

UNIVERSITATEA „BABEȘ-BOLYAI”
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE DIPLOMĂ

**STUDIUL EFICIENȚEI FOLOSIRII UNEI
ARHITECTURI BAZATE PE MICROSERVICII
ÎNTR-UN SISTEM DE PLATĂ A UTILITĂȚILOR**

Coordonatori științifici:

lect. dr. **Mircea Ioan Gabriel**

Absolvent:

Petruțiu Paul-Gabriel

Cluj-Napoca

2019

Cuprins

Listă de figuri	3
1 Introducere	4
2 Concepte de bază	6
2.1 Arhitectura unei aplicații soft	6
2.2 Șabloane de proiectare	6
2.3 Serviciu Software	7
3 Aplicații Monolit	8
3.1 Ce este o aplicație monolit?	8
3.2 Avantaje și dezavantaje	8
4 Arhitectura bazată pe servicii	10
4.1 Fundamentele arhitecturii bazate pe servicii	10
4.2 Principiile arhitecturii bazate pe servicii	11
4.2.1 Autonomia	11
4.2.2 Cuplajul slab	12
4.2.3 Abstractizarea	12
4.2.4 Contractul Formal	13
5 Arhitectura bazată pe microservicii	14
5.1 Ce sunt microserviciile	14
5.2 Comunicarea între microservicii	15
5.2.1 Diferite tipuri de comunicare	15
5.2.2 Utilizarea unui broker de mesaje	15
5.3 Avantaje și dezavantaje ale microserviciilor	16
5.3.1 Avantaje	16
5.3.2 Dezavantaje	17

5.3.3	Cand putem să utilizăm microserviciile?	17
6	Studiu de Caz	19
6.1	Introducere	19
6.2	Tranziția către microservicii în compania Netflix	19
6.2.1	Arhitectura aplicației de azi	20
6.2.2	Motive	21
6.2.3	Avantaje și dezavantaje, costuri și sacrificii	22
6.2.4	Descrierea tranziției către microservicii	24
7	Aplicație Practică	27
7.1	Idee	27
7.2	Arhitectura aplicației	28
7.3	Persistența datelor	31
7.4	Interacțiunea cu utilizatorul	32
7.4.1	Plata unei facturi	34
7.4.2	Generarea unui raport	35
7.5	Specificații tehnice	36
7.6	Analiza arhitecturii	37
8	Concluzii	39
9	Bibliografie	41

Listă de figuri

3.1	Arhitectură Aplicație Monolit(imagine : http://bits.citrusbyte.com)	9
5.1	Arhitectură monolit vs arhitectură bazată pe microservicii [15]	16
5.2	Complexitatea si productivitatea in arhitecturi diferite [8]	18
6.1	Arhitectura Aplicației Netflix [12]	20
6.2	Arhitectura Netflix - „Death Star” [10]	21
6.3	Chaos Monkey [12]	22
6.4	Procesul de dezvoltare intr-o aplicație monolit [12]	23
6.5	Procesul de dezvoltare intr-o aplicație bazată pe microservicii [12]	23
6.6	Ciclul de viață al unui microserviciu [12]	25
6.7	Migrarea catre cloud a aplicațiilor [10]	26
7.1	Diagrama de arhitectură	29
7.2	Diagrama de clase a unui microserviciu atribuit unui furnizor	30
7.3	Bazele de date pentru fiecare microserviciu	31
7.4	Componenta bazelor de date	31
7.5	Obiect în cod C# și obiect în baza de date	32
7.6	Diagrama cazurilor de utilizare	33
7.7	Diagrama de secvență pentru plata unei facturi	35
7.8	Diagrama de secvență pentru generarea unui raport	36
7.9	Analiza asupra codului, generată de Visual Studio 2019	38
7.10	Formula de calcul pentru „Maintainability Index”	38

Capitolul 1

Introducere

Deoarece mediul IT este într-o continua dezvoltare, mereu apar noi concepte legate despre arhitecturi și șabloane de proiectare care ajută programatorii să dezvolte aplicațiile mai repede și mai eficient. În această lucrare se pune pe accentul pe stilul arhitectural care ar trebui adoptat pentru aplicațiile din ziua de azi, lucrurile care pot decide stilul arhitectural cât și diferențele dintre două arhitecturi diferite.

În lucrare se discută despre două tipuri arhitecturale diferite, una provenind din modul normal de gândii, aceasta fiind arhitectura monolit, iar una provenind din nevoia unei soluții care să rezolve unele probleme, precum creșterea exponențială a costului scalabilității sau scaderea productivității în aplicații foarte mari, aceasta arhitectură fiind arhitectura bazată pe servicii. În această lucrare se studiază și o ramură a arhitecturii orientată pe servicii, arhitectura pe microservicii, care este considerată o variație mai rafinată a arhitecturii bazate pe servicii.

Scopul final al lucrării este acela de a evidenția avantajele pe care o arhitectură bazată pe microservicii le poate aduce într-o aplicație care vrea să intermedieze diferite operații între utilizatori și diferiți furnizori care au obiective comune. Acest lucru se va realiza cu ajutorul unei aplicație prin care utilizatorii își pot plăti facturile de la diferiți furnizori.

Lucrearea este împărțită în opt capitole, care la randul lor sunt împărțite din secțiuni și subsecțiuni. În capitolul al doilea se prezintă concepte generale despre ce înseamnă o arhitectură, un șablon de proiectare sau un serviciu software. În capitolul trei se prezintă arhitectura monolit cu avantajele dezavantajele pe care le aduce. În capitolul patru se prezintă concepte generale și principii legate de o arhitectură bazată pe servicii, urmând ca în capitolul cinci să se vorbească despre arhitectura bazată pe microservicii, ce este un microserviciu, tipurile de comunicare dintre microservicii, cât și despre avantajele și dezavantajele ei. În capitolul se studiază tranziția aplicației

Netflix de la o arhitectură de tip monolit la una bazată pe microservicii. Se prezintă motivele care au dus la luarea acestei decizii, structura finală a proiectului cât și problemele întâmpinate pe parcurs.

În capitolul șapte se va prezenta o platformă menită să intermedieze plățile facturilor dintre consumatori și diferiți furnizori. Se va prezenta structura aplicației și deciziile arhitecturale luate pentru a se respecta principiile unei arhitecturi bazate pe microservicii. Pentru a se demonstra că o arhitectură bazată pe microservicii este o decizie corectă, se prezintă o analiză a codului, calculată cu ajutorul Visual Studio.

Ultimul capitol al lucrării conține concluziile cu privire la avantajele unei arhitecturi bazate pe microservicii și argumentează motivele pentru care o astfel de arhitectură reprezintă o soluție modernă de a structura o aplicație software.

Capitolul 2

Concepte de bază

2.1 Arhitectura unei aplicații soft

Deoarece tot mai multe companii din diferite domenii au nevoie de o aplicație software personală pentru a-și îmbunătăți calitatea produselor, pentru prezentarea produselor sau pentru ușurarea unor activități din cadrul companiei, cererea de a crea cod este tot mai mare. Pentru ca aplicațiile să fie ușor de dezvoltat, pentru ca o îmbunătățire să poată fi implementată fără prea mult efort, este important ca aplicația să aibă la bază o structură bine definită. Acest lucru face ca o aplicație să fie ușor de întreținut în decursul timpului.

Având în vedere standardul IEEE Std 1471, arhitectura unei aplicații soft este definită ca „Organizarea fundamentală a unui sistem încorporat în componentele sale, relațiile dintre ele și mediul, precum și principiile care conduc proiectarea și evoluția sa.”([11])

Așadar, putem considera un sistem ca fiind o aplicație sau un set de aplicații care împreună rezolvă diferite probleme.

2.2 Șabloane de proiectare

Un șablon de proiectare este reprezentat ca fiind o soluție cunoscută pentru o problemă recurentă.

„Șabloanele de proiectare pot fi văzute ca un mijloc de a reuși reutilizarea pe scară largă prin captarea unei practici de design de dezvoltare a software-ului de succes într-un anumit context” ([1]). Deci, un șablon de proiectare reprezintă doar în mod abstract o soluție pentru o problemă.

Din acest motiv, șabloanele de proiectare pot fi aplicate în oricare limbaj de programare, în funcție de contextul problemei.

Motivul pentru care șabloanele de proiectare sunt relevante indiferent de limbajul de programare folosit, este acela că șabloanele de proiectare sunt doar concepte despre cum ar trebui să fie implementat codul și nu cod propriu zis.

2.3 Serviciu Software

Un serviciu este o componentă a unei aplicații soft care furnizează una sau mai multe funcționalități altor componente din sistem. Aceste componente pot să fie aplicații web, mobile sau chiar alte servicii.

Spre exemplu, putem presupune ca avem un site în care utilizatorii pot să achite facturile de curent, gaz, etc.. În momentul în care utilizatorul efectuează o plată, browser-ul va apela un serviciu care va procesa, în spate, această tranzacție (verificarea validității sumei introduse, dacă suma este disponibilă, etc.)

Sistemele ce folosesc mai multe servicii, similar cu exemplul expus mai sus, sunt considerate ca fiind sisteme care au la bază o arhitectură orientată pe servicii.

Capitolul 3

Aplicații Monolit

3.1 Ce este o aplicație monolit?

O aplicație monolit este o aplicație a cărei cod este scris în cadrul unei singure unitați structurale. Componentele din care este alcătuită aplicația sunt gândite în așa fel încât să funcționeze împreună și să se folosească de același spațiu de memorie și de aceleași resurse.

Aplicațiile monolit sunt printre cele mai răspândite din lume. Acest fapt se datorează felului în care oamenii abordează problemele. O soluție de tip monolit este prima soluție pe care un programator o va avea, deoarece este un mod natural de a gândi. În plus, pentru multe din aplicații, o soluție monolit va fi soluția perfectă, având în vedere că multe companii mici spre medii doresc să aibă o aplicație care să automatizeze anumite procese, procese care nu au o complexitate extrem de ridicată, iar numărul utilizatorilor nu este foarte mare.

Spre exemplu, să spunem că o firmă are un sediu destul de mare care are 5 săli de conferință, având în vedere numărul mare de întâlniri din interiorul firmei, aceștia doresc o aplicație în care să poată rezerva o sala de conferință pentru o anumită perioadă într-o anumită zi. O astfel de aplicație se poate realiza ușor și rapid, sub forma unei aplicații web de tip server client(3.1).

3.2 Avantaje și dezavantaje

O aplicație monolit are mai multe avantaje în contextul unui business mic:

- Faza de dezvoltare durează mai puțin timp, IDE-urile moderne reușind să genereze mult cod automat, ceea ce implică scăderea costurilor de producție.

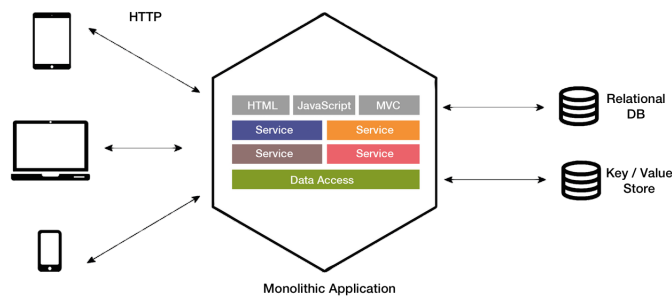


Figura 3.1: Arhitectură Aplicație Monolit(imagine : <http://bits.citrusbyte.com>)

- Aplicația este ușor de testat, automatizând procesul de testare a codului folosind o combinație de unit teste și integration test.
- Procesul de lansare în producție a unei aplicații nu este complicat, deoarece este nevoie de rularea unei singure aplicații per server.

Pe de altă parte, în momentul în care aplicația va crește ca și volum al codului, un număr mai mare de programatori vor trebui implicați în procesul de dezvoltare al aplicației. Acest lucru nu este un lucru rău, doar că aceștia vor întâmpina anumite impedimente:

- Cuplajul dintre componentele aplicației crește, iar acest fapt va îngreuna procesul de dezvoltare a unor noi funcționalități, acest lucru va afecta timpul și costul necesar dezvoltării noii funcționalități.
- Alegerea tehnologiilor folosite pentru dezvoltarea aplicației este permanentă.
- Integrarea/Intrarea unui programator pe proiect va fi mai dificilă. Volumul aplicației fiind mare, va fi necesar un timp mai mare pentru înțelegerea tuturor funcționalităților și a deciziilor luate pe proiect în trecut.

Acestea sunt doar câteva dintre avantajele și dezavantajele unei aplicații monolit, dar sunt suficiente încât putem sublinia o idee generală. Aplicațiile bazate pe o arhitectura monolit sunt mai ușor de implementat în primă fază cu un cost mai redus, dar în momentul în care aplicația ajunge la un nivel mai avansat, este nevoie de o reconstruire parțială sau totală a aplicației sau cel puțin este nevoie de o refactorizare care să diminueze dezavantajele create de stilul arhitectural folosit. Majoritatea aplicațiilor încep ca și aplicații tip monolit, iar mai târziu se pot transforma în aplicații cu arhitecturi bazate pe microservicii de exemplu.([17])

Capitolul 4

Arhitectura bazată pe servicii

4.1 Fundamentele arhitecturii bazate pe servicii

Unul din factorii care au dus la modificarea conceptelor despre cum trebuie să fie structurată o aplicație, a fost evoluția tehnologică rapidă și numărul tot mai ridicat de persoane care aveau acces la internet.

Arhitectura bazată pe servicii (în engleză „Service Oriented Architecture”, prescurtat „SOA”), este un tip de arhitectură care s-a născut datorită nevoii tot mai mari de dezvoltare rapidă a cât mai multor funcționalități într-un timp cât mai scurt, și în paralel pe cât posibil.

Unul din motivele principale pentru apariția acestui stil arhitectural s-a datorat dorinței producătorilor ca o aplicație să poată să fie folosită de pe diferite platforme. Dacă pentru fiecare platformă diferită era nevoie ca toată aplicația să fie rescrisă, costul ar fi fost prea mare. Această soluție nu era plauzibilă. Datorită acestui fapt s-a decis că logica aplicației ar trebui să fie decuplată de partea din față a aplicației. În felul acesta, fiind necesară doar rescrierea unei părți mult mai mici din aplicație pentru ca aceasta să poată funcționa pe diferite tipuri de platforme. Odată cu această decizie, un nou stil arhitectural a început să apară.[6]

Pentru început, s-a încercat enunțarea a opt principii, care ar trebui respectate într-o arhitectură bazată pe servicii.[3]

Acestea sunt:

- Serviciile sunt autonome.
- Serviciile sunt slab cuplate.

- Serviciile trebuie să abstractizeze logica pe care o folosesc.
- Serviciile împart un contract formal.
- Serviciile sunt compozabile.
- Serviciile sunt refolosibile.
- Serviciile nu au stare.
- Serviciile se pot descoperii.

Din aceste opt principii, primele patru sunt cele mai importate, acestea reprezentând fundația arhitecturii bazate pe servicii. Toate cele opt principii se suportă între ele, dar primele patru principii le produc și pe restul.

4.2 Principiile arhitecturii bazate pe servicii

4.2.1 Autonomia

Orietarea către servicii cere o atitudine serioasă atunci când vine vorba de împărțirea lor în blocuri logice de sine stătătoare. Pentru ca serviciile să fie fiabile și previzibile, trebuie să se exercite un grad mai ridicat de control asupra resurselor pe care le folosesc.

Odată cu creșterea nivelului de control al unui serviciu asupra propriului context de execuție, se reduc dependențele din serviciu, ceea ce ne duce spre un alt principiu, cel al cuplajului slab. Chiar dacă exclusivitatea asupra logicii pe care un serviciu o încapsulează nu poate să fie deplină, principalul obiectiv este atribuirea unui nivel rezonabil de control asupra oricărei logici pe care o prezintă într-un moment al execuției.[5]

Având în vedere că pot exista deferite metode de a măsura autonomia, este bine să facem distincție între ele. Spre exemplu, un serviciu poate avea mai multe nivele de autonomie:

- Autonomie la nivelul serviciului. În acest tip de autonomie granițele dintre servicii sunt bine definite, dar resursele folosite de server pot să fie distribuite cu alte servicii. Spre exemplu, putem avea un serviciu care încapsulează un mediu de lucru mai vechi, acesta deja fiind folosit de un alt serviciu vechi.
- Autonomie pură. Când vorbim de autonomie pură, ne referim la faptul că întreaga logică de care se folosește un serviciu, îi servește exclusiv lui.

Cu siguranță, se dorește ca un număr cât mai mare să fie de servicii pur autonome, deoarece acestea ar rezolva ușor problemele de concurență și ar împinge serviciile importiva problemei „unui singur punct de eșec” (în engleză „single point of failure”). Oricum, pentru a se atinge această performanță, procesul de dezvoltare este mai lung, deoarece logica din servicii trebuie să fie rescrisă, iar partea de lansare în execuție a serviciului ar avea nevoie de atenție sporită. Așadar efortul și costurile ar crește, iar uneori aceste sacrificii nu sunt posibile.

4.2.2 Cuplajul slab

Nimeni nu poate să prezică în direcție va evolua domeniul IT. Nu se poate ști evoluția pentru automatizarea soluțiilor, integrarea sau schimbarea lor, acestea fiind influențate de cauze din afara mediului IT. Având în vedere faptul că în cazul unei modificări neprevăzută programatorii trebuie să fie gata să răspundă într-o manieră eficientă, un lucru necesar este lucrul cu forme abstracte ale serviciilor și mesajelor. Acest fapt, susține agilitatea de a compune o soluție folosind serviciile disponibile. [4]

Putem spune că cuplajul între două unități logice și structurale poate fi văzut ca o măsurătoare a dependențelor dintre ele. Ceea ce înseamnă că un număr mare de dependențe poate fi definit ca și un cuplaj mare. În cazul în care nu există dependențe între două servicii, putem spune că ele sunt decuplate. Prin implementarea consistenței a cuplajului slab, unitățile logice dezvoltate ulterior dobândesc independență. Din acest fapt, în timp, se acumulează o multitudine de servicii care sunt blocuri logice de sine statatoare, care pot fi folosite în noi compoziții, și care pot fi întreținute ușor, fără a influența alte servicii.[4]

4.2.3 Abstractizarea

Abstractizarea este un concept cunoscut în programare, fiind unul din principiile fundamentale ale paradigmei programării orientate pe obiecte. Cam în aceeași direcție se îndreaptă și abstractizarea logicii din servicii. Acest concept sugerează că un serviciu ar trebui scris ca o cutie neagră, adică ascunzând detaliile de un potențial consumator. Acest lucru se realizează folosind contracte pentru fiecare serviciu, acesta limitând accesul către serviciu. Cu ajutorul unui contract, se obține foarte ușor un grad mare de separare între ce este privat și ce este public pentru consumator. Ca rezultat, acest concept susține conceptul dezbătut anterior, cel al cuplajului slab.[4]

Dacă ar fi să discutăm despre cantitatea de logică pe care un serviciu o poate expune, putem spune că nu există nici o limitare din acest punct de vedere. Un serviciu poate fi creat cu scopul de a servi îndeplinirii unui singur feature sau poate fi un punct de intrare în întreaga aplicație.

Așadar se poate observa că întreaga atenție se poate îndrepta se modul în care un contract este conceput. Cu cât mai multă informație este expusă prin contract, cu atât mai puțină informație poate fi abstractizată. Cu cât un contract este mai generic, cu atât mai mult crește capacitatea lui de a fi reutilizat.

4.2.4 Contractul Formal

În mod normal, serviciile trebuie definite în mod formal folosind unul sau mai multe documente descriptive. Spre exemplu, pentru un serviciu web se folosesc documente de tip WSDL și o schema XSD. Documentele care ajută la descrierea unui serviciu pot fi considerate colectiv ca fiind un contract de servicii. Conform arhitecturii bazate pe servicii, un contract al unui serviciu ar trebui să definească următoarele lucruri:

- Punctele de acces în serviciu (în engleză „endpoint”).
- Fiecare operație pe care o poate efectua serviciul.
- Pentru fiecare operație, formatul datelor de intrare cât și al celor de ieșire
- Regulile și caracteristicile serviciului.

Definirea contractelor necesită o atenție sporită pentru că acestea sunt distribuite printre servicii. După momentul în care sunt lansate în execuție, definiția contractelor s-ar putea să devină o dependență pentru o sursă externă, ceea ce înseamnă că trebuie avut grijă ca definițiile din contract să nu sufere schimbări după lansarea principală. Prin urmare, contractele formale reprezintă un principiu de bază într-o arhitectura bazată pe servicii. În plus acest concept susține alte principii despre care am discutat, acestea fiind: abstractizarea și cuplajul slab, dar și alte principii care nu au fost abordate în detaliu precum: compozabilitatea și descoperirea serviciilor. [4]

Capitolul 5

Arhitectura bazată pe microservicii

5.1 Ce sunt microserviciile

Microserviciile sunt un concept apărut nu cu mult timp în urmă, iar acesta este unul dintre motivele pentru care acestea nu au o definiție exactă. Un alt motiv ar putea fi acela că microserviciile sunt o derivare ușoară a serviciilor. Având în vedere încercările de a crea o definiție pentru microservicii, putem găsi câteva caracteristici comune. Una dintre aceste definiții sugerează că arhitectura bazată pe microservicii este, de fapt o arhitectură bazată pe servicii în care toate principiile au fost aplicate corect. [9]

Dacă ar fi să extragem aceste caracteristici speciale pe care trebuie să le aibă un microserviciu, putem porni chiar de la cuvântul „microserviciu” care ar putea fi împărțit în „micro” și „serviciu”. Deci microserviciile sunt de fapt servicii de dimensiune mică, autonome și fără stare, acestea lucrând împreună pentru a rezolva o problemă.

Dacă tot ne bazăm pe „definițiile” din mediul online, putem găsi și mici diferențe între arhitectura bazată pe servicii și cea bazată pe microservicii. Spre exemplu comunicarea în servicii se face în general cu ajutorul unui service bus, iar în microservicii se încearcă folosirea unor sisteme de mesaje mult mai simple, serviciile sunt contruite printr-o abordare „distribuie cât mai mult” iar microserviciile printr-o abordare „distribuie cât mai puțin”. Serviciile se orientează mult mai mult pe reutilizare, în schimb microserviciile sunt orientate în primul rând pe un cuplaj cât mai mic, sau inexistent, acest lucru susținând principiul cuplajului slab.

5.2 Comunicarea între microservicii

5.2.1 Diferite tipuri de comunicare

Dacă vrem să discutăm despre comunicarea între microservicii, comunicarea ar putea să fie sincronă sau asincronă. În cazul microserviciilor se preferă comunicarea asincronă. Chiar dacă comunicarea sincronă este mai ușor de implementat, spre exemplu, se face un call către un microserviciu și se așteaptă până când acesta returnează un mesaj sau până eșuează. Dar acest lucru înseamnă că cele două servicii au un grad de cuplare mai mare, acest lucru fiind un dezavantaj al unui principiu de bază, cel al cuplajului slab. Dacă vorbim despre comunicare asincronă, ne referim la faptul că un call este creat, dar restul execuției nu este blocată până se primește un răspuns, acest lucru fiind favorabil atunci când logica din spate durează mai mult timp. Spre exemplu, dacă dacă se efectuează un call care ar trebui să aducă o listă mai mare de obiecte pentru a fi afișate. Nu se dorește ca interfața pe care utilizatorul o vede să fie blocată, așadar comunicarea asincronă va permite asta.

Comunicarea între microservicii se poate realiza sub forma de call-uri, așa cum deja am precizat mai sus, dar nu este singura variantă. O altă modalitate de comunicare între microservicii este comunicarea bazată pe evenimente, care funcționează invers într-o oarecare măsură. Acest lucru înseamnă că în momentul în care se produce o acțiune într-un microserviciu, aceasta produce un eveniment prin care anunță alte microservicii sau alte unități ale aplicației că o modificare a fost efectuată, iar acestea, având o logică implementată, vor putea gestiona situația.

5.2.2 Utilizarea unui broker de mesaje

Un broker de mesaje reprezintă de fapt o componentă exterioară a aplicației, care are scopul de stoca și persista mesajele sau evenimentele până când acestea sunt consumate de serviciul necesar. Un broker de mesaje necesar atunci când vorbim de o comunicare asincronă, mai ales când este vorba de comunicare cu ajutorul evenimentelor.

Un exemplu care determină utilitatea unui broker de mesaje, ar putea fi următorul. Să presupunem că avem un magazin online, și avem un microserviciu care se ocupă de plasarea unei comenzi, iar un microserviciu se ocupă cu logica pentru transportul pachetului. Dacă am presupune că microserviciul pentru transport nu funcționează, printr-o metodă sincronă de comunicare, rezultatul ar fi aruncarea unei excepții care menționează că nu se poate plasa o comandă. Dacă se folosește un broker de mesaje, comanda va fi procesată cu succes iar în momentul în care micro-

serviciul care se ocupă de transportul produsului va functiona, procesul de livrare al produsului va începe.

5.3 Avantaje și dezavantaje ale microserviciilor

Ca să vorbim de avantaje și dezavantaje, ar trebui să realizăm o comparație, iar pentru aceasta comparație ne vom folosi de o arhitectură monolit. Așadar în figura de mai jos (5.1) putem vedea diferențele de structură.

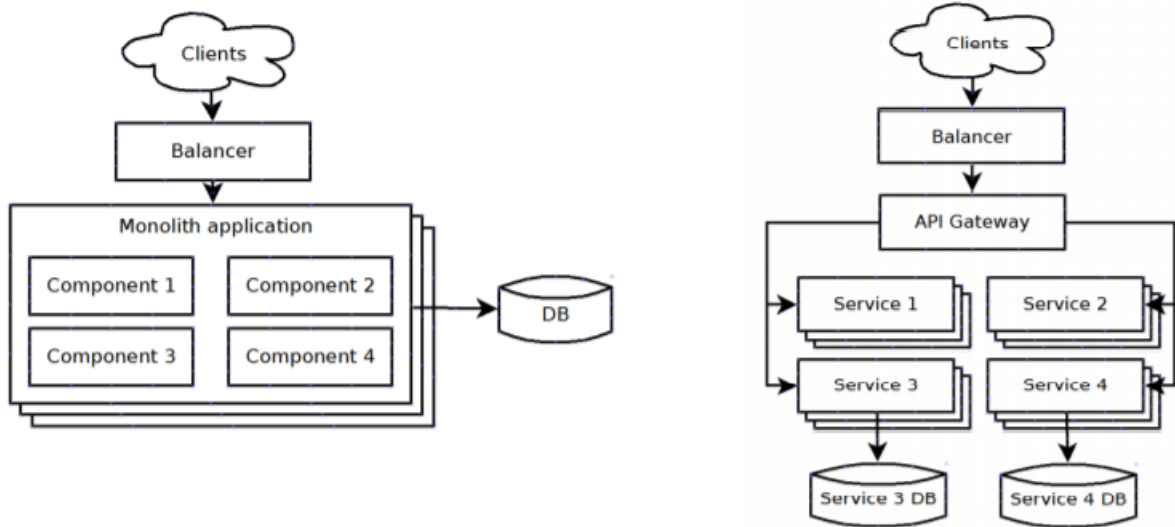


Figura 5.1: Arhitectură monolit vs arhitectură bazată pe microservicii [15]

5.3.1 Avantaje

Un avantaj al microserviciilor este oferit chiar de structura lor, după cum se observă și în figura 5.1, microserviciile sunt independente unele de celelalte, acest lucru făcând mult mai ușoară dezvoltarea și îmbunătățirea lor. Mult mai rapidă înțelegerea codului iar prin urmare, în opoziție cu o aplicație monolit, integrarea unui om nou pe proiect este mai ușoară.

Microserviciile pot să fie văzute ca niște bucăți de lego care împreună crează o întreagă aplicație. Din acest motiv microserviciile sunt ușor scalabile. Iar când ne referim la scalabilitatea microserviciilor, ne referim la o scalabilitate pe orizontală. Având în vedere dimensiunea unui

microserviciu în comparație cu dimensiunea întregii aplicații, e foarte evident că un microserviciu poate gestiona mult mai ușor și mai flexibil resursele unui server. Acest lucru înseamnă că resursele unui server mai puțin folosit, pot fi redistribuite pentru un server mai ocupat, pentru a se evita blocajele [16]. Așadar, din cauza flexibilității și a capacității de scalare pe orizontală, un sistem bazat pe microservicii are un cost mai mic decât un sistem monolit, deoarece nu toate microserviciile trebuie scalate în aceeași măsură, iar o aplicație monolit nu îți poate oferi opțiunea de a alege. [13]

5.3.2 Dezavantaje

Chiar dacă structura acestei arhitecturi aduce avantaje, aceasta aduce și dezavantaje. De exemplu, datorită complexității pe care o are o arhitectură bazată pe microservicii, o aplicație mică sau medie, ar avea doar de pierdut, timpul de dezvoltare crește și nevoia de resurse este mai mare.

Trecerea la microservicii, necesită un cost ridicat, mai ales la început. Acest lucru va fi evidențiat în studiul de caz despre trecerea la o arhitectură pe bază de microservicii în compania Netflix. Tot odată, nivelul cunoștințelor pe care un programator lucrează pe o asemenea tehnologie trebuie să fie mai vastă, iar acesta ar trebui să aibă mai multă experiență ca să poată lua anumite decizii, iar acest lucru va duce la un timp crescut de dezvoltare a aplicației.

5.3.3 Când putem să utilizăm microserviciile?

Răspunsul la întrebarea „Când putem să utilizăm o arhitectură bazată pe microservicii?” este „Oricând”. Orice aplicație poate să fie remodelată pe o arhitectură bazată pe microservicii, dar nu neapărat acest lucru ar fi un lucru bun. De exemplu, aplicațiile care nu au un nivel de maturitate mare, aplicațiile mici sau medii în care scalabilitatea ca aplicație monolit nu aduce costuri prea mari, nu au nevoie să fie mutate pe o arhitectură bazată pe microservicii, acest lucru fiind doar o risipă de resurse, după cum se vede și în figura 5.2

Un alt exemplu în care microserviciile nu ar fi o soluție bună ar fi aplicațiile care procesează fluxuri de date în timp real, date care vin de la diferiți senzori. Într-o situație de genul acesta, orice timp contează, iar comunicarea dintre microservicii, chiar dacă este rapidă, poate influența rezultatul, așadar într-o astfel de situație, o soluție monolit este favorabilă.

În concluzie, am putea spune că stilul arhitectural bazat pe microservicii poate fi aplicat în majoritatea aplicațiilor, dar este indicat să fie folosit doar atunci când aplicația este suficient de mare încât este greu de scalat sau scalarea devine prea scumpă sau în momentul în care sistemul devine atât de complex încât productivitatea este extrem de mică. În capitoul următor va fi

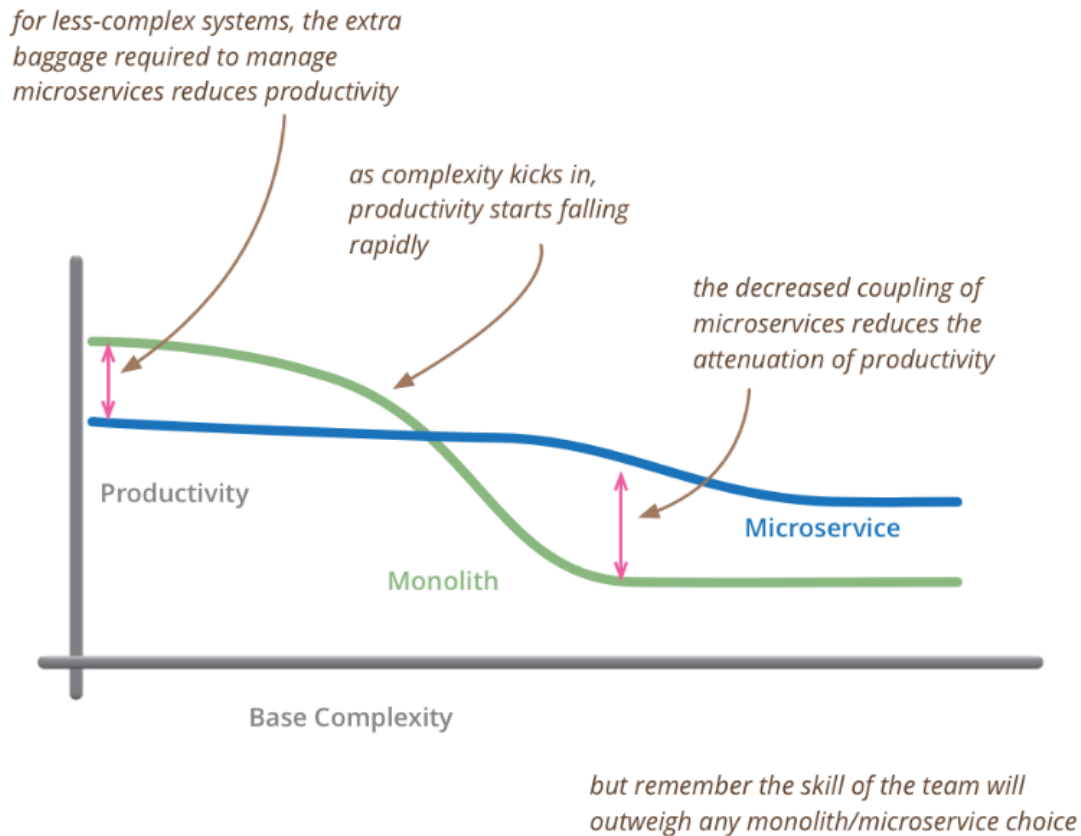


Figura 5.2: Complexitatea si productivitatea in arhitecturi diferite [8]

prezentată în detaliu o trecere a unei aplicații de la o arhitectură monolitică la una bazată pe microservicii, împreună cu avantajele, costurile si compromisurile care au fost facute, dar si o prezentare a rezultatului final.

Capitolul 6

Studiu de Caz

6.1 Introducere

Netflix este o companie care a început prin a vinde sau închiria filme pe DVD-uri. Ulterior furnizând acces online la filme și seriale. Netflix fiind un gigant în industria televiziunii online. Aplicația netflix este o aplicație la scară mondială, care în momentul în care firma a hotărât schimbarea arhitecturii avea un trafic de 8 milioane de utilizatori, ajungând la finalul anului 2018 la 139 de milioane de utilizatori.

După cum am discutat până acum, o arhitectură bazată pe microservicii nu are chiar o definiție propriu zisă, dar după cum susține Martin Fowler, microserviciile sunt implementarea corectă a arhitecturii bazate pe servicii. [9]

În acest capitol vom cuprinde trecerea de la o arhitectură monolit la o arhitectură bazată pe servicii, motivele pentru care s-a făcut această tranziție, pașii prin care s-a făcut această trecere, avantajele cât și dezavantajele acestei treceri, cât și despre rezultatul final.

6.2 Tranziția către microservicii în compania Netflix

În acest studiu de caz o să ne bazăm pe informațiile oferite de două dintre personajele importante care au luat parte la tranziția către microservicii:

- Ruslan Meshenberg
- Adrian Cockcroft
- Josh Evans

6.2.1 Arhitectura aplicației de azi

În următoarea diagrama, este reprezentată, arhitectura bazată pe microservicii a aplicației Netflix.

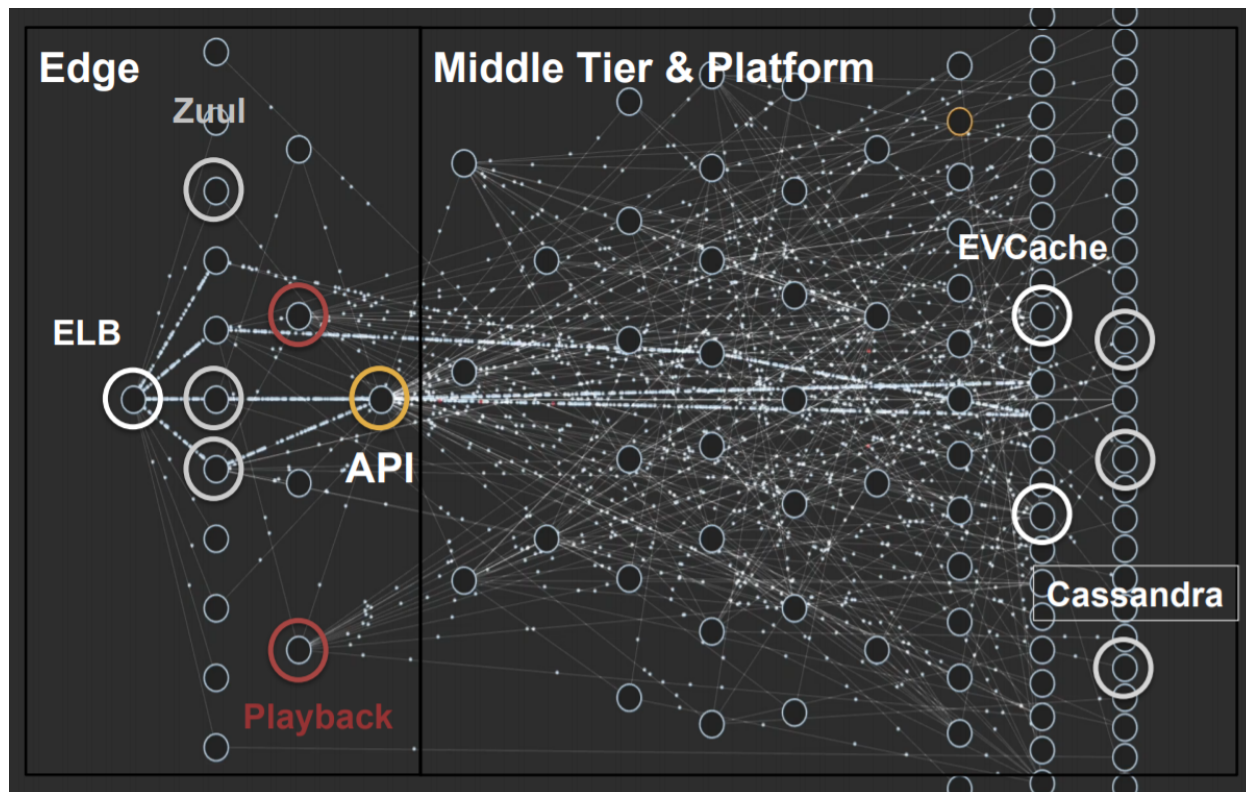


Figura 6.1: Arhitectura Aplicației Netflix [12]

Făcând o scurtă analiză asupra diagramei, putem să observăm că prezentarea este una stratificată, nodul notat „ELB” reprezintă un balansier de capacitate elastic (Elastic Load Balancer), care se ocupă cu distribuirea uniformă a cererilor către microservicii. Stratul în care avem nodul notat „Zuul”, este un strat de tip proxy care ajută la oprirea atacurilor cibernetice. Iar zona dintre acest strat și „Middle Tier” este reprezentată de componente accesibile clienților. În partea a 2-a a diagramei, este un strat destul de stratificat de microservicii care conțin mai multă logică și servere componente din prima parte a diagramei. Ultimul strat, cel care conține și nodul notat cu „Cassandra” pe diagramă, reprezintă stratul de acces la bazele de date, datele fiind salvate separat în baza de date NoSQL. Iar între ultimele 2 straturi explicate, avem un strat care se ocupă de caching. [7]

O diagramă mai reprezentativă pentru arhitectura bazată pe microservicii a Netflix, care conține mai mult de 500 de microservicii este numită și diagrama de arhitectură „Death Star”(6.2).

Așadar, având în vedere scala la care știm deja că funcționează aplicația Netflix, putem deja considera că arhitectura bazată pe microservicii își servește bine scopul. În continuare vom discuta efectiv despre motivele, pașii, beneficiile și costurile acestei schimbări de arhitectură. [2]



Figura 6.2: Arhitectura Netflix - „Death Star” [10]

6.2.2 Motive

Unul dintre primele motive care au împins compania Netflix să își mute aplicația către cloud și odată cu asta, începerea tranziției către noua arhitectură, a fost o criză întâmpinată în 2008 când baza lor de date a fost coruptă, iar acest lucru a dus la întreruperea activității timp de 3 zile.

Al doilea motiv a fost ritmul alert în care compania creștea și numărul tot mai mare de utilizatori. Toată lumea era conștientă că numărul de utilizatori va crește și mai mult iar aplicația trebuia scalată din ce în ce mai mult.

În momentul în care vorbim de scalabilitate, putem să ne gândim la asta în 2 feluri, scalare pe verticală (termenul în engleză „scale up”) sau scalare pe orizontală (termenul în engleză „scale out”). Scalarea pe verticală poate fi văzută ca îmbunătățirea constantă a sistemului, nevoia suplimentară de memorie, spațiu pe disk, puterea de procesare mai mare. Acest lucru este posibil doar până la un anumit punct, ajungându-se la atingerea limitei tehnologice. În plus scalarea pe verticală devenind din ce în ce mai costisitoare.

Scalarea pe verticală a devenit în mod rapid, o soluție mult mai ușoară, acest tip de scalare presupune distribuirea aplicației pe mai multe sisteme care lucrează împreună. Așadar, în momentul în care aplicația trebuie scalată, este nevoie doar de introducerea unei noi componente în sistem și crearea unei noi instanțe pentru serviciul pentru care se dorește scalarea, iar având în vedere ca Netflix migra în același timp înspre cloud, activarea unei noi unități și instanțierea unui nou serviciu se poate face foarte simplu și rapid.

Mergând mai departe, se dorește eliminarea tuturor punctelor critice, ceea ce înseamnă ca se dorește ca în sistem să nu existe posibilitatea ca o eroare să se propage, generând în cascadă și alte erori ale sistemului. Soluția agreeată pentru această problemă a fost crearea de servicii fara stare (în engleză „stateless”), aceastea având proprietatea că anumite date de intrare vor fi procesate și transformate în date de ieșire în același mod indiferent de instanța serviciului. Iar ca și testare a acestei idei, cei de la Netflix folosesc un tool numit „Chaos Monkey”, care aleator distruge câte o instanță al unui serviciu pentru a se garanta că o asemenea eroare nu este propagată în întreg sistemul. Ca acest concept să funcționeze, este nevoie de un balansier, pentru ca datele să fie redistribuite către o altă instanță a aceluiași serviciu.



Figura 6.3: Chaos Monkey [12]

6.2.3 Avantaje și dezavantaje, costuri și sacrificii

Din punct de vedere al avantajelor pe care microserviciile le aduc, unul dintre ele a fost probabil cel mai apreciat de Netflix, a fost viteza de dezvoltare a produsului. Acest lucru, probabil fiind datorat faptului că fiind primul care aduce o îmbunătățire sau un lucru nou, este favorizat față de cei care îl urmează. Așadar, microserviciile au permis o redistribuire a echipelor, iar noile

echipe având responsabilități separate, conceptul de a aștepta după o altă echipă pentru a putea să îți faci treaba, începe să dispară.

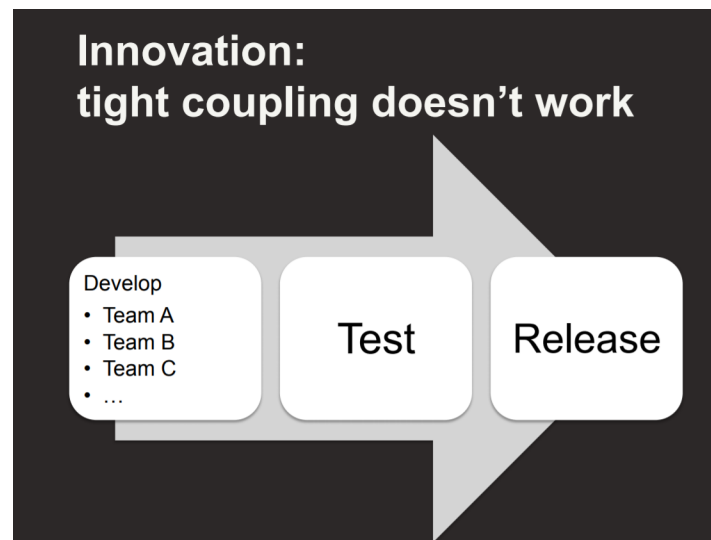


Figura 6.4: Procesul de dezvoltare într-o aplicație monolit [12]

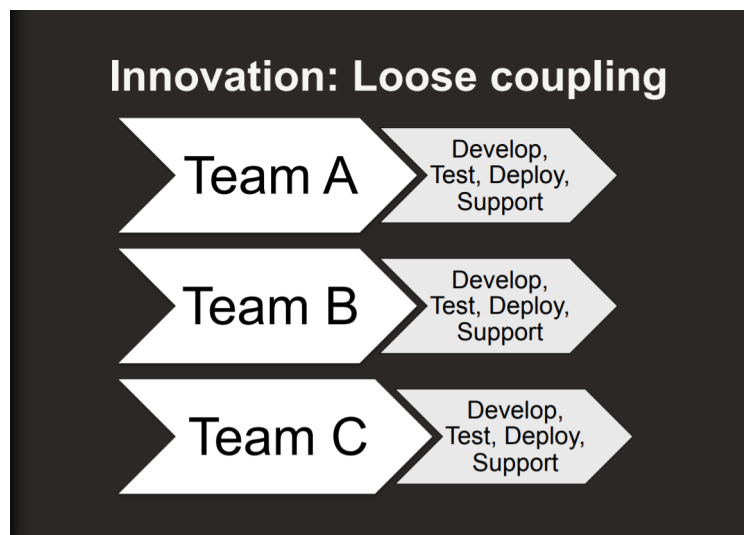


Figura 6.5: Procesul de dezvoltare într-o aplicație bazată pe microservicii [12]

După cum se poate observa în prima figură (6.4), în prima parte trebuie ca toată aplicația să fie dezvoltată, apoi testată și apoi lansată. Iar în a doua figură (6.5), se poate observa că echipele pot să lucreze fiecare în ritmul lor, nefiind nevoiți să depindă constant de restul. Microserviciile nefiind nevoite să fie lansate în execuție împreună.

Pe de altă parte, trebuie să avem în vedere faptul că în această schimbare au fost nevoie și de sacrificii, unele referindu-se la costuri, altele la relație oamenilor din echipe. Printre acestea se pot număra următoarele:

- În primul rând, costul de dezvoltare și mentenanță a aplicație a crescut. Motivul fiind următorul: tranziția a fost una de durată, iar în timp ce se crea aplicația cu noua arhitectură, aplicația monolit trebuia să rămână în picioare. Și pe de altă parte, datele trebuiau să fie replicate și salvate atât în aplicația nouă cât și în cea curentă.
- În al doilea rând, tehnologiile din noua aplicație s-au schimbat parțial față de aplicația monolit, iar compania trebuie să suporte toate tehnologiile existente din ambele proiecte. Un exemplu este legat de bazele de date, în aplicația monolit existând baze de date relaționale, iar bazele de date din microservicii au devenit baze de date NoSQL. Mai târziu evoluția noii arhitecturi vor apărea microservicii scrise și în NodeJS, Python și alte limbaje de programare.
- În al treilea rând, un alt compromis, poate nu atât de mare, a fost în momentul în care compania a ales să trăiască într-o zonă hibridă, asta însemnând că au început să existe servicii scrise în diferite limbaje de programare cum ar fi NodeJS, Python și altele. Acest lucru fiind posibil atât timp cât era respectat un format pentru datele de intrare cât și pentru datele ieșire.
- În ultimul rând, un alt compromis care a trebuit făcut a fost schimbarea structurii echipelor. De la echipe specializate pe dezvoltare, echipe specializate pe testare și echipe specializate în lansarea aplicației în mediul online, s-a ajuns la echipe mai mici care să se ocupe de întreg ciclul de viață al unui microserviciu(6.6), De la implementare, până la lansarea lui în execuție.

6.2.4 Descrierea tranziției către microservicii

Unul din cei mai importanți oameni care a ajutat la tranziția aplicație către microservicii și mutarea acestora în cloud, este specialistul în scalabilitate de la Netflix, Adrian Cockcroft. După cum am amintit și mai sus, posibilitățile de scalare erau pe verticală și pe orizontală. După cum Adrian Cockcroft a susținut în timpul unei conferințe, scalarea pe verticală ar fi însemnat în momentul acela, anul 2009, construirea unui nou centru de date (în engleză „data center”) cu un cost estimativ de aproximativ 100 de milioane de dolari. Iar pe aceeași temă, a glumit, spunând că banii respectivi ar fi mai bine folosiți dacă Netflix ar cumpăra încă un sezon din „House Of Cards”(acesta fiind un serial TV). Soluția plauzibilă care a rămas, fiind scalarea pe orizontală. [10]

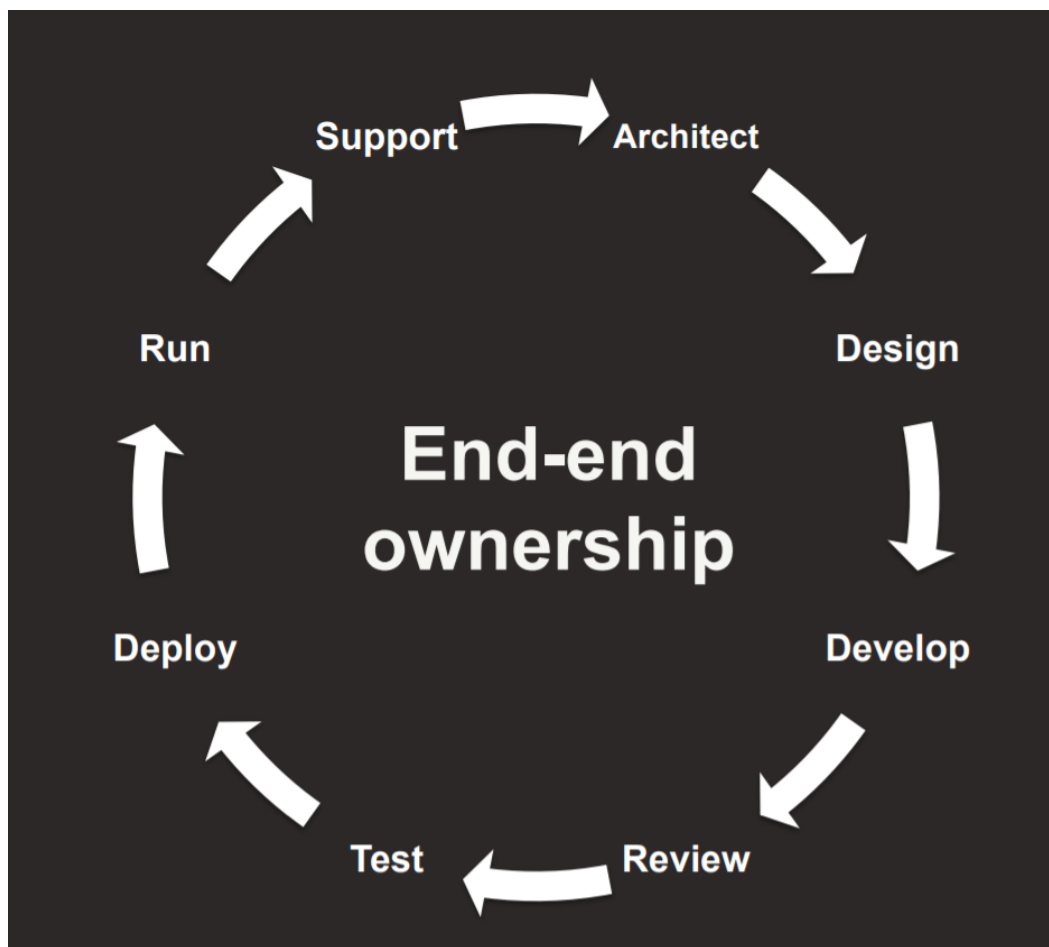


Figura 6.6: Ciclul de viață al unui microserviciu [12]

Primul pas fiind de test, s-a încercat mutarea unui serviciu în cloud, iar alegerea nu a fost întâmplătoare, a fost ales un serviciu care să nu fie în prima linie pentru utilizator, adică un serviciu care mai mult procesează cantități mari de date în spate. Spre exemplu, algoritmul folosit pentru auto completarea câmpului de cautare. După ce acest pas a fost realizat cu succes, și serviciul era folosit de aplicația monolit, procesul a continuat.

Un alt pas important a fost schimbarea bazei de date, de la baze de date relaționale Oracle, la baza de date NoSQL („Cassandra”). Acesta bază de date este folosită și acum pentru aplicația Netflix. Având în vedere că este open source (în engleza „open source”), Netflix a dezvoltat un feature care ajută la replicarea ușoară a datelor.

După ce întreaga tranziție a fost gata, după anul 2012, multe dintre soluțiile aplicate în timpul tranziției au fost publicate, sursele devenind surse deschise.

Așadar succesul Netflix se datorează în mare parte lui Adrian Cockcroft, care este un vizionar, reușind tranziția Netflix către o arhitectură bazată pe microservicii în cloud, într-un moment în care nimeni nu dorea să creadă că cloud-ul ar putea să fie o soluție(6.7).

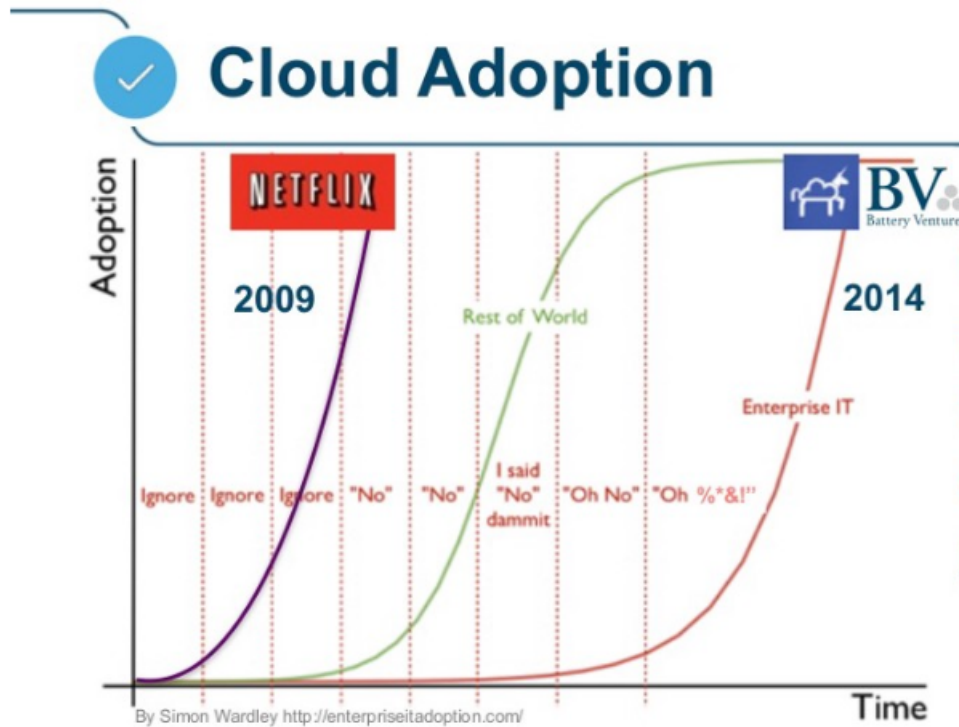


Figura 6.7: Migrarea către cloud a aplicațiilor [10]

Capitolul 7

Aplicație Practică

7.1 Idee

Ideea creării aplicației a venit de la o problemă personală, iar aceasta ar fi următoarea: în fiecare luna scrisorile cu facturi ajung în poștă, iar pentru ca nu vin toate în aceeași dată și pentru că nu vin în aceeași data a lunii, trebuie mereu să verifici posta ca să nu uiti să plătești o factură. În al doilea rând trebuie să merg cu facturile la bancă ca să le platesc, sau poate chiar la firma de la care am primit factura, în caz că nu suportă plata prin bancă. Așadar am decis să creez o aplicație care să ajute în acest scop, iar această aplicație ar putea avea urmatorul enunț.

Se dorește construirea unui prototip al unei platforme care să acționeze ca un intermediar între oameni și furnizorii de utilități, în care un utilizator să poată să primească facturile de la toate sau o parte din utilitățile pe care le folosește, să poată să își încarce o anumită sumă de bani pe platforma (în varianta prototip nu este necesară utilizarea banilor reali), să poată să folosească banii introduși pentru a achita facturile, și să poată vedea detaliile facturilor cât și a tranzacțiilor pe care le efectuează pe platformă. Pentru ca relația dintre utilizatorii normali și furnizori să poată fi creată ușor, conturile utilizatorilor vor folosi CNP-ul pentru a fi create. Din punctul de vedere al furnizorilor de utilități, aceștia pot să fie capabili să „își conecteze” aplicațiile actuale la platformă, aceasta furnizând puncte de acces pentru adăugarea facturilor, chitanțelor, și pentru cererea unei statistici a tranzacțiilor făcute către ei între două date alese. Toate plățile se vor face pe platformă, acestea efectuându-se fără transfer direct de bani între platformă și furnizori. Platforma va efectua un singur transfer de bani, la finalul zilei, un singur transfer cu suma totală a platilor efectuate către furnizorul în cauză. După cum este specificat și mai sus, transferurile nu vor folosi bani reali. Deoarece furnizorii se vor conecta la aplicație prin call-uri la terminalele oferite de aplicație, aceștia vor trimite la fiecare call, un set de date de autentificare generate pentru fiecare furnizor în

parte, în momentul în care un furnizor dorește să se lege la platformă.

Având în vedere cerința prezentată, putem deduce că vom avea două tipuri de utilizatori, utilizatori normali, care vor plăti facturi și furnizorii, care vor adauga facturi, chitanțe și vor genera rapoarte. Putem determina următoarele funcționalități în funcție de tipul utilizatorului:

Pentru utilizatorii direcți ai platformei avem următoarele funcționalități:

- Înregistrare
- Autentificare
- Introducere de bani pe platformă
- Vizualizarea facturilor pentru fiecare utilitate
- Plata facturilor
- Vizualizarea tranzacțiilor făcute pe platformă
- Vizualizarea chitanțelor.

Pentru furnizorii de utilități care doresc conectarea la platformă avem următoarele funcționalități:

- Adăugarea de facturi
- Adăugarea chitanțelor după ce o factură a fost plătită
- Generarea unui raport cu toate tranzacțiile făcute între două date
- Recunoașterea/Autentificarea fiecărui furnizor cu ajutorul unor credențiale alese

7.2 Arhitectura aplicației

Aplicația creată are la bază o arhitectură bazată pe microservicii, acest lucru oferind aplicației scalabilitatea necesară, în momentul în care numărul de utilizatori va crește foarte mult, iar acest lucru este datorat de existența unui microserviciu dedicat utilizatorilor. Tot această arhitectură poate oferi flexibilitatea creării unui nou furnizor în sistem, doar prin lansarea unui nou microserviciu cu un fișier de configurare specific pentru un nou furnizor.

În figura de mai jos, figura 7.1, se poate observa că aplicația poate fi împărțită din punct de vedere al arhitecturii în trei straturi și o componentă ajutătoare denumită „Common”.

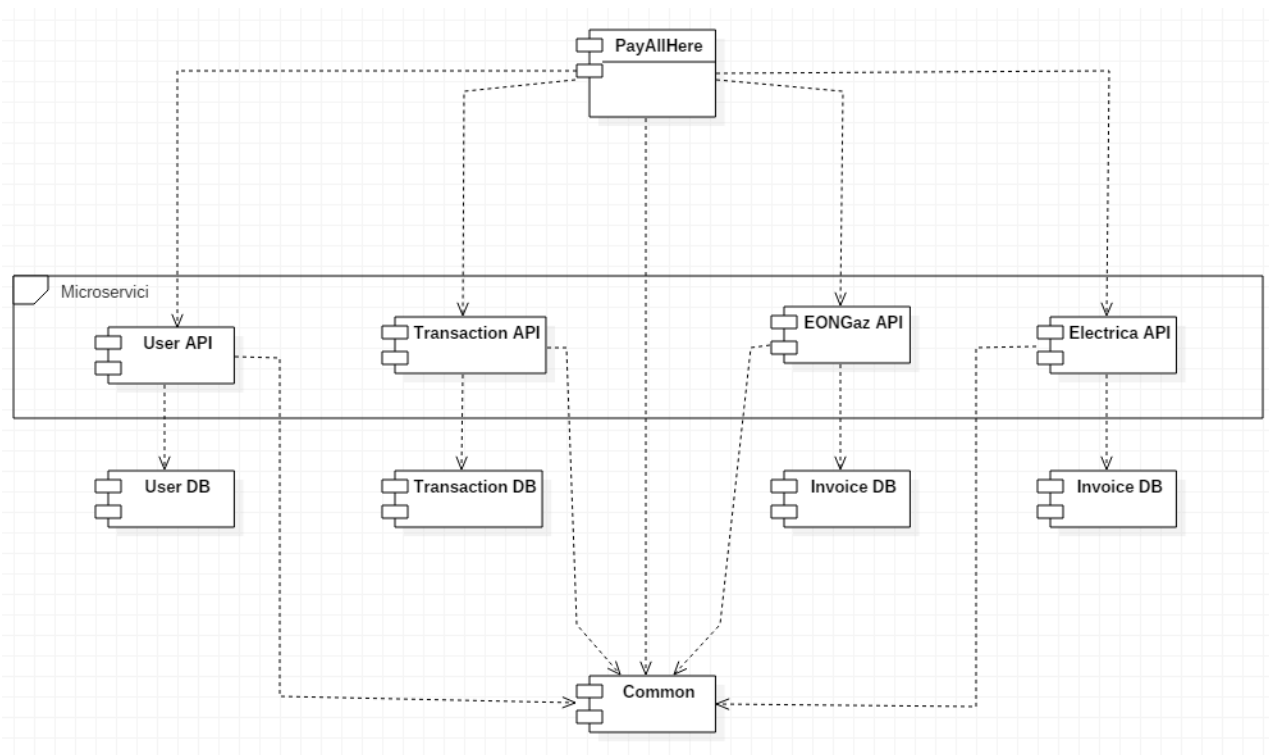


Figura 7.1: Diagrama de arhitectură

În stratul superior se poate observa componenta „PayAllHere” care reprezintă de fapt poarta de intrare a unei arhitecturi bazate pe microservicii (în engleza „gateway”). Aceasta este de fapt o aplicație de tip MVC, a cărei singură dependență din proiect este componenta „Common”, dar acest fapt nu crează un cuplaj între componente, deoarece aceasta este doar o librărie (în engleză „Class Library”). Această componentă doar cere informații din stratul următor prin call-uri asincrone, ceea ce înseamnă ca există un cuplaj slab între cele două straturi.

Al doilea strat al aplicației este format din microservicii. Acestea sunt unități independente ale aplicației, nu sunt cuplate între ele, și nu au stare. Fiecare microserviciu se ocupă independent de câte o secțiune a aplicației, în acest fel, avem un microserviciu pentru utilizatori, acesta se ocupă de înregistrarea utilizatorilor, autentificarea lor, cât și să păstreze date despre utilizatori. Un al doilea microserviciu este cel care se ocupă de tranzacții, acesta gestionează tranzacțiile care se întâmplă pe platformă, cât și de filtrarea unor tranzacții după anumite criterii: data, persoane, etc..

Al treilea și al patrulea microserviciu se ocupă de gestionarea facturilor, fiecare pentru un furnizor specific, ceea ce este bine de precizat, este faptul că structura celor două microservicii este identică, iar diferențele dintre ele se pot face doar folosindu-se un fișier de configurare, ceea ce înseamnă că adăugarea unui nou furnizor va necesita foarte puțin timp de dezvoltare. Tot procesul

fiind doar o lansare în execuție a unui nou microserviciu cu un fișier de configurare schimbat. O prezentare a felului în care un microserviciu de acest tip este implementat se poate observa în diagrama de clase prezentată mai jos, figura 7.2

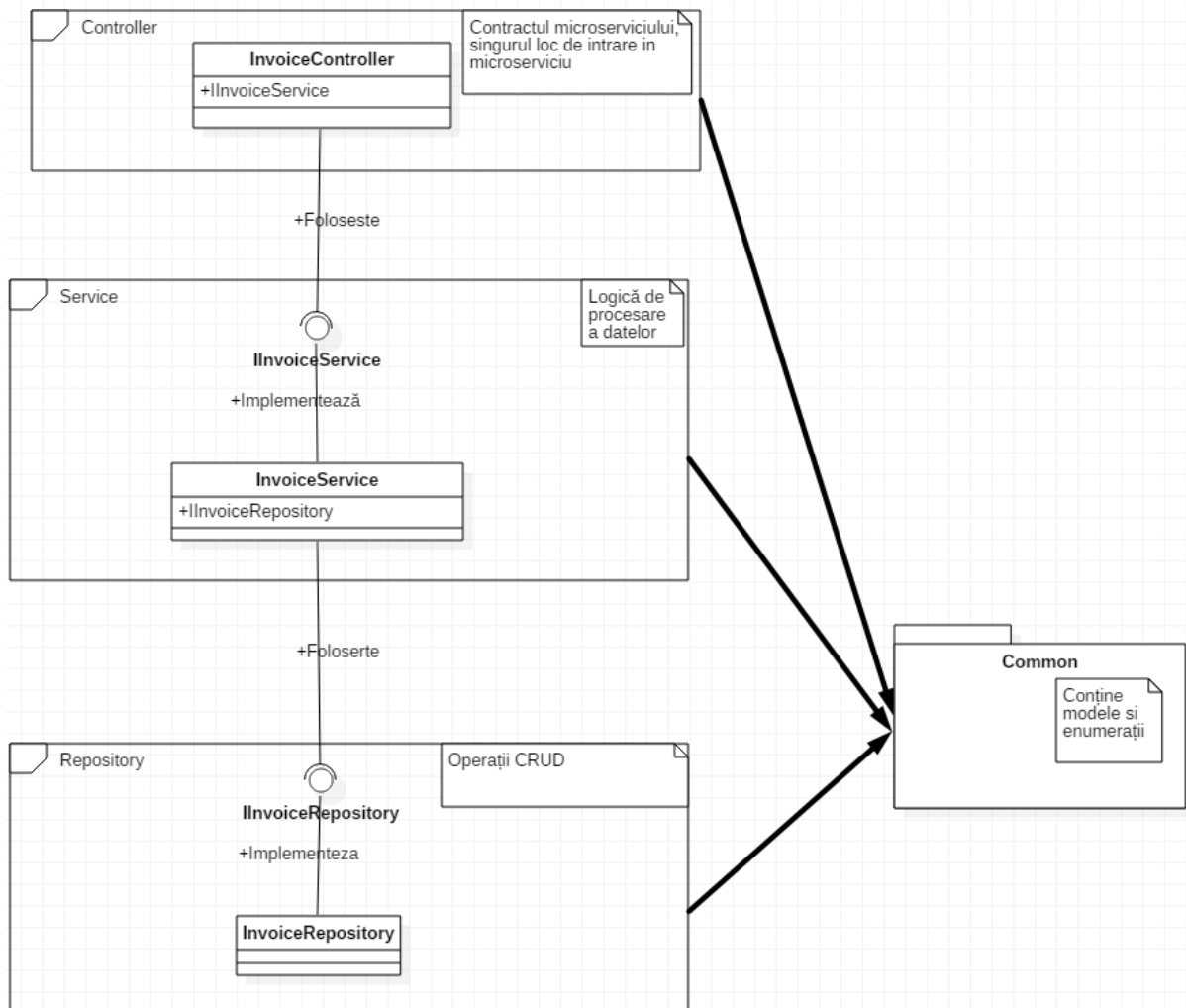


Figura 7.2: Diagrama de clase a unui microserviciu atribuit unui furnizor

Al treilea strat este reprezentat de stratul de persistentă. După cum se poate observa în figura 7.1, fiecare microserviciu are câte o bază de date la care are acces unic, ceea ce înseamnă că persistența datelor nu reprezintă un factor de cuplaj între microservicii.

Ultima componentă este componenta „Common”, în această componentă se găsesc modele și enumerații care se folosesc atât în aplicația MVC, cât și în microservicii, acest lucru nu cuplează

serviciile, doar se asigură că informația trimisă dintr-o componentă înspre alta să poată sa fie recepționată în același format în care a fost transmisă.

7.3 Persistenta datelor

Dacă e să vorbim de persistența datelor, așa cum s-a menționat mai sus, fiecare microserviciu are acces la propria lui bază de date (7.3).

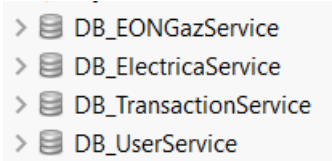


Figura 7.3: Bazele de date pentru fiecare microserviciu

Baza de date folosită este una non relațională, această alegere a fost făcută deoarece este ușor de gestionat. În cazul unei modificări în cod asupra clasei salvate în baza de date, schimbarea se va propaga în baza de date fără ca programatorul să depună un efort suplimentar. Fiecare bază de date conține câte o colecție de obiecte (7.4). Așadar, fiindcă baza de date nu este relațională, orice legătură între două clase diferite trebuie efectuată la nivel de logică în cod.

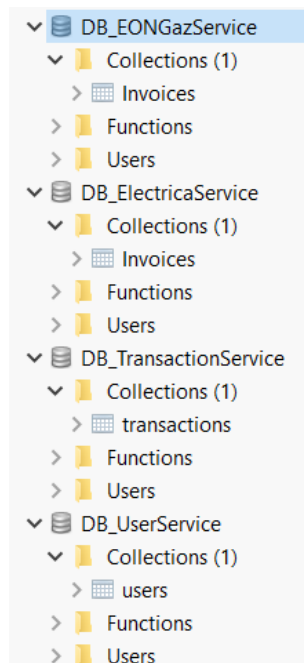
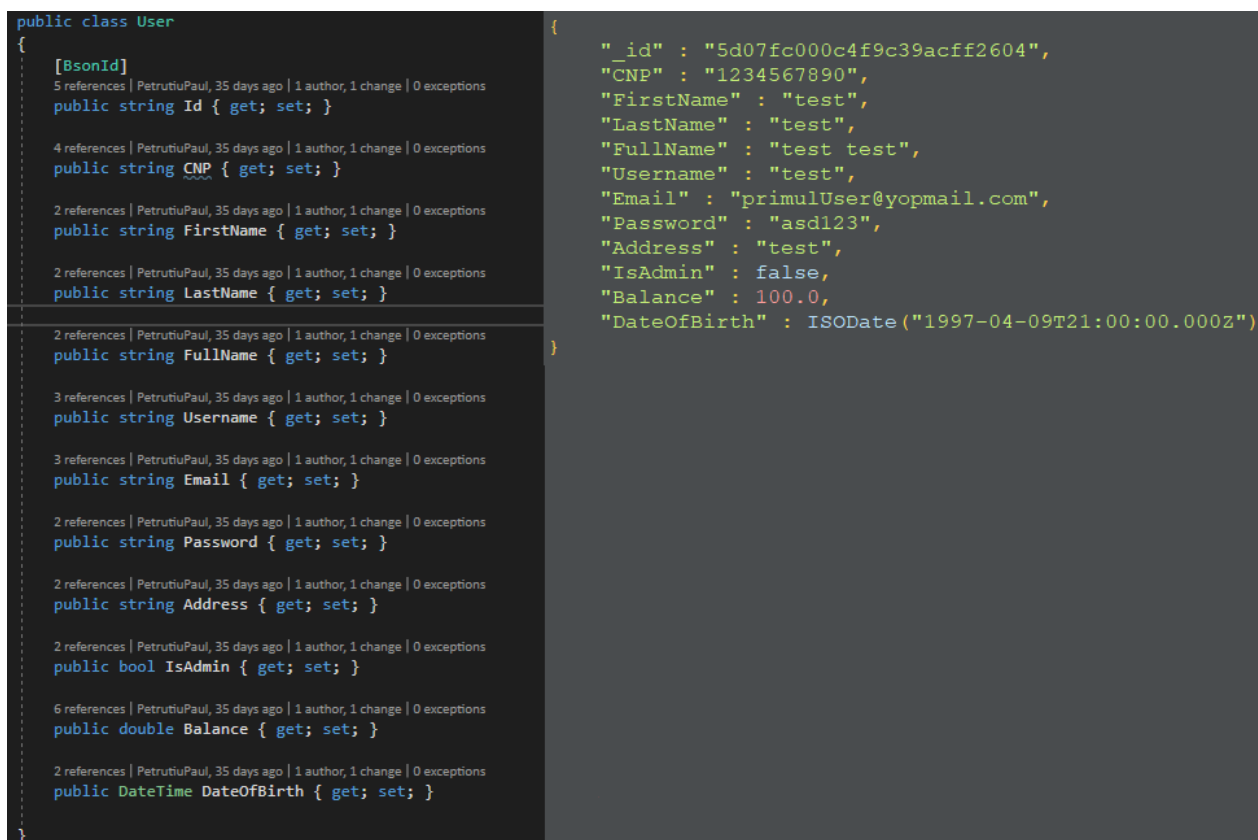


Figura 7.4: Componenta bazelor de date

Baza de date folosita este MongoDB. In această bază de date se pot salva colectii care au o structura fluidă, dar elementele vor fi salvate in funcție de modelul clasei corespunzătoare. Fiecare obiect este salvat in baza de date serializat JSON. Un exemplu se poate observa mai jos in figura 7.5.



```
public class User
{
    [BsonId]
    5 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public string Id { get; set; }

    4 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public string CNP { get; set; }

    2 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public string FirstName { get; set; }

    2 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public string LastName { get; set; }

    2 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public string FullName { get; set; }

    3 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public string Username { get; set; }

    3 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public string Email { get; set; }

    2 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public string Password { get; set; }

    2 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public string Address { get; set; }

    2 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public bool IsAdmin { get; set; }

    6 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public double Balance { get; set; }

    2 references | PetrutuPaul, 35 days ago | 1 author, 1 change | 0 exceptions
    public DateTime DateOfBirth { get; set; }
}

{
  "_id" : "5d07fc000c4f9c39acff2604",
  "CNP" : "1234567890",
  "FirstName" : "test",
  "LastName" : "test",
  "FullName" : "test test",
  "Username" : "test",
  "Email" : "primulUser@yopmail.com",
  "Password" : "asd123",
  "Address" : "test",
  "IsAdmin" : false,
  "Balance" : 100.0,
  "DateOfBirth" : ISODate("1997-04-09T21:00:00.000Z")
}
```

Figura 7.5: Obiect în cod C# și obiect în baza de date

Singurul lucru care diferă este reprezentare identificatorului unic, care in cod C# este „Id” iar in baza de date este „_id”, acest lucru se datorează atributului „[BsonId]” care marchează câmpul „Id” ca fiind identificator unic, iar reprezentarea identificatorului unic in baza de date MongoDB este „_id” (figura 7.5).

7.4 Interacțiunea cu utilizatorul

Din punct de vedere al utilizatorilor, aceștia se împart în doua categorii, utilizatori externi și furnizori de utilități (gaz, curent, etc.). Iar în funcție de acest criteriu și funcționalitățile pe care deja le-am definit, putem sa considerăm următoarele cazuri de utilizare, prezentate în diagrama 7.6.

Este important faptul că doar utilizatorilor externi le este adresată o interfață grafică, furnizorilor oferindu-li-se acces la anumite puncte de intrare („endpoints”) în aplicație la care se pot conecta cu ajutorul unor apeluri făcute din aplicațiile proprii.

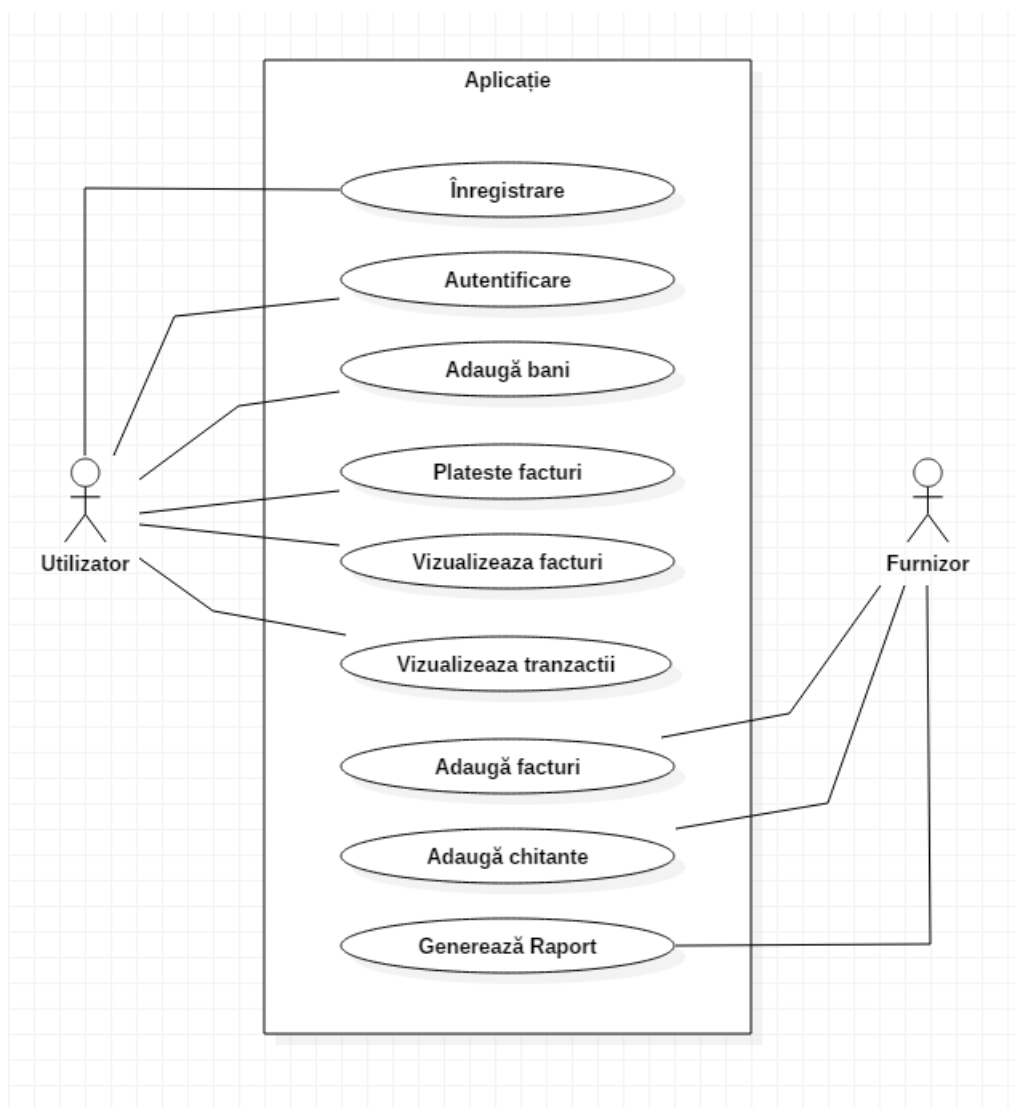


Figura 7.6: Diagrama cazurilor de utilizare

În continuare vom exemplifica în detaliu câte un caz de utilizare pentru fiecare dintre tipurile de utilizatori. Pentru utilizatorii externi vom parcurge fiecare pas din fluxul de lucru (în engleză „workflow”) al plății unei facturi, iar pentru utilizatorii de tip furnizor, vom parcurge fiecare pas din fluxul de lucru pentru generarea unui raport pentru doua date selectate. Pentru o mai bună înțelegere se va prezenta câte o diagrama de secvență pentru fiecare funcționalitate.

7.4.1 Plata unei facturi

Acest caz de utilizare a fost ales deoarece este unul mai complex, acesta implicând aproape toate microserviciile. Pentru completarea cu succes al acestei operații, se vor efectua următoarele evenimente:

- Utilizatorul se autentifică
- Utilizatorