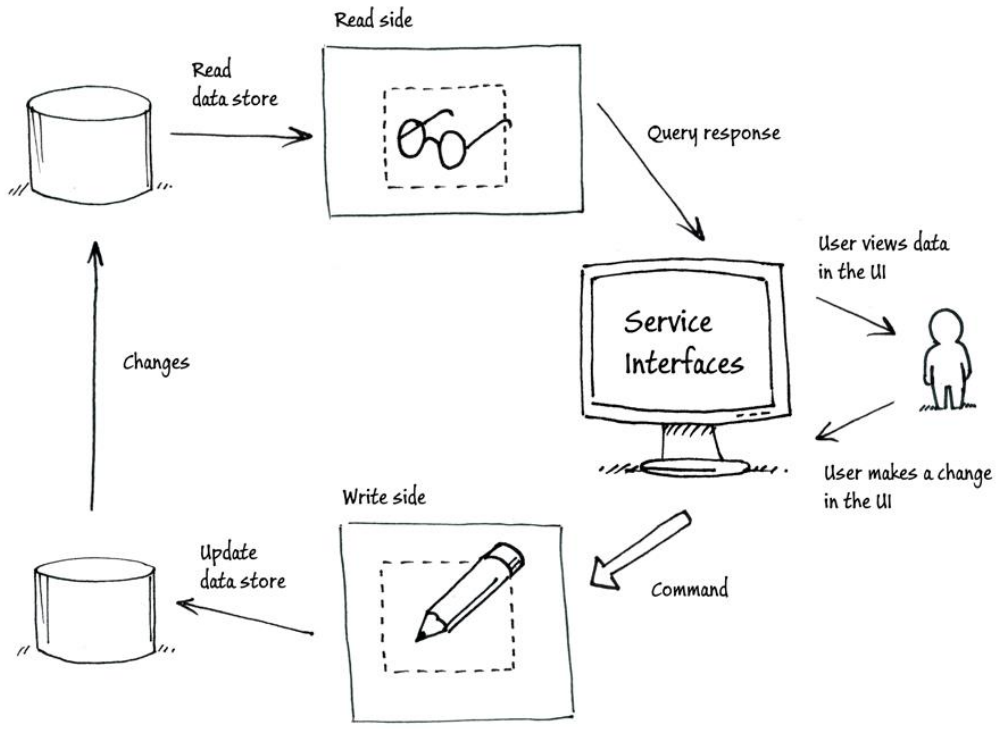
Gammelgaard vorbește în citatul dat despre limbajul folosit de experții în domeniu. Eric Evans, autorul cărții „Domain-Driven Design: Tackling Complexity în the Heart of Software”, numește acest limbaj „ubiquitous language”, același termen ce va fi folosit și în restul lucrării.

2.3. Command/Query Responsibility Segregation (CQRS)

„CQRS este pur și simplu crearea a doua obiecte acolo unde înainte era doar unul. Separarea are loc în funcție de tipul metodelor, ce pot fi ori comenzi, ori interogări (având aceeași definiție care a fost folosită de Mayer în *Command and Query Separation*: o comandă este o metodă care alterează starea obiectului, iar o interogare este o metoda care returnează o valoare).” (Young, 2010)

CQRS este un pattern arhitectural simplu care, atunci când este folosit corect, ajuta la obținerea scalabilității sistemului, gestionarea complexității sistemului și gestionarea regulilor legate de business aflate în schimbare.

Utilizarea acestui pattern ne permite scalarea și optimizarea individuală a componentelor, ceea ce este foarte util în situațiile în care, într-un sistem, numărul de operații de citire și numărul de operații de scriere nu sunt echilibrate (ceea ce se întâmplă frecvent).

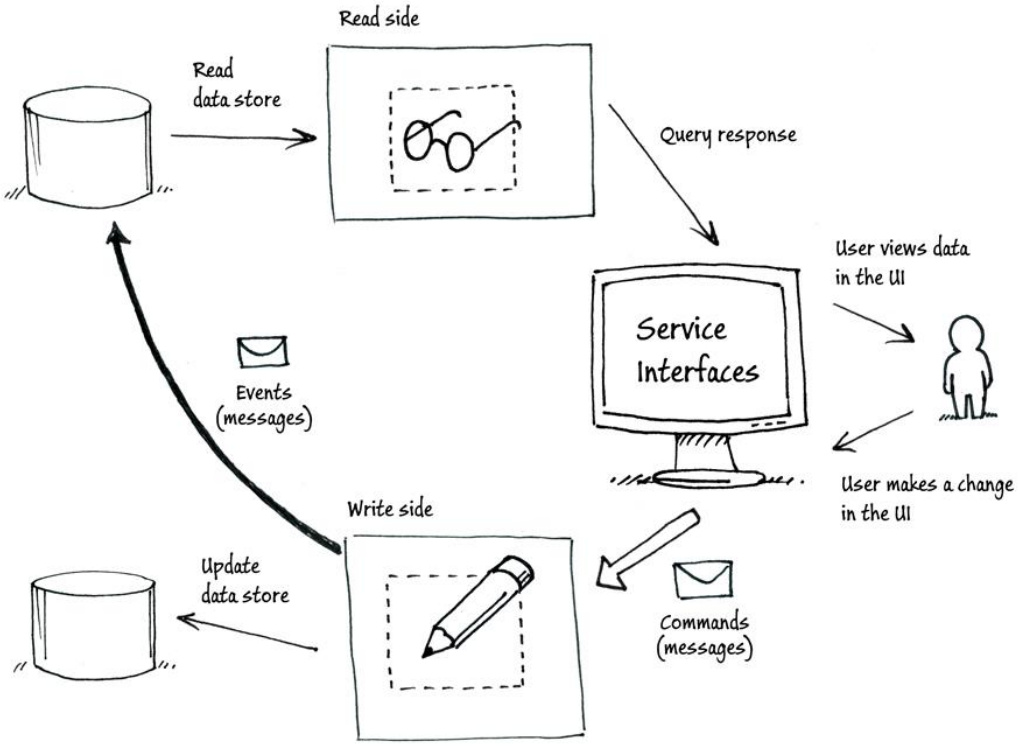


Figură 1 O posibila implementare a pattern-ului CQRS. (Microsoft)

CQRS permite și optimizarea modelelor pentru citire, respectiv scriere. Spre exemplu pe componenta responsabila cu gestionarea comenzilor, poate exista un model complex, ce este capabil să execute operații logice complexe, iar pe componenta responsabilă cu gestiunea interogărilor, modelul poate consista din DTO-uri simple, optimizate pentru citire. Segregarea componentelor duce, deci, la modele mai simple, mai ușor de întreținut și mai flexibile.

În plus, segregarea se poate extinde și la nivel de persistență a datelor, unde fiecare componentă poate avea propria ei baza de date optimizată pentru operațiile de care este responsabilă. În acest caz este nevoie de un mecanism de sincronizare intre cele doua, cum ar fi unul event-based (un astfel de sistem va fi demonstrat în capitolul următor).

(Microsoft)



Figură 2 Events în pattern-ul CQRS (Microsoft)

2.4. Event Sourcing

„Event sourcing se referă la stocarea stării curente a unui obiect sub forma unei serii de evenimente și reconstruirea stării prin derularea acestei serii de evenimente” (Young, 2010)

Deși implementarea pattern-ului *Event Sourcing* nu este necesara pentru realizarea unui sistem CQRS, cele doua concepte pot beneficia în urma colaborării.

Luând ca exemplu un sistem ce delimitează modelele dedicate operațiilor de scriere, respectiv de citire, cel mai probabil sincronizarea dintre acestea se va face într-o manieră *event-based*. În cazul acesta, un sistem de gestiune a evenimentelor va exista deja. Luând în considerare și faptul ca pe partea de scriere, modelul nu mai trebuie să suporte operații de citire, *event sourcing* poate fi o metodă foarte bună de a salva starea curentă a datelor.

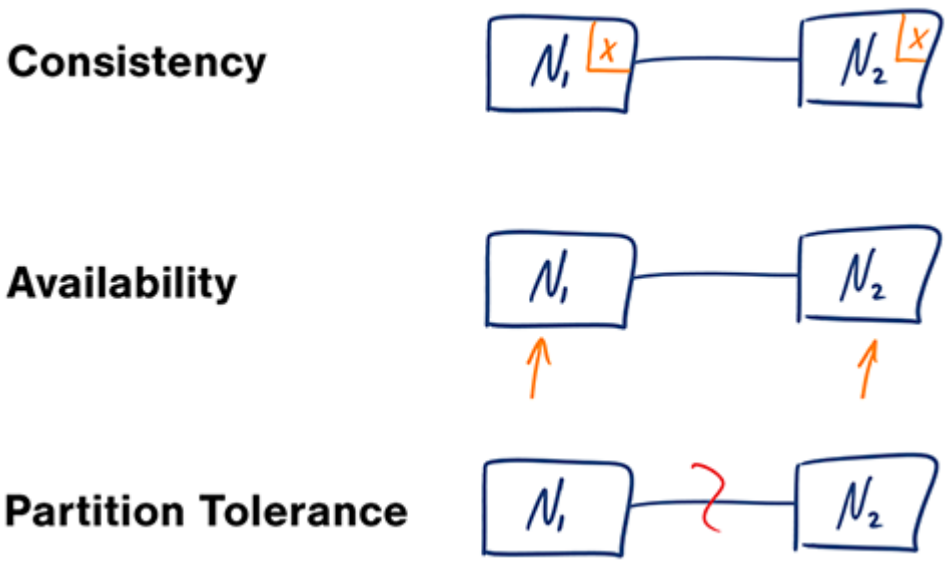
(Young, 2010)

2.5. Teorema CAP

Teorema CAP (numită și „Teorema lui Brewer”) susține că, într-un sistem distribuit, cum este acela bazat pe microservicii, la un moment dat, putem avea doar doua din următoarele garanții:

- Consistență (en. **C**onsistency) – este garantat faptul că o operație de citire va returna cea mai recentă versiune a informației cerute.
- Disponibilitate (en. **A**vailability) – este garantat faptul că un nod viu (care nu a întâmpinat erori) va returna un răspuns acceptabil într-un interval de timp acceptabil (nu o eroare sau un „timeout”).
- Toleranță la partiționare (en. **P**artition Tolerance) – este garantat că sistemul va continua să funcționeze chiar și după partiționarea rețelei.

