Cuprins

[Introducere 1](#_Toc486086370)

[1. Contribuții 3](#_Toc486086371)

[2. Aspecte teoretice 4](#_Toc486086372)

[2.1 Microservicii – concepte generale 4](#_Toc486086373)

[2.2. Domain-Driven Design (DDD) 10](#_Toc486086374)

[2.3. Command/Query Responsibility Segregation (CQRS) 12](#_Toc486086375)

[2.4. Event Sourcing 14](#_Toc486086376)

[2.5. Teorema CAP 14](#_Toc486086377)

[3. Aspecte practice 19](#_Toc486086378)

[3.1. Venture – introducere 19](#_Toc486086379)

[3.2. Utilizarea DDD pentru identificarea microserviciilor. 20](#_Toc486086380)

[3.3. Implementarea CQRS cu ajutorul microserviciilor 24](#_Toc486086381)

[3.4. Aplicarea pattern-ului Event Sourcing 25](#_Toc486086382)

[3.5. Consecințele teoremei CAP 26](#_Toc486086383)

[3.6. Privire de ansamblu 29](#_Toc486086384)

[3.7. Tehnologii folosite 30](#_Toc486086385)

[3.8. Detalii de implementare 37](#_Toc486086386)

# Introducere

Lucrarea de față își propune să prezinte principalele aspecte teoretice și practice ale arhitecturii bazate pe microservicii, in comparație cu arhitectura clasică, monolitică.

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

Arhitectura monolitica vine cu avantajele si 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 sa 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 in restul sistemului. De asemeni, elimina angajamentul pe termen lung fața de o anumita tehnologie – spre exemplu: servicii diferite pot fi scrise in limbaje de programare diferite – și permite un timp scurt de la începerea dezvoltării pana la lansarea in producție. (Badola, 2015)

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

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

Al treilea capitol, intitulat „Aspecte practice”, va exemplifica noțiunile prezentate in capitolul precedent prin aplicația „Venture”. Aceasta vine in ajutorul oricărei persoane ce dorește sa 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 si ultimul capitol al lucrării, intitulat „Posibile îmbunătățiri”, va veni cu idei referitoare la viitoarea dezvoltare a proiectului.

# Contribuții

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

# 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 in acest capitol sunt preluate in principal din cartea „Microservices in .NET Core”, scrisă de Christian Horsdal Gemmelgaard, un arhitect/developer expert in materie de .NET. Alte idei sunt preluate din diferite articole publicate pe internet, sau din concluzii proprii.

## Microservicii – concepte generale

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

Christian Horsdal Gemmelgaard remarca, in cartea sa, intitulată „Microservices in .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 in producție in 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 in producție in mod individual.

Când un microserviciu se schimba, el trebuie sa poată fi relansat in mod individual, fără a relansa orice alta parte a sistemului, iar celelalte servicii ale sistemului trebuie sa poată continua funcționarea pe durata lansării in mediul de producție a serviciului modificat si 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 in 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 si riscant. Acest fenomen ar trebui evitat, in favoarea lansărilor dese si 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 sa fie „backward-compatible”, astfel încât serviciile care comunică cu acesta să nu trebuiască modificate la rândul lor. In 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 sa fie încapsulat in unul sau mai multe procese separate.

Daca acest principiu nu ar fi respectat și am avea doua sau mai multe microservicii care rulează in cadrul aceluiași proces, atunci o eroare ce apare într-unul din ele se poate propaga si 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 in 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 sa 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 in ceea ce privește costurile de dezvoltare, performanță si 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 sa fie posibila, serviciul trebuie sa fie de mici dimensiuni. Un serviciu care creste prea mult, devine greu de întreținut si modificat – similar unei aplicații monolitice; un astfel de serviciu ar trebui împărțit la rândul lui in altele mai mici.

(Gammelgaard, 2017)

Un microserviciu este ușor de înlocuit.

Echipa care întreține un microserviciu trebuie sa 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 sa fie de mici dimensiuni.

Același autor subliniază, in 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 si 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 sa ne asigurăm că codul este întotdeauna într-o stare din care poate fi lansat cu ușurință, chiar si in cazul unde avem un număr mare de programatori ce fac schimbări in fiecare zi. Astfel este eliminată nevoia pentru fazele de integrare, testare si întărire, cât si 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 in cadrul sistemelor bazate pe această arhitectură prin orientarea către servicii care pot fi dezvoltate si modificate cu ușurința si rapiditate, pot fi testate utilizând teste automate, pot fi lansate in mod independent și pot fi operate in 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 in mod independent, rapid, si 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 in mod individual.

În consecință, problemele din mediul de producție pot fi descoperite si 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, in funcție de locul in care se produce fenomenul de „bottleneck”. In plus, microserviciile sunt orientate spre o forma asincronă de comunicare și spre toleranța la eșec in cazurile in 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 in cazul arhitecturii monolitice și testarea poate fi mai greoaie.

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

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

(Gammelgaard, 2017)

## 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 in domeniu pentru a vorbi despre acesta. Totuși, se observă că limbajul folosit de experți nu este consecvent in toate cazurile.

In 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 in contexte diferite.

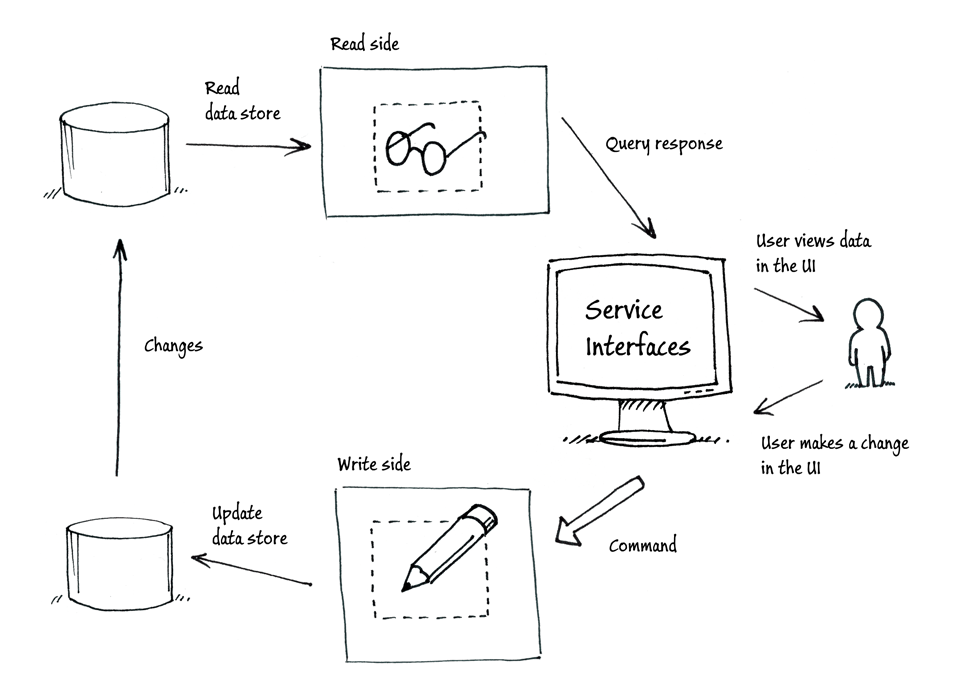
Un *bounded context*, în cadrul DDD, este o parte dintr-un domeniu mai mare, în care cuvintele au aceeași semnificație. *Bounded contexts* sunt legate, dar în același timp diferite de conceptul de capabilitate de business. Un bounded context definește o zona în care limbajul este consecvent. Capabilitățile de business, pe de alta parte, reprezintă ceea ce este necesar ca business-ul sa 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 in citatul dat despre limbajul folosit de experții in domeniu. Eric Evans, autorul cărții „Domain-Driven Design: Tackling Complexity in the Heart of Software”, numește acest limbaj „ubiquitous language”, același termen ce va fi folosit și in restul lucrării.

## Command/Query Responsibility Segregation (CQRS)

„CQRS este pur și simplu crearea a doua obiecte acolo unde înainte era doar unul. Separarea are loc in funcție de tipul metodelor, ce pot fi ori comenzi, ori interogări (având aceeași definiție care a fost folosită de Mayer in *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 in schimbare.

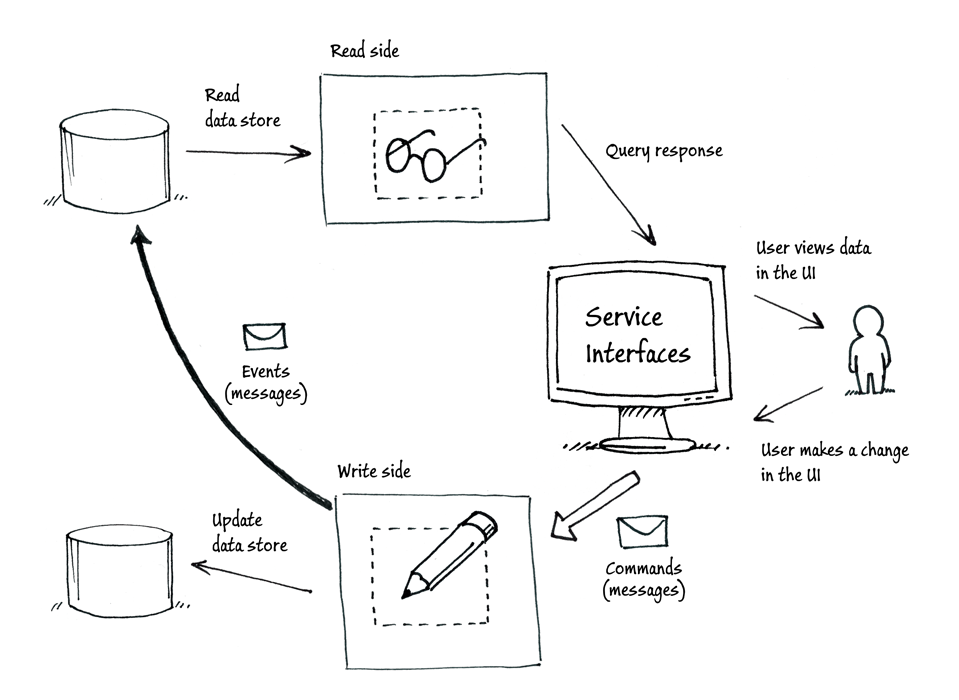


Figură 1 O posibila implementare a pattern-ului CQRS. (Microsoft)

Utilizarea acestui pattern ne permite scalarea și optimizarea individuală a componentelor, ceea ce este foarte util in 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).

CQRS permite si optimizarea modelelor pentru citire, respectiv scriere. Spre exemplu pe componenta responsabila cu gestionarea comenzilor, poate exista un model complex, ce este capabil sa 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.

In 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 in capitolul următor).



Figură 2 Events în pattern-ul CQRS (Microsoft)

(Microsoft)

## 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 in considerare și faptul ca pe partea de scriere, modelul nu mai trebuie sa suporte operații de citire, *event sourcing* poate fi o metodă foarte bună de a salva starea curentă a datelor.

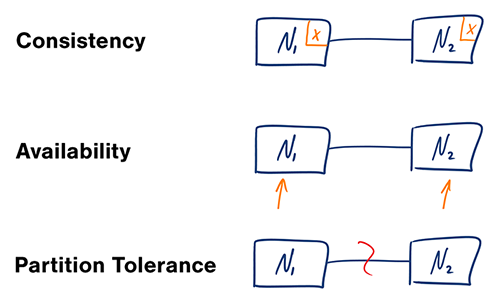
(Young, 2010)

## Teorema CAP

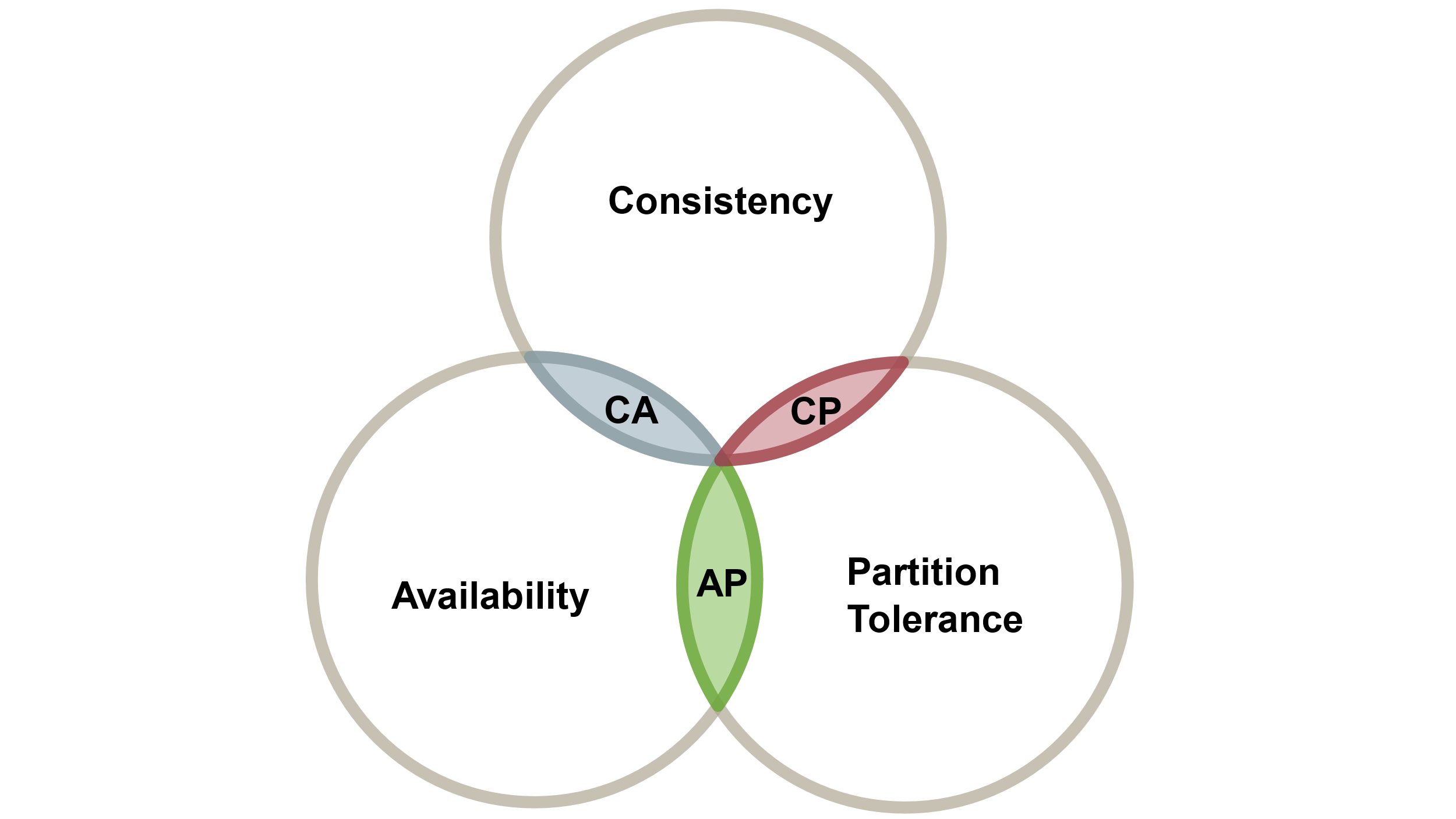
Teorema CAP (numită și „Teorema lui Brewer”) susține că, într-un sistem distribuit, cum este acela bazat pe microservicii, la un moment dat, putem avea doar doua din următoarele garanții:

* Consecvență (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 si după partiționarea rețelei.

Teoretic se observa cu ușurința ca aceasta teorema ne oferă trei opțiuni: putem sacrifica ori garanția de consecvență (cazul AP, după cum se vede in Fig. 2), ori garanția de disponibilitate (cazul CP), ori garanția de toleranța la partiționare (cazul AP). Totuși, după cum vom vedea mai departe, in practica nu există atât de multe opțiuni.



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



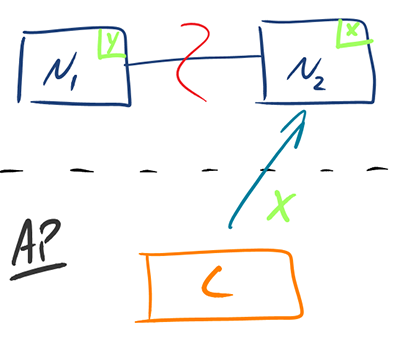
Figură 4 Combinațiile de garanții permise de teorema CAP (Erb)

O presupunere greșită, întâlnită des in contextul sistemelor distribuite, este aceea că rețeaua este de încredere. În actualitate, rețelele cad in mod frecvent și pe neașteptate. Trebuie, în consecință sa acceptăm ca aceste eșecuri vor avea loc și că nu avem niciun control asupra momentului in care vor avea loc.

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

(Greiner, 2014)

Cazul AP – În cazul in 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 in momentul in care partiționarea rețelei este reparată.

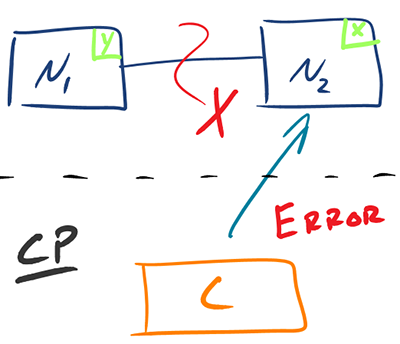


Figură 5 Cazul AP – Consecvența este sacrificată (Greiner, 2014)

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

(Greiner, 2014)

Cazul CP – O componentă a unui astfel de sistem, in condițiile in 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)

In ambele cazuri, pe durata partiției rețelei, sistemul nu este disponibil. Această abordare este potrivită in cazurile unde se cerințele sistemului impun operații de citire si 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 sa 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 sa pună bazele unui start-up, persoane ce vor sa pună in practica o idee inovativă, dar nu posedă cunoștințele tehnice pentru a o face, persoane ce doresc sa 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, si 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 in 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 si 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 in .NET Core”, Christian Horsdal Gammelgaard subliniază faptul că un microserviciu trebuie sa fie responsabil de o singură capabilitate a sistemului.

„Pentru a avea succes în implementarea microserviciilor, este important ca arhitectul să aleagă dimensiunile fiecărui microserviciu in 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 sa crească. Dacă microserviciile au dimensiunile potrivite, dar granițele dintre ele au fost trasate in mod greșit, atunci cuplajul iar tinde sa 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 in 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 in 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 in aplicarea metodei DDD, considerat însăși fundația practicii, este identificarea unui „Ubiquitous Language” (din engleză, „limbaj comun”). Acesta este menit sa servească ca un limbaj comun între experții in domeniu, și echipa de dezvoltare. Prin urmare, acesta trebuie sa 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 in contexte diferite (mai exact, in 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 in 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 in 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 sa îi corespundă cel puțin un microserviciu.

Utilizând conceptul de bounded context, putem împarți domeniul in 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 in sensul atingerii unui țel comun.
* **Project Owner** – Aferent unui **team,** reprezintă **user**-ul ce a createchipa, 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 in 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 sa avem un model mai specializat – observăm in acest sens faptul ca un user, in 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 in 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 in 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 in schimbare.

Utilizarea acestui pattern ne permite scalarea și optimizarea individuală a componentelor, ceea ce este foarte util in 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 in 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, s-a 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 sa 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, in 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 si 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 sa 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: consecvență (**C**onsistency), disponibilitate (**A**vailability) și toleranță la partiționare (**P**artition Tolerance).

După cum a fost demonstrat in 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 consecvența datelor oferite de sistem.

Considerăm cazul următor, în contextul aplicației Venture: Datorită unor factori ce se află in afara controlului nostru, comunicarea între microserviciile Venture.ProjectWrite și Venture.ProjectRead nu mai poate fi efectuată. În această durată, un utilizator scrie un comentariu in chatul public al unui proiect. La scurt timp, un alt utilizator deschide pagina acestui proiect. În acest moment, dacă am decide să garantăm consecvența datelor in 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 in descrierea acestuia. Chiar daca aceste acțiuni nu au avut loc in realitate, in contextul exemplului dat, serviciul nu poate garanta consecvența datelor, deci decide sa 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 in majoritatea domeniilor de business (unde nu se impun operații atomice de citire și scriere), sacrificarea disponibilității in favoarea consecvenței 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 sa 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 consecvenței. 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 sa 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 consecvenței.

## 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 si 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 sa 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 in 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). Această opțiune conferă un număr de avantaje ce vor fi discutate în următorul subcapitol.

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, in funcție de nevoile fiecăruia, dar pe primul nivel se află întotdeauna stratul „Service” – responsabil cu oferirea serviciului propriu-zis către restul sistemului.

## 3.7. Tehnologii folosite

În acest subcapitol se va intra mai adânc in 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

În primul rând a fost nevoie de un framework peste care sa 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 in prea mult detaliu asupra ei.

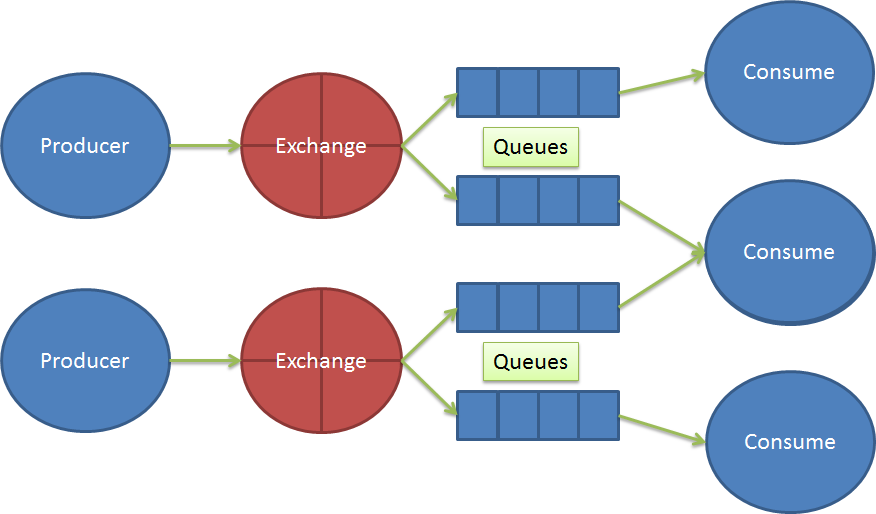
Stratul „Service” al componentei Venture.Gateway este de fapt o aplicație ASP.NET ce oferă un WEB API accesibil oricărui client. Restul microserviciilor (cele „interne”) folosesc pe nivelul de serviciu un „Console Application”.

RabbitMQ și RawRabbit

S-a menționat in 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.



Figură 7 Privire de ansamblu asupra modului de funcționare al RabbitMQ (Freeman, 2013)

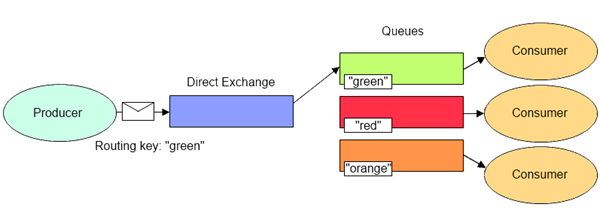
RabbitMQ funcționează implementând pattern-ul producător/consumator. Mesajele de la un producător sunt rutate spre consumatorul sau consumatorii corecți utilizând o „topologie” constituită din exchanges (din engleză, „schimburi”), routing keys (din engleză, „chei de rutare”) și queues (din engleză, „cozi”).

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 in exchange, RabbitMQ trebuie sa ruteze mesajul spre queue-urile potrivite. 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.



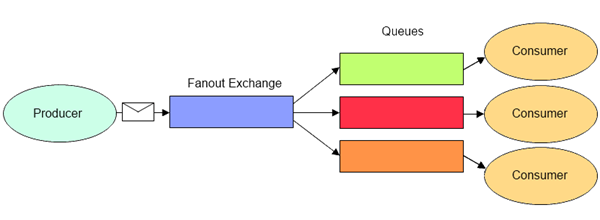
Figură 8 Direct Exchange (Greer, 2012)

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 în sensul in 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 9 Topic Exchange (Greer, 2012)



Figură 10 Fanout Exchange (Greer, 2012)

Exchange-urile de tip „headers” funcționează într-un mod similar cu cele de tip topic. Diferența constă in faptul ca rutarea nu se face in baza cheii de rutare, ci in baza unor headere definite atât in 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 sa coincidă cu cele din headerele cozii. Numărul de perechi cheie-valoare ce trebuie sa coincidă este specificat prin valoare unei chei speciale din cadrul cozii.



Figură 11 Headers Exchange

Aplicația „Venture” folosește RabbitMQ pentru a obține un număr de beneficii.

În primul rând, folosind service bus-ul RabbitMQ 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 si 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 sa 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 multă ușurință. RabbitMQ 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.

SQL Server Express și Entity Framework

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

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 in-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 in-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.

## 3.8. Detalii de implementare

Lorem ipsum