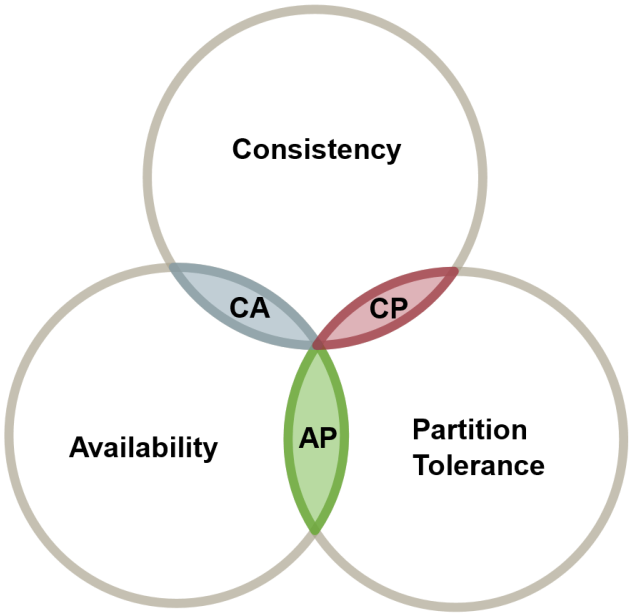
Teoretic se observa cu ușurința ca aceasta teorema ne oferă trei opțiuni: putem sacrifica ori garanția de consistență (cazul AP, după cum se vede în Figură 5), ori garanția de disponibilitate



Figură 3 Reprezentare a garanțiilor teoremei CAP (Greiner, 2014)

(cazul CP), ori garanția de toleranța la partiționare (cazul AP). Totuși, după cum vom vedea mai departe, în practica nu există atât de multe opțiuni.

O presupunere greșită, întâlnită des în contextul sistemelor distribuite, este aceea că rețeaua este de încredere. În actualitate, rețelele cad în mod frecvent și pe neașteptate. Trebuie, în



Figură 4 Combinațiile de garanții permise de teorema CAP (Erb)

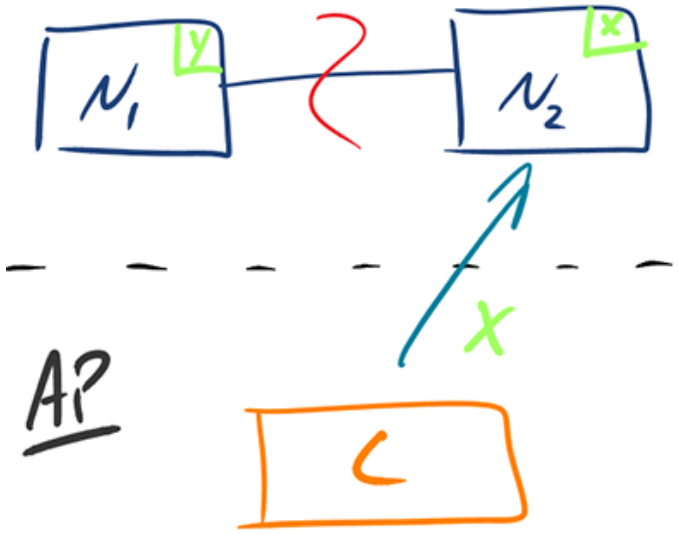
consecință să acceptăm ca aceste eșecuri vor avea loc și că nu avem niciun control asupra momentului în care vor avea loc.

A vând în vedere faptul ca rețelele, prin intermediul cărora colaborează componentele sistemului nostru distribuit, nu sunt de încredere, toleranța la partiționare trebuie să existe. Conform teoremei CAP, în cazul acesta ne rămâne alegerea între a sacrifica garanția de consistență (cazul AP), sau garanția de disponibilitate (cazul CP).

(Greiner, 2014)

Cazul AP – În cazul în care rețeaua este partiționată, o componentă interogată va returna cea mai recentă versiune a datelor cerute pe care o posedă, deși aceasta ar putea fi învechită. Un astfel de sistem va putea accepta și cereri de scriere care vor putea fi procesate doar în momentul în care partiționarea rețelei este reparată.

Din moment ce datele returnate pot fi învechite, sistemul nu este consistent. Această abordare este de dorit atunci când cerințele permit un anumit nivel de flexibilitate.

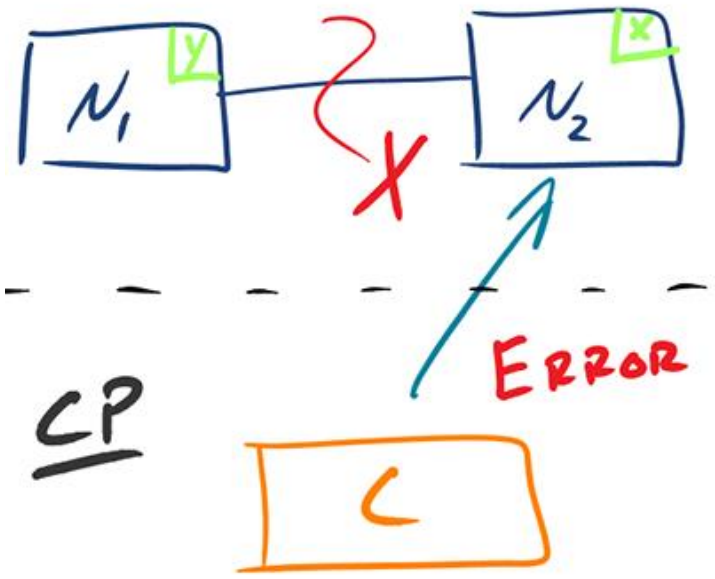


Figură 5 Cazul AP – Consecvența este sacrificată (Greiner, 2014)

(Greiner, 2014)

Cazul CP – O componentă a unui astfel de sistem, în condițiile în care este interogată pe durata unei partiționări ale rețelei, va aștepta un răspuns de la componenta ce deține autoritatea

asupra datelor cerute – ceea ce poate rezulta într-o eroare de tip „timeout”, sau va răspunde cu o altă eroare.



Figură 6 Cazul CP – Disponibilitatea este sacrificată (Greiner, 2014)

În ambele cazuri, pe durata partiției rețelei, sistemul nu este disponibil. Această abordare este potrivită în cazurile unde se cerințele sistemului impun operații de citire și scriere atomice.

(Greiner, 2014)

3. Aspecte practice

În acest capitol, se vor explora conceptele prezentate mai sus, prin intermediul aplicației „Venture”.

3.1. Venture – introducere

Aplicația „Venture” își propune să ofere utilizatorilor o platforma online ce să faciliteze colaborarea cu alți utilizatori în vederea enunțării și realizării proiectelor propuse de aceștia.

Audiența țintă a aplicației vor fi deci antreprenori începători ce doresc să pună bazele unui start-up, persoane ce vor să pună în practica o idee inovativă, dar nu posedă cunoștințele tehnice pentru a o face, persoane ce doresc să pornească un proiect de voluntariat, sau oricine altcineva care are o idee de proiect și dorește să o expună publicului, pentru a fi discutată și/sau pentru a căuta parteneri.

În scopul realizării țelului acestei aplicații, utilizatorii vor putea descrie un proiect nou, discuta proiectele create de ceilalți utilizatori, și vor putea gestiona o echipă aferentă proiectului.

Motivația din spatele alegerii temei aplicației a fost domeniul relativ simplu ce se cere modelat, ideal în scopul demonstrării unui sistem distribuit bazat pe microservicii.

3.2. Utilizarea DDD pentru identificarea microserviciilor.

Stilul arhitectural bazat pe microservicii este încă într-o stare incipientă și drept urmare nu există încă o metoda „acceptată” de implementare a unui astfel de sistem. Acest fapt oferă multă libertate în procesul de dezvoltare, ceea ce înseamnă multe decizii care trebuie făcute, în privința tehnologiilor alese și a detaliilor de implementare în general.

În rândurile ce urmează se vor explica dilemele întâmpinate în proiectarea aplicației și procesul gândirii ce a avut loc în scopul rezolvării lor.

În capitolul precedent s-a explicat legătura strânsa între arhitectura bazată pe microservicii și metoda DDD (Domain-Driven Design). În cartea sa, „Microservices în .NET Core", Christian Horsdal Gammelgaard subliniază faptul că un microserviciu trebuie să fie responsabil de o singură capabilitate a sistemului.

„Pentru a avea succes în implementarea microserviciilor, este important ca arhitectul să aleagă dimensiunile fiecărui microserviciu în mod corect. Dacă microserviciile sunt prea mari, atunci durata de dezvoltare a unor noi facilități, precum și implementarea reparațiilor pentru bug-uri, devine prea lungă. Dacă microserviciile sunt prea mici, atunci cuplajul dintre acestea tinde să crească. Dacă microserviciile au dimensiunile potrivite, dar granițele dintre ele au fost trasate în mod greșit, atunci cuplajul iar tinde să crească, iar un cuplaj crescut duce la o durată mai mare de dezvoltare. În alte cuvinte, dacă nu reușești să alegi corect dimensiunile microserviciilor, atunci vei pierde multe dintre beneficiile oferite de acestea. (...)

Principalul factor în identificarea și dimensionarea microserviciilor sunt capabilitățile de business; pe plan secundar se află capabilitățile tehnice. Urmarea acestor doi factori duce la microservicii ce se aliniază la lista caracteristicilor microserviciilor (prezentată în sub-capitolul 3.1. al acestei lucrări)"

(Gammelgaard, 2017)

Tot Gammelgaard, explică că identificarea capabilităților de business se poate realiza prin aplicarea Domain-Driven Design.

Un prim pas în aplicarea metodei DDD, considerat însăși fundația practicii, este identificarea unui „Ubiquitous Language" (din engleză, „limbaj comun"). Acesta este menit să servească ca un limbaj comun între experții în domeniu, și echipa de dezvoltare. Prin urmare, acesta trebuie să reprezinte domeniul de business, fără a conține termeni tehnici.

Luând în considerare aceste concepte, s-a creat următorul dicționar pentru domeniul aplicației „Venture":

- **User** – Un utilizator al aplicației.
- **Project** – Un proiect public, creat de un **user**.
- **Project Owner** – aferent unui **project**, reprezintă user-ul ce a creat proiectul, și are drepturi asupra lui.
- **Comment** – Un comentariu, adus de un **user**.
- **Chat** (în contextul unui **project**) – O colecție de **comments** aferente unui **project**, vizibile oricărui **user**.
- **Team** - Un grup de **users** aferent unui **project**, ce reprezintă echipa ce contribuie la realizarea **project**-ului.
- **Chat** (în contextul unui **team**) – O colecție de **comments** aferente unui **team**, vizibile oricărui **user** ce aderă la acel **team**.

Se observă deja o problema: termenul „**chat**” nu este consecvent – el reprezintă concepte diferite în contexte diferite (mai exact, în contextul unui project fața de contextul unui team).

Pentru a rezolva această problemă se poate și este indicat să se apeleze la un alt concept din cadrul Domain-Driven Design – Bounded Context (din engleză, „context delimitat”).

„Ideea unui Ubiquitous Language poate părea prea atotcuprinzătoare. Într-adevăr, așa este – în afara cazului în care lucrezi la o aplicație foarte simplă, un limbaj pur și simplu nu este de ajuns. Nu numai asta, dar un singur *model mamut*, va fi de asemenea prea mare, greoi, și se va transforma, într-un final, într-un *Big Ball of Mud* (din engleză, „bilă mare de noroi”). Aici intervine idea de Bounded Context.”

(Charlton, 2009)

Amintim din capitolul anterior ca un bounded context este, pe scurt, o parte a unui domeniu mai mare, pe suprafața căreia înțelesul oricărui cuvânt rămâne neschimbat. Identificarea acestor contexte ne poate ajuta și să delimităm microserviciile ce vor compune aplicația. Amintim de asemenea din capitolul anterior faptul că un bounded context nu coincide direct cu o capabilitate de business (despre care am stabilit că este factorul principal în delimitarea microserviciilor) ci poate conține mai multe capabilități. Dar simultan, o capabilitate de business nu se poate (sau nu

ar trebui) să se extindă peste mai multe contexte. Prin urmare, știm că unui bounded context trebuie să îi corespundă cel puțin un microserviciu.

Utilizând conceptul de bounded context, putem împarți domeniul în felul următor:

Project Context:

- **User** – Un utilizator al aplicației.
- **Project** – Un proiect public, creat de un **user**.
- **Project Owner** – aferent unui **project**, reprezintă user-ul ce a creat proiectul, și are drepturi asupra lui.
- **Comment** – Un comentariu, adus de un **user**.
- **Chat** – O colecție de **comments** aferente unui **project**, vizibile oricărui **user**.

Team Context:

- **User** – Un membru al echipei.
- **Team** – Un grup de **users** ce colaborează în sensul atingerii unui țel comun.
- **Project Owner** – Aferent unui **team,** reprezintă **user**-ul ce a creat echipa, și are drepturi asupra ei.
- **Comment** – Un comentariu, adus de un **user**.
- **Chat** – O colecție de **comments** aferente unui **team**, vizibile oricărui **user** ce aderă la acel **team**.

Observăm că prin împărțirea domeniului în contexte, am obținut doua sub-domenii, fiecare mai simplu decât cel de la care am pornit, în care toți termenii sunt consecvenți. Această împărțire ne permite de asemenea să avem un model mai specializat – observăm în acest sens faptul ca un user, în contextul unui team este definit ca un membru al unei echipei (în acest context, utilizatorul nu poate exista în afara echipei), iar un project owner reprezintă creatorul echipei (în acest context, noțiunea de project nu ne interesează).

Având în vedere aceste bounded contexts, se vor extrage următoarele capabilități de business: gestionarea proiectelor și gestionarea echipelor. În plus vom mai specifica încă o

capabilitate: autentificare și autorizarea utilizatorilor. Acestor capabilități le vor fi atribuite serviciile **Venture.Projects**, **Venture.Teams** respectiv **Venture.Users**.

3.3. Implementarea CQRS cu ajutorul microserviciilor

Am vorbit în capitolul precedent despre pattern-ul arhitectural CQRS (Command/Query Responsibility Segregation) și despre beneficiile aduse de acesta. Amintim că, atunci când este implementat corect, CQRS ajută la obținerea scalabilității sistemului, gestionarea complexității sistemului și gestionarea regulilor legate de business aflate în schimbare.

Utilizarea acestui pattern ne permite scalarea și optimizarea individuală a componentelor, ceea ce este foarte util în situațiile în care, într-un sistem, numărul de operații de citire și numărul de operații de scriere nu sunt echilibrate.

Considerăm, spre exemplu, cazul chat-ului din contextul proiectelor. Putem presupune că utilizatorul, pe parcursul folosirii aplicației, va face mult mai multe operații de citire (care au loc în mod pasiv, doar încărcând pagina proiectului) decât operații de scriere asupra comentariilor din cadrul chat-ului. Același lucru este valabil și pentru proiectele în sine. Un utilizator va vedea, probabil, mult mai multe proiecte decât va crea.

Prin urmare, am ajuns la concluzia că această componentă ar putea beneficia de separarea și de posibilitatea scalării individuale a componentelor de scriere, respectiv citire.

În acest scop, în cazul aplicației „Venture”, se va încerca implementarea pattern-ului CQRS cu ajutorul microserviciilor. Mai exact, componentele Venture.Projects și Venture.Teams se vor sparge mai departe în componentele Venture.ProjectWrite respectiv Venture.ProjectRead și Venture.TeamWrite respectiv Venture.TeamRead.

Astfel, componentele specializate pe operații de scriere vor modela domeniul astfel încât introducerea de date noi și modificarea datelor să fie făcute într-un mod cât mai simplu, iar pe partea de citire, vor exista doar niște simple DTO-uri (data transfer object – din engleză, „obiect pentru transferul datelor”), optimizate pentru operațiile de citire.

Aceste servicii, pentru a fi într-adevăr individuale și pentru a respecta criteriile definite în capitolul precedent, vor trebui să aibă propriile lor baze de date. Ceea ce înseamnă ca va fi necesar un mecanism de sincronizare a datelor între componentele de read și componentele de write.

Abordarea pe care o vom folosi pentru realizarea unui astfel de mecanism, va fi o forma de comunicare event-based între componente.

3.4. Aplicarea pattern-ului Event Sourcing

Implementarea mecanismului de sincronizare bazat pe evenimente, presupune existența modelării evenimentelor în cadrul domeniului. Având aceste evenimente, introducerea unui sistem pentru persistarea datelor de tipul Event Sourcing este foarte ușoară.

„Event sourcing se referă la stocarea stării curente a unui obiect sub forma unei serii de evenimente și reconstruirea stării prin derularea acestei serii de evenimente” (Young, 2010)

Spre exemplu, în cadrul serviciului Venture.ProjectWrite, un proiect ar putea fi stocat astfel:

Type	*Occurred On*	*Payload*
ProjectCreated	2017-06-21T00:22:19	Title: „MyProject”
ProjectDescriptionUpdated	2017-06-21T00:22:40	NewDesc: „MyDesc”
ProjectCommentPosted	2017-06-21T00:23:02	Comment: „Hello”
ProjectCommentPosted	2017-06-21T00:23:02	Comment: „Hi there”
ProjectDeleted	2017-06-21T10:53:36	

Tabel 1 Un exemplu simplificat de obiect stocat sub forma de sir de evenimente.

Acest tip de persistență este foarte util pentru serviciile de write ale aplicației. În contextul event sourcing, o operație de scriere presupune pur și simplu adăugarea unui nou eveniment, împreuna cu publicarea acestuia către restul sistemului.

3.5. Consecințele teoremei CAP

Aplicația „Venture”, fiind implementată printr-o arhitectură bazată pe microservicii, este un sistem distribuit, ceea ce înseamnă ca trebuie să se supună teoremei cap.

Teorema CAP (numită și „Teorema lui Brewer”) susține că, într-un sistem distribuit, la un moment dat, se pot garanta doar doua din următoarele trei proprietăți: consistență (**C**onsistency), disponibilitate (**A**vailability) și toleranță la partiționare (**P**artition Tolerance).

După cum a fost demonstrat în capitolul anterior, proprietatea de toleranță la partiționarea rețelei nu poate fi sacrificată, deoarece rețelele sunt, prin natura lor, nesigure – nu putem preveni, sau prevedea căderea lor.

Prin urmare mai rămâne de decis între a garanta disponibilitatea sistemului, sau consistența datelor oferite de sistem.

Considerăm cazul următor, în contextul aplicației Venture: Datorită unor factori ce se află în afara controlului nostru, comunicarea între microserviciile Venture.ProjectWrite și Venture.ProjectRead nu mai poate fi efectuată. În această durată, un utilizator scrie un comentariu în chatul public al unui proiect. La scurt timp, un alt utilizator deschide pagina acestui proiect. În acest moment, dacă am decide să garantăm consistența datelor în defavoarea disponibilității sistemului, microserviciul Venture.ProjectRead s-ar bloca, așteptând să primească versiunea cea mai recentă a chat-ului cu pricina de la Venture.ProjectWrite, care deține aceste informații, rezultând cel mai probabil într-o eroare de tip timeout. Astfel utilizatorul nu va putea vedea chat-ul proiectului, deoarece acesta ar fi „stale”.

Aceeași situație poate fi extinsă la întregul proiect. Serviciul Venture.ProjectRead nu poate garanta ca informațiile pe care le deține despre proiect sunt la curent – autorul ar fi putut modifica între timp titlul, sau ar fi putut corecta o greșeală gramaticală în descrierea acestuia. Chiar daca aceste acțiuni nu au avut loc în realitate, în contextul exemplului dat, serviciul nu poate garanta consistența datelor, deci decide să fie indisponibil.

În acest moment, utilizatorul va primi o pagină de eroare, datorită faptului ca nu pot fi afișate ultimele modificări, care pot, sau nu, să fi avut loc.

Lucrarea de față susține ca, în domeniul aplicației „Venture", cât și în majoritatea domeniilor de business (unde nu se impun operații atomice de citire și scriere), sacrificarea disponibilității în favoarea consistenței datelor nu își are rostul, după cum este prezentat mai departe.

În contextul exemplului dat mai sus, din prisma experienței utilizatorului, afișarea paginii de eroare este un lucru grav. De regulă, utilizatorii aplicațiilor web nu sunt răbdători – dacă după un refresh, sau doua, problema nu este rezolvată, utilizatorul va începe probabil să caute o alternativă pentru aplicația cu pricina și probabil o va și găsi. O pagină de eroare poate însemna clienți pierduți.

Considerăm același exemplu de mai sus, dar presupunând că aplicația „Venture" favorizează disponibilitatea înaintea consistenței. În acest caz, microserviciul Venture.ProjectRead, în situația în care primește o cerere de a oferi datele legate de proiect, nu va interacționa deloc cu Venture.ProjectWrite (vom vedea mai departe, în acest capitol, că de fapt nici nu știe de existența acestuia), ci va oferi imediat datele pe care le are, cu riscul ca acestea să fie „stale".

Pentru utilizator, această abordare înseamnă ca va vedea pagina proiectului, împreuna cu chat-ul aferent acestuia, conținând toate comentariile, mai puțin ultimul. În acest scenariu este foarte posibil ca utilizatorul să nu realizeze ca a avut vreodată loc o eroare. Comentariile noi, precum și orice modificări aduse proiectului în sine, vor fi afișate în momentul în care conexiunea poate fi refăcută, iar funcționarea aplicației rămâne afectată doar în mod minim.

În concluzie, în aplicația „Venture”, oriunde a fost cazul, s-a ales întotdeauna abordarea AP din teorema CAP, unde disponibilitatea este plasată înaintea consistenței.

3.6. Privire de ansamblu

În continuare, se va prezenta o privire de ansamblu asupra arhitecturii aplicației „Venture”, componentelor sale și asupra interacțiunii dintre ele.

După cum a fost stabilit în subcapitolul 3.2. („Utilizarea DDD pentru identificarea microserviciilor.”), microserviciile aplicației vor fi după cum urmează:

- **Venture.ProjectWrite** – responsabil cu crearea, ștergerea și modificarea proiectelor și a chat-urilor publice aferente.
- **Venture.ProjectRead** – responsabil cu redarea datelor aferente proiectelor.
- **Venture.TeamWrite** – responsabil cu gestiunea echipelor aferente proiectelor, precum și a chat-urilor private.
- **Venture.TeamRead** – responsabil cu redarea datelor aferente echipelor.
- **Venture.Users** – responsabil cu autentificarea și autorizarea utilizatorilor, precum și gestiunea datelor personale.

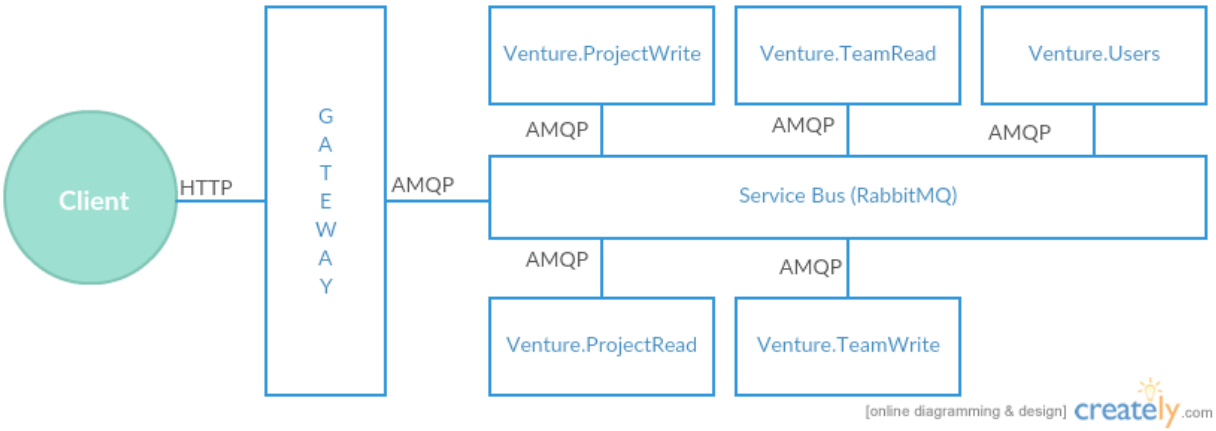
În plus, va mai fi necesar un al șaselea microserviciu – **Venture.Gateway** – ce va fi responsabil cu preluarea cererilor de la utilizatorii aplicației și trimiterea lor către microserviciile corecte, precum și întoarcerea către utilizator a datelor oferite de acestea.

Comunicarea directă între un client și unul dintre serviciile interne va fi imposibilă – orice cerere trebuie să treacă prin gateway. Acest fapt aduce un număr de beneficii.

În primul rând, ajută la încapsulare – implementarea bazată pe microservicii este ascunsă de client. Din exterior, api-ul serverului arată ca oricare altul, diferența între el și api-ul unui server construit pe o arhitectura monolitica este imperceptibilă.

În al doilea rând autentificarea utilizatorilor și autorizarea cererilor poate fi centralizată în serviciul de gateway. În consecință, orice formă de comunicare internă între microservicii poate fi considerată sigură.

Pentru implementarea comunicării externe, între gateway și client, s-a ales utilizarea mesajelor JSON peste HTTP. Pentru comunicarea internă, însă, s-a ales transmiterea mesajelor printr-un service bus, peste AMQP (**A**dvanced **M**essage **Q**ueueing **P**rotocol), după cum se poate vedea în Figură 7 Relațiile între componentele aplicației „Venture”. Această opțiune conferă un număr de avantaje ce vor fi discutate în următorul subcapitol.



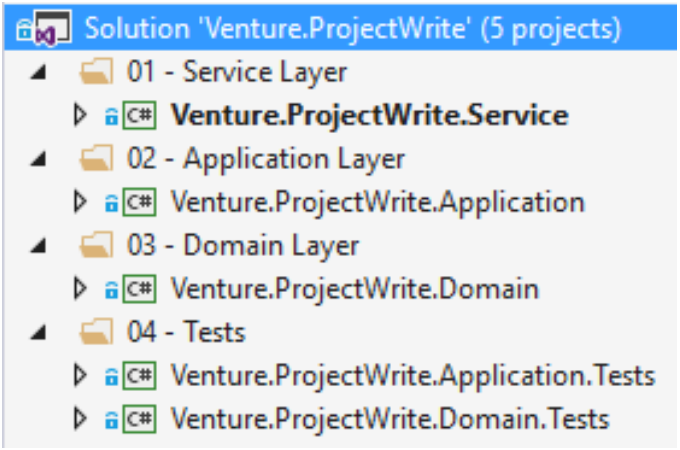
Figură 7 Relațiile între componentele aplicației „Venture”.

În primul rând, folosind un service bus pentru interacțiunea dintre microservicii, putem obține un cuplaj foarte scăzut – microserviciile individuale nu știu de existența celorlalte (incluzând chiar serviciul de gateway). Un serviciu pur și simplu se abonează, prin intermediul bus-ului, la mesajele de care este interesat, și publică, prin intermediul bus-ului, mesaje pentru celelalte servicii. Astfel, serviciul nu trebuie să cunoască nimic despre celelalte servicii, și acestea pot fi înlocuite cu ușurință, atât timp cât respectă un format standard al mesajelor.

În al doilea rând, utilizând această metodă, se poate obține o foarte ușoară scalabilitate. Dacă un anumit serviciu este suprasolicitat, se poate crea o altă instanță a lui și cupla la service bus. Datorită cuplajului foarte scăzut al microserviciilor din cadrul sistemului, operația aceasta se poate face repede și cu ușurință. Service bus-ul este capabil apoi să împartă sarcinile în mod „round-

robin” între consumatorii acestora. Astfel, obținem un sistem de distribuire a sarcinii (en. „load-balancing”) fără a mai depune niciun efort suplimentar.

Toate serviciile sunt construite după arhitectura 3-layer (din engleză, „trei straturi”), cu excepția serviciului Venture.Gateway, ce are doar două straturi. Denumirea exactă a straturilor diferă de la serviciu la serviciu, în funcție de nevoile fiecăruia, dar pe primul nivel se află întotdeauna stratul „Service” – responsabil cu oferirea serviciului propriu-zis către restul sistemului.



Figură 8 Structura „Layered” a serviciului ProjectWrite

3.7. Tehnologii folosite

În acest subcapitol se va intra mai adânc în detaliile de implementare ale aplicației „Venture”. Se vor discuta, în rândurile ce urmează, provocările și nevoile tehnice întâmpinate pe parcursul dezvoltării sistemului, precum și tehnologiile oferite de terțe părți pentru rezolvarea acestora.

.NET Core [1]

[1] .NET Core: https://www.microsoft.com/net/core#windowsvs2017

În primul rând a fost nevoie de un framework peste care să fie dezvoltate microserviciile. Deși stilul arhitectural bazat pe microservicii ne permite folosirea de tehnologii diferite pentru implementarea serviciilor diferite (și această proprietate a fost menționată ca un avantaj în capitolele anterioare), s-a decis ca, pentru consecvență, toate microserviciile să fie implementate utilizând limbajul C# și framework-ul .NET Core.

.NET Core este ultima versiune a platformei .NET, dezvoltată de compania Microsoft. Spre deosebire de versiunile mai vechi, .NET Core este open-source și cross-platform. Aceasta este o tehnologie bine cunoscută și nu se va intra în prea mult detaliu asupra ei.

Stratul „Service" al componentei Venture.Gateway este de fapt o aplicație ASP.NET ce oferă un WEB API accesibil oricărui client. Restul microserviciilor (cele „interne") folosesc pe nivelul de serviciu un „Console Application".

RabbitMQ și RawRabbit [2] [3]

S-a menționat în subcapitolul 3.6. („Privire de ansamblu") faptul că interacțiunea internă a microserviciilor se va efectua utilizând un service bus și protocolul AMQP.

În acest scop, s-a optat pentru utilizarea „RabbitMQ" – un message broker open-source construit de compania Pivotal, împreună cu RawRabbit – un client modern, high-level, compatibil cu .NET Core, pentru comunicarea peste platforma oferită de RabbitMQ.

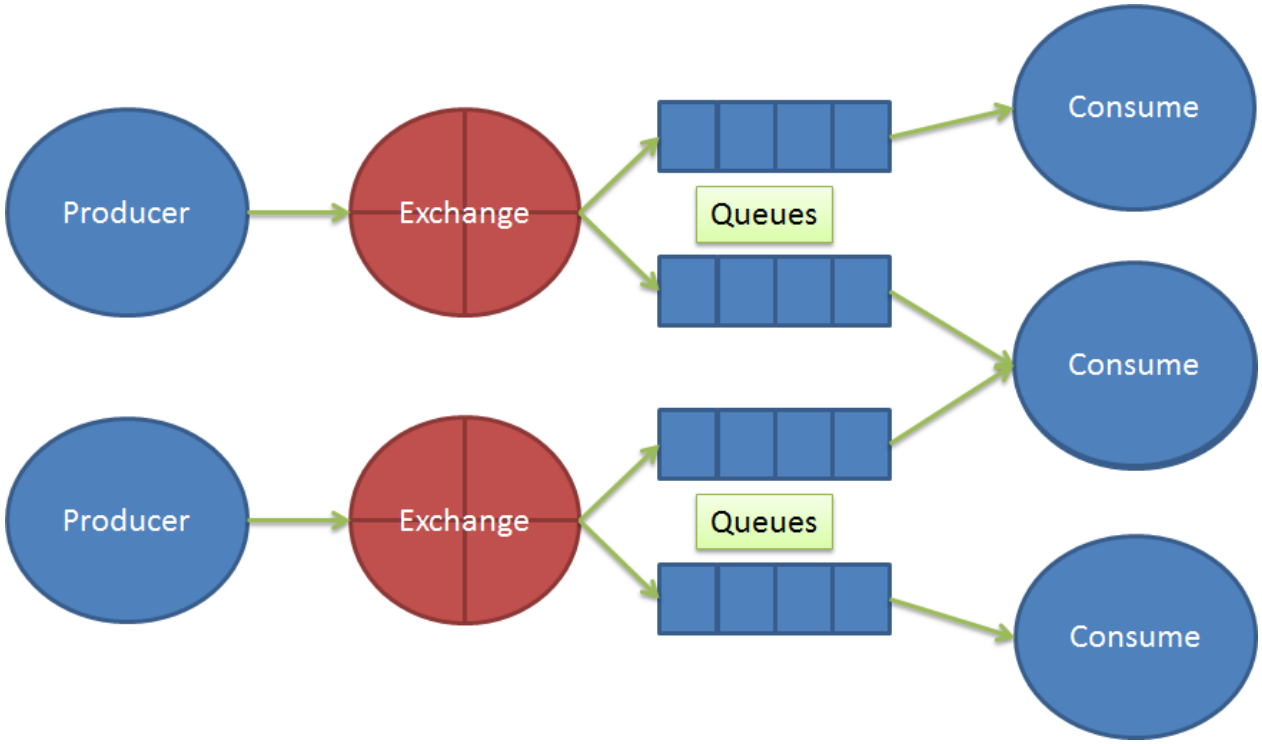
Pentru a înțelege cum funcționează comunicarea dintre microserviciile aplicației „Venture", va trebui să ne formăm o idee despre modul de lucru al message broker-ului RabbitMQ.

RabbitMQ funcționează implementând pattern-ul producător/ consumator. Mesajele de la un producător sunt rutate spre consumatorul sau consumatorii corecți utilizând o „topologie" constituită din exchanges (din engleză, „schimburi"), routing keys (din engleză, „chei de rutare") și queues (din engleză, „cozi").

[2] RabbitMQ: https://www.rabbitmq.com/
[3] RawRabbit: https://github.com/pardahlman/RawRabbit

Ca prim pas, producătorul publică un mesaj în cadrul unui exchange, împreună cu o cheie de rutare. În momentul în care aceasta publicare a avut loc cu succes, producătorul primește un răspuns de tip ack („acknowledged”) drept confirmare. Din acest moment se consideră ca RabbitMQ preia responsabilitatea pentru livrarea mesajului publicat către consumatorul sau consumatorii potriviți.

Odată ajuns în exchange, RabbitMQ trebuie să ruteze mesajul spre queue-urile potrivite.

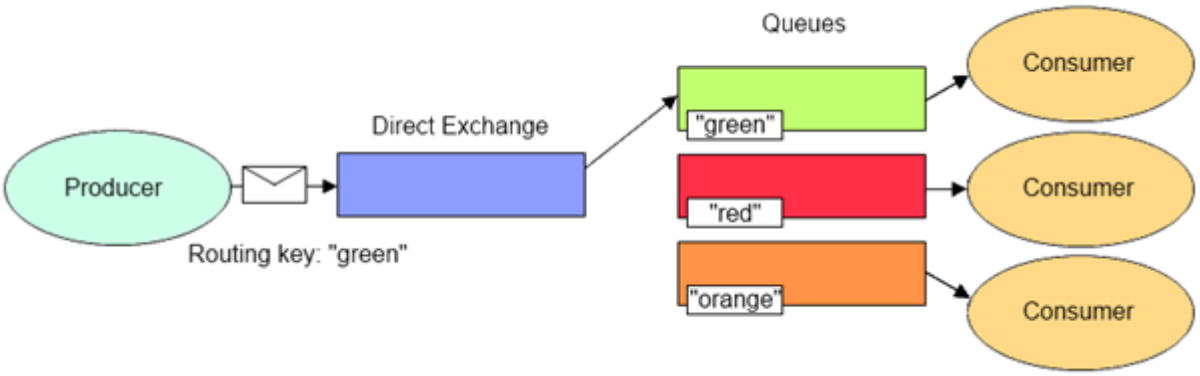


Figură 9 Privire de ansamblu asupra modului de funcționare al RabbitMQ (Freeman, 2013)

Aceasta se realizează prin intermediul cheilor de rutare. Fiecare queue deține propria lui cheie, iar RabbitMQ poate compara cheile de rutare ale queue-urilor cu cele ale mesajelor publicate. Condițiile exacte pentru potrivirea cozilor cu mesajele publicate depind de setările alese pentru exchange-ul în care au fost publicate.

RabbitMQ oferă patru tipuri de exchange: direct, fanout, topic și headers.

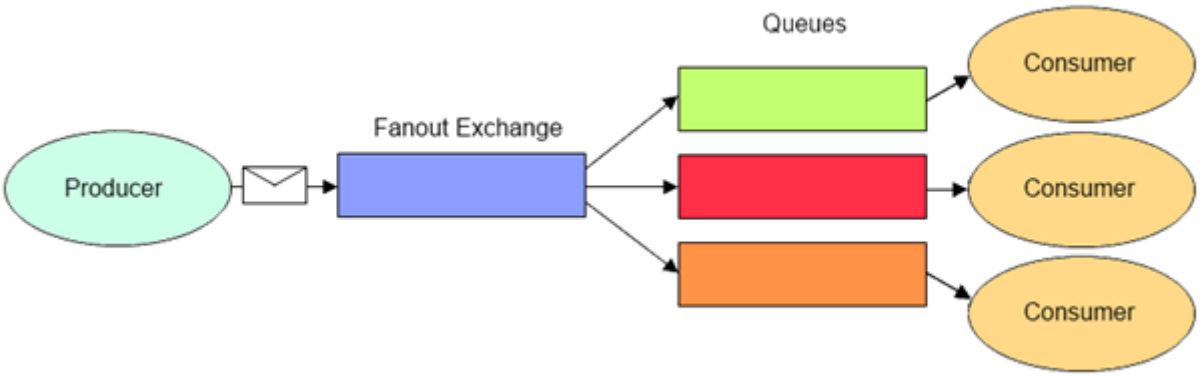
Exchange-urile de tip „direct” rutează mesajele către cozile abonate la acestea, ale căror cheie de rutare sunt egale cu cheia mesajului.



Figură 10 Direct Exchange (Greer, 2012)

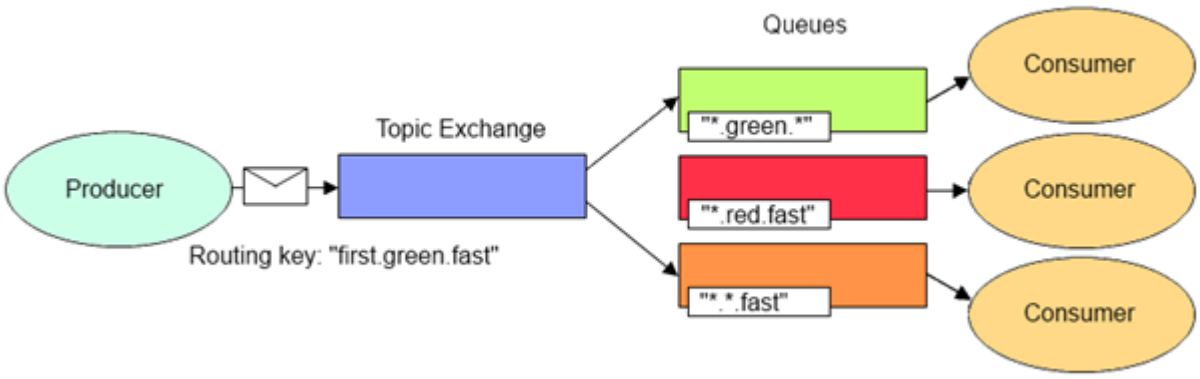
Exchange-urile de tip „fanout” (din engleză, „evantai”) distribuie mesajele publicate către toate cozile abonate, pur și simplu ignorând cheia de rutare. Acest tip de exchange este folositor pentru a implementa o rutare de tip broadcast.

Exchange-urile de tip „topic” (din engleză, „subiect”) sunt similare cu cele de tip fanout



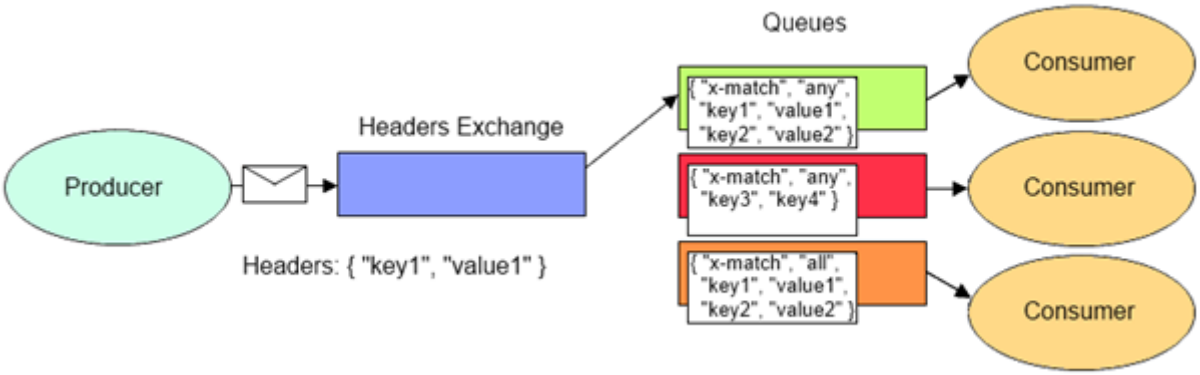
Figură 11 Fanout Exchange (Greer, 2012)

în sensul în care pot fi utilizate pentru a obține o rutare de tip broadcast, dar de această dată într-un mod mai selectiv. Un astfel de exchange va ruta mesajele spre cozile abonate ale căror cheie de rutare se potrivește perfect cu cheia mesajului, sau este o expresie regulată ce se potrivește cu cheia mesajului.



Figură 12 Topic Exchange (Greer, 2012)

Exchange-urile de tip „headers” funcționează într-un mod similar cu cele de tip topic. Diferența constă în faptul ca rutarea nu se face în baza cheii de rutare, ci în baza unor headere definite atât în cadrul mesajului cât și al cozilor abonate la exchange. Headerele sunt de tip cheie-valoare, și pentru a se ruta un mesaj la o coadă, măcar una, sau toate perechile cheie valoare din headerele mesajului trebuie să coincidă cu cele din headerele cozii. Numărul de perechi cheie-valoare ce trebuie să coincidă este specificat prin valoare unei chei speciale din cadrul cozii.



Figură 13 Headers Exchange (Greer, 2012)

SQL Server Express și Entity Framework [4] [5]

Pentru nevoile de stocare de date ale aplicației s-a ales o bază de date relațională pentru serviciile dedicate operațiilor de citire, și o bază de date ne-relațională pentru serviciile dedicate operațiilor de scriere, care vor apela la pattern-ul „Event Sourcing”.

Ca bază de date relațională, s-a ales SQL Server Express, un server SQL dezvoltat de compania Microsoft, împreună cu ORM-ul („Object Relational Mapping”) dezvoltat de aceeași companie.

Aceste tehnologii au fost alese datorită maturității lor – Entity Framework a fost o parte integrală a framework-ului .NET până la versiunea a șasea a acestuia. Deși poate fi considerat un framework heavy-weight, Entity Framework este o tehnologie high-level ce simplifică mult procesul de dezvoltare.

Redis[6]

În cadrul pattern-ului „Event Sourcing”, o bază de date relațională nu aduce prea multe beneficii – una de tip NoSQL fiind mult mai potrivită acestui context.

În acest sens, pentru a satisface nevoile de stocare ale microserviciilor ce implementează acest pattern, s-a ales Redis – un data store în-memory, ne-relațional, ce oferă o viteză foarte buna atât pentru operațiile de scriere, cât și pentru cele de citire.

Deși, folosind configurarea standard, natura în-memory a Redis-ului poate duce la probleme de durabilitate – o pana de curent, spre exemplu, poate cauza pierderea datelor, care poate cauza mai departe lipsa de sincronizare a datelor din diferite microservicii ale sistemului – el poate fi configurat să salveze datele pe disc la orice modificare, ceea ce ne oferă o durabilitate mai bună cu prețul unei mici scăderi în performanță. Chiar și plătind acest preț, Redis este încă una dintre cele mai performante baze de date de pe piață.

[4] SQL Server Express: https://www.microsoft.com/en-us/sql-server/sql-server-editions-express
[5] Entity Framework: https://docs.microsoft.com/en-us/ef/
[6] Redis: https://redis.io/

3.8. Detalii de implementare

Până acum, s-a discutat despre aplicația „Venture” doar la un nivel conceptual. În acest subcapitol se va intra în detaliile de implementare ale componentelor considerate mai interesante.

În primul rând, pe lângă proiectele aferente microserviciilor, s-a mai creat un al șaptelea proiect de suport, intitulat **Venture.Common**. Acesta are rolul de a reduce duplicarea codului întinderea sistemului. Acest proiect nu reprezintă un microserviciu de sine stătător, ci o simplă librărie de clase distribuită serviciilor sistemului sub forma unui pachet NuGet.

Rutare

Una din funcționalitățile încapsulate de acest pachet este sistemul de rutare al mesajelor între microservicii, care poate fi considerat „inima” aplicației.

În contextul aplicației „Venture”, s-au identificat trei tipuri de mesaje, diferite prin modul în care acestea trebuie gestionate de către service bus:

- **Comenzi** („Commands”) – Comenzile se referă la cele definite în pattern-ul CQRS; ele sunt cele care cer modificarea stării sistemului, și sunt de tip „fire-and-forget” (adică nu necesită un răspuns, dincolo de confirmarea faptului că au fost preluate de bus). Comenzile trebuie să fie rutate către un singur consumator (microserviciu), chiar daca există mai mulți abonați la același tip de comandă (pentru a preveni modificarea stării sistemului de doua ori).
- **Interogări** („Queries”) – Interogările se referă, de asemenea, la cele definite în pattern-ul CQRS; acestea reprezintă mesajele care cer un răspuns și nu pot modifica starea sistemului. Acest tip de mesaje va trebui implementat utilizând paradigma RPC („Remote Procedure Call”). Din fericire, clientul RawRabbit deja implementează o astfel de funcționalitate. Similar comenzilor, un mesaj de tip interogare trebuie rutat spre un singur consumator, chiar dacă există mai mulți abonați.

- **Evenimente** („Domain Events”) – Evenimentele sunt mesaje care indică faptul că starea sistemului s-a schimbat într-un anumit moment din trecut. Pentru a fi clară semnificația aceasta, denumirile evenimentelor vor conține, prin convenție, verbe la timpul trecut – spre exemplu „ProjectCreated” (spre deosebire de comenzi sau interogări, care semnifică o cerere – spre exemplu „CreateProject” sau „GetProject”). Evenimentele se comportă diferit fața de comenzi și interogări din punctul de vedere al rutării; acestea trebuie trimise într-o manieră „broadcast” – adică toți consumatorii abonați la un tip de eveniment trebuie să îl primească. Pentru implementarea acestora, exchange-urile de tip topic, sau fan-out sunt cele mai potrivite.

Pentru rutarea acestor mesaje, se folosesc trei exchange-uri, intitulate **Venture.Commands**, **Venture.Queries** și respectiv, **Venture.EventFeed**.

Pentru mesajele de tip comandă sau interogare, rutarea funcționează aproape identic – diferența constă în faptul că interogările necesită un răspuns. Mesajele de acest tip sunt publicate în exchange-ul potrivit lor (Venture.Commands pentru comenzi și Venture.Queries pentru interogări) cu o cheie de rutare echivalentă cu numele comenzii/interogării (spre exemplu „CreateProjectCommand”). Pentru fiecare tip (nume) de mesaj se creează un singur queue la care se vor abona toți consumatorii, intitulat după convenția Venture.<numele_mesajului>.Queue (ex. „Venture.CreateProjectCommand.Queue”). RabbitMQ va distribui apoi mesajele către consumatorii abonați în mod round-robin.

Pentru mesajele de tip eveniment, mesajele se plasează similar comenzilor sau interogărilor, în exchange-ul Venture.EventFeed, cu cheia de rutare egală cu numele evenimentului (spre exemplu, „ProjectCreatedEvent”). Fiecare consumator va crea mai apoi propriul queue, utilizând aceeași cheie de rutare și titlul format după convenția Venture.<numele_serviciului>.<numele_mesajului>.Queue.<guid>. Numele serviciului este adăugat pentru lizibilitate, iar guid-ul de la final este adăugat pentru unicitate. Astfel, evenimentele sunt rutate spre toți consumatorii în mod broadcast.

Publicarea și abonarea la mesaje se efectuează prin intermediul unei implementări a interfeței IBusClient (oferită de RawRabbit și specializată pentru aplicația Venture folosind extensii proprii).

```
public static class BusClientExtensions
    {
        (...)

        public static void PublishCommand<TCommand>(
              this IBusClient bus,
              TCommand command)
              where TCommand : class, ICommand
        {...}

        /// <summary>
        /// Attaches a handler to a specific command.
        /// </summary>
        public static void SubscribeToCommand<TCommand>(
              this IBusClient bus,
              ICommandHandler<TCommand> commandHandler)
              where TCommand : class, ICommand
        {...}

        /// <summary>
        /// Publish a query.
        /// This is a synchronous RPC call.
        /// If no subscribers answer for a number of seconds,
        /// then a timeout exception will be thrown.
        /// </summary>
        public static TResult PublishQuery<TQuery, TResult>(
              this IBusClient bus,
              TQuery query)
              where TQuery : class, IQuery<TResult>
        {...}

        /// <summary>
        /// Attaches a handler to a specific query.
        /// </summary>
        /// <typeparam name="TResult">The expected result type</typeparam>
        public static void SubscribeToQuery<TQuery, TResult>(
              this IBusClient bus,
              IQueryHandler<TQuery, TResult> queryHandler)
              where TQuery : class, IQuery<TResult>
        {...}

        public static void PublishEvent<TEvent>(
              this IBusClient bus,
              TEvent domainEvent)
              where TEvent : DomainEvent
        {...}

        /// <summary>
        /// Attach a handler to a specific event.
        /// </summary>
        public static void SubscribeToEvent<TEvent>(
              this IBusClient bus,
              IEventHandler<TEvent> eventHandler)
              where TEvent : DomainEvent
        {...}
    }
```

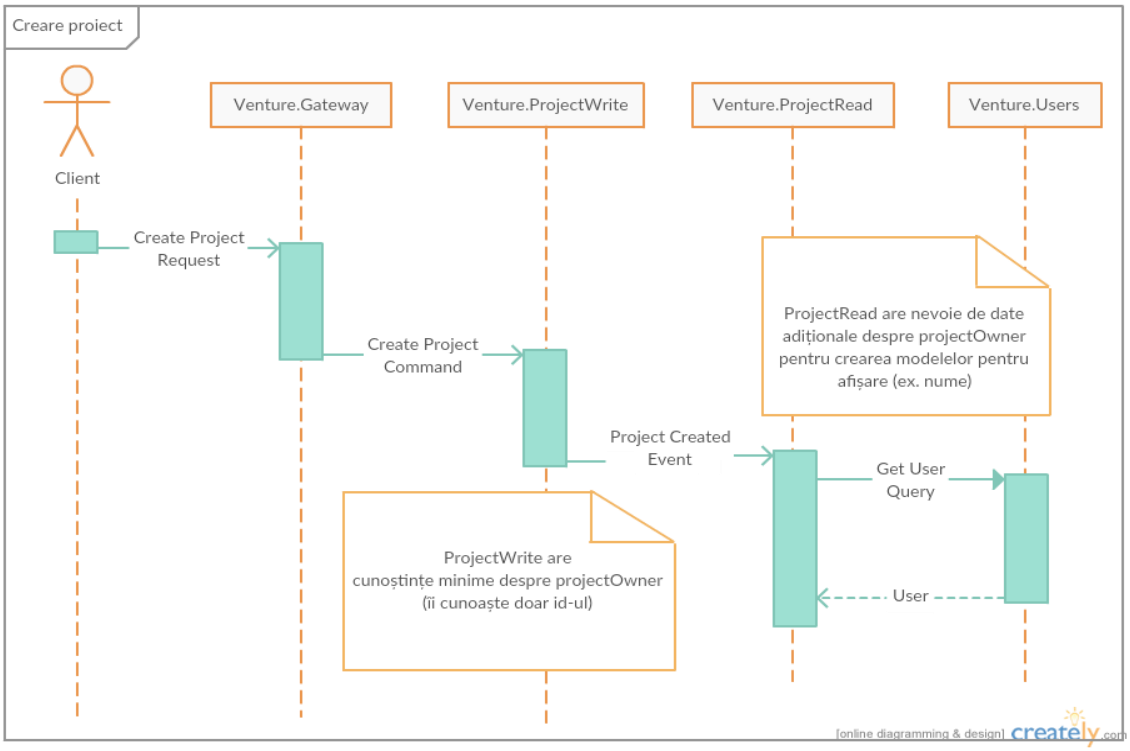
Figură 14 Extensiile pentru interfața IBusClient.

Sincronizare

Perechile de microservicii axate pe citire/scriere (ProjectRead și ProjectWrite respectiv, TeamRead și TeamWrite) dețin propriile lor baze de date conform proprietăților descrise în subcapitolul 2.1. („Microservicii – concepte generale”). Din acest motiv integrarea acestora nu va putea fi făcută la nivel de data store (cum nici nu este recomandat), ci la nivel de business logic – mai exact, utilizând un sistem de sincronizare event based.

Astfel, flow-ul aplicației pentru o operație de scriere va fi în felul următor:

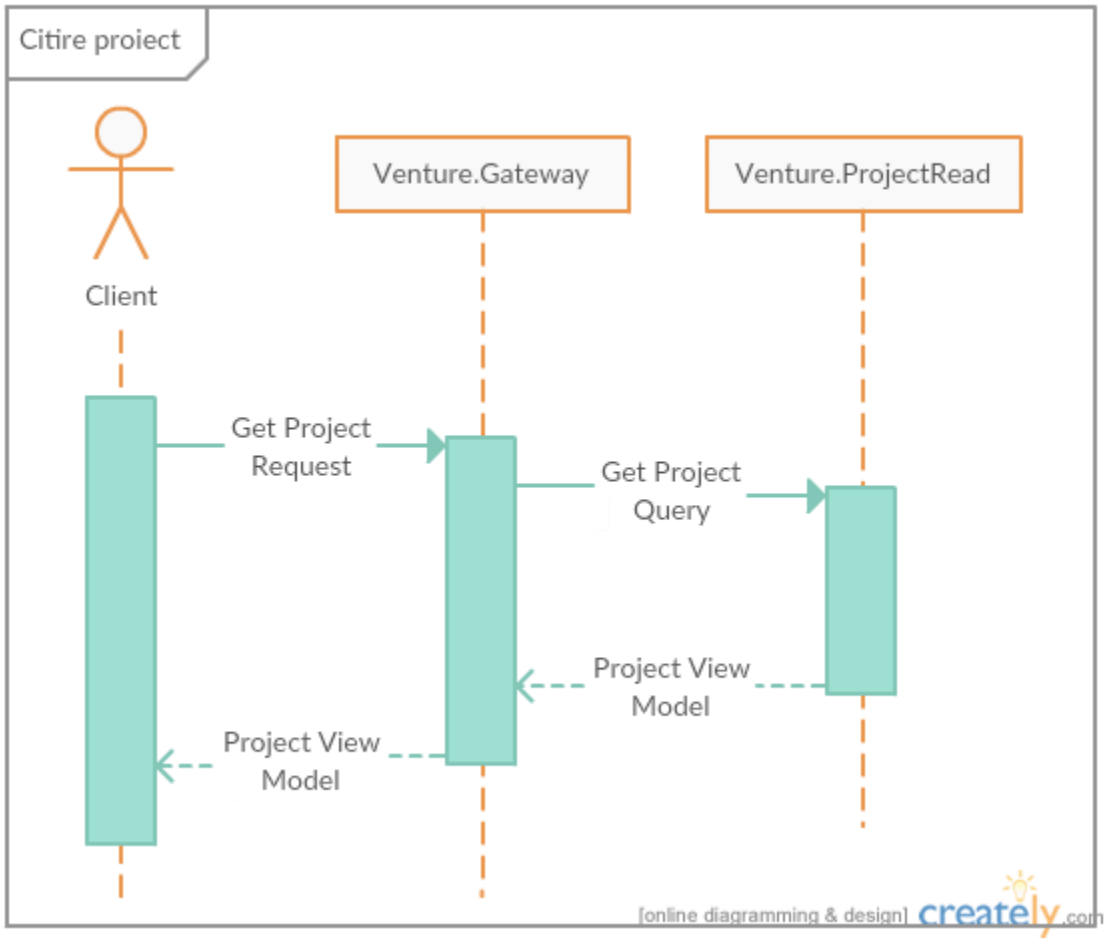
1. Un client va face o cerere HTTP de tip Post sau Patch către api-ul oferit de Gateway.
2. Mai departe gateway-ul, după ce realizează autorizarea utilizatorului, va traduce cererea într-un mesaj de tip Comandă care va fi publicat în service bus.
3. Serviciul de scriere, care va fi abonat, prin intermediul IBusClient, la acest tip de comandă, o va procesa și va înregistra datele noi sub formă de evenimente.
4. Serviciul de scriere publică mai departe evenimentele ce descriu schimbările efectuate, prin intermediul IBusClient, care sunt preluate de serviciul de citire.
5. Serviciul de citire trece evenimentele la care este abonat printr-un „Denormalizer” – o clasă specială care are rostul să traducă evenimentele în clase de tip DTO, care apoi sunt stocate într-o bază de date relațională.



Figură 15 Operația de scriere a unui proiect nou.

O operație de citire este apoi mai simplă:

1. Un client efectuează o cerere HTTP de tip Get către api-ul oferit de Gateway.
2. Gateway-ul traduce această cerere într-un mesaj de tip Interogare ce este publicat prin intermediul IBusClient – Spre deosebire de comenzi și evenimente, publicarea interogărilor este tratată ca un RPC („Remote Procedure Call”) și este efectuată în mod sincron.
3. Serviciul de citire abonat la această interogare răspunde cu DTO-ul pregătit deja în baza lui de date.
4. Gateway-ul preia acest DTO și îl întoarce clientului, completând operația de citire.



Figură 16 Operația de citire a unui proiect existent.

Event Sourcing

O altă componentă a pachetului Venture.Common este implementarea pattern-ului de event sourcing. În această componentă există doua elemente cheie: interfața IEventStoreable și clasa abstractă EventRepository. În rândurile ce urmează, se va explica în detaliu modul de funcționare al acestora.

Interfața IEventStoreable, marchează faptul ca orice clasa ce o implementează este capabilă să încarce o anumită stare dintr-o serie de evenimente, numită istoric. În plus, din partea unei clase ce implementează această interfață se așteaptă ca orice modificare nouă asupra stării să fie înregistrată sub forma unui eveniment în lista „UncommitedChanges”.

```
public interface IEventStoreable
{
    /// <summary>
    /// Gets the version of the last event that changed the event storeable's state.
    /// </summary>
    int Version { get; }

    /// <summary>
    /// Gets a list of events that changed this event storeables state,
    /// but were not yet marked as commited.
    /// </summary>
    IList<DomainEvent> UncommitedChanges { get; }

    /// <summary>
    /// Mark all uncomitted changes as commited.
    /// </summary>
    void MarkChangesAsCommited();

    /// <summary>
    /// Load the event storeable object’ state from a history of events.
    /// </summary>
    /// <param name="history"></param>
    void LoadFromHistory(IEnumerable<DomainEvent> history);

    /// <summary>
    /// Delete the event storeable object.
    /// </summary>
    void Delete();
}
```

Figură 17 Interfața IEventStoreable

Clasa abstractă EventRepository implementează un simplu pattern repository, pentru o entitate generică ce implementează IEventStoreable. În Figură 18, se observă că repository-ul cere

de la containerul IoC al aplicației o instanță a IEventStore – care oferă persistența evenimentelor și o instanță a IBusClient – prin care va publica evenimente noi.

```
public abstract class EventRepository<TEntity> : IRepository<TEntity>
        where TEntity : class, IEntity, IEventStoreable, new()
    {
        private readonly IEventStore _eventStore;
        private readonly IBusClient _bus;

        protected EventRepository(IEventStore eventStore, IBusClient bus)
        {
            Guard.AgainstNullArgument(nameof(eventStore), eventStore);
            Guard.AgainstNullArgument(nameof(bus), bus);

            _eventStore = eventStore;
            _bus = bus;
        }

              (...)

        public TEntity Get(Guid id)
        {
            var entity = new TEntity();
            var history = _eventStore.GetEvents(id).OrderBy(e => e.Version);
            entity.LoadFromHistory(history);

            if (entity.Id != id)
            {
                // no entity found with specified id
                return null;
            }

            return entity;
        }

              (...)

        public void Update(TEntity entity)
        {
            Guard.AgainstNullArgument(nameof(entity), entity);

            var uncommitedChanges = entity.UncommitedChanges;
            foreach (var change in uncommitedChanges)
            {
                _eventStore.Raise(change);
                _bus.PublishEvent(change);
            }

            entity.MarkChangesAsCommited();
        }
    }
```

Figură 18 Clasa abstractă EventRepository

4. Posibile îmbunătățiri

O prima posibilă îmbunătățire a proiectului este o mai buna organizare a pachetului Venture.Common, și împărțirea acestuia în mai multe pachete modulare.

La momentul de față, Venture.Common încapsulează toată funcționalitatea care este necesară în fiecare microserviciu, dar și unele funcționalități ce nu sunt utilizate decât în anumite servicii. Asta înseamnă, spre exemplu, că microserviciul Venture.ProjectRead, care nu folosește event sourcing pentru stocarea datelor, va trebui să instaleze un pachet ce oferă această funcționalitate. Acesta este o violare a „Interface-segregation principle”, unul dintre principiile SOLID, care spune că un client nu ar trebui să fie forțat să depindă de metode pe care nu le folosește.

O a doua îmbunătățire posibilă a aplicației ar fi implementarea unui mecanism de sincronizare între mai multe instanțe ale serviciilor dedicate operațiilor de scriere.

În implementarea curentă, pot exista oricâte instanțe ale serviciilor de citire sunt necesare pentru ducerea cererii clienților, dar nu poate exista decât un singur serviciu de scriere. Deși nu ar fi grea instanțierea unui nou serviciu de scriere în sistem, iar mecanismul de rutare ar împărți comenzile în mod egal între instanțele existente, acestea nu ar putea să se sincronizeze între ele. Aceasta ar putea duce, spre exemplu, la o situație în care un serviciu de scriere primește o comandă de a adăuga un nou comentariu în chat-ul unei echipe de care el nu știe că există, deoarece comanda de cerere a acelei echipe a fost preluată de pe altă instanță a lui.

O a treia posibilă îmbunătățire a sistemului ar fi implementarea unui mecanism care să recunoască scenariul în care un anumit serviciu este suprasolicitat și să creeze în mod automat o noua instanță a acestuia. Un astfel de sistem ar îmbunătăți cu mult ușurința scalării sistemului.

Concluzii

După cum am arătat pe parcursul lucrării, stilul arhitectural bazat pe microservicii rezolvă multe dintre problemele și provocările de care suferă un sistem echivalent construit pe o arhitectură monolitică. Printre aceste probleme se numără lipsa de flexibilitate și dificultatea modificării sistemului, complexitatea sporită și dificultatea de înțelegere a unui monolit de proporții mari de către un programator nou, costurile de mentenanță ce cresc exponențial cu dimensiunea monolitului și dificultatea în scalarea sistemului.

Acestea fiind spuse, majoritatea dificultăților ce apar în dezvoltarea aplicațiilor monolitice nu sunt măsurabile decât atunci când sistemul capătă proporții mari. Pentru majoritatea sistemelor dezvoltate în prezent, cel mai probabil, o arhitectură monolitică este suficientă.

Cu alte cuvinte, într-o arhitectură monolitică, complexitatea sistemului este mică, inițial, dar crește exponențial cu dimensiunile acestuia, pe când într-o arhitectură bazată pe microservicii, deși complexitatea inițială este comparativ mare, urmează o curbă de creștere mult mai blândă, proporțional cu dimensiunile sistemului.

În consecință, arhitectul care dorește să implementeze un sistem de proporții mari, odată ce depășește efortul inițial, are mult de câștigat de pe urma utilizării microserviciilor.

Bibliografie

Badola, V. (2015, Noiembrie 30). *Microservices architecture: advantages and drawbacks.* Preluat de pe cloudacademy.com: https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/

Charlton, d. (2009, Februarie 13). *Domain Driven Design: A Step by Step Guide - Part 1.* Preluat de pe developerfusion.com: http://www.developerfusion.com/article/9794/domain-driven-design-a-step-by-step-guide-part-1/

Erb, B. (fără an). *Concurrent Programming for Scalable Web Architectures.* Preluat de pe berb.github.io: http://berb.github.io/diploma-thesis/original/061_challenge.html

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity în the Heart of Software.* Addison-Wesley Professional.

Freeman, J. (2013, Iulie 13). *Examining how NServiceBus configures and uses RabbitMQ / AMQP*. Preluat de pe joe.blog.freemansoft.com: http://joe.blog.freemansoft.com/2013/07/

Gammelgaard, C. H. (2017). *Microservices în .NET Core.* Manning.

Gibson, S. (2015, Martie 13). *Monoliths are Bad Design... and You Know It.* Preluat de pe thoughtworks.com: https://www.thoughtworks.com/insights/blog/monoliths-are-bad-design-and-you-know-it

Gooen, O. (fără an). *Advantages and Disadvantages of a Monolith Application.* Preluat de pe impact.hackpad.com: https://impact.hackpad.com/Advantages-and-Disadvantages-of-a-Monolith-Application-ZlrQRl3LHCg

Greer, D. (2012, Martie 28). *RabbitMQ for Windows: Exchange Types.* Preluat de pe lostechies.com: https://lostechies.com/derekgreer/2012/03/28/rabbitmq-for-windows-exchange-types/

Greiner, R. (2014, August 14). *CAP Theorem: Revisited.* Preluat de pe http://robertgreiner.com: http://robertgreiner.com/2014/08/cap-theorem-revisited/

Hao, A. (2016, Octombrie 4). *Ubiquitous Language & the joy of naming.* Preluat de pe blog.carbonfive.com: http://blog.carbonfive.com/2016/10/04/ubiquitous-language-the-joy-of-naming/

Microsoft. (fără an). *Introducing the Command Query Responsibility Segregation Pattern.* Preluat de pe msdn.microsoft.com: https://msdn.microsoft.com/en-us/library/jj591573.aspx

Mitra, S. (2016, August 26). *Why Microservices?* . Preluat de pe dzone.com: https://dzone.com/articles/microservices-basics

What is Continuous Delivery? (fără an). Preluat de pe continuousdelivery.com: https://continuousdelivery.com/

Young, G. (2010, Februarie 16). *CQRS, Task Based UIs, Event Sourcing agh!* Preluat de pe codebetter.com: http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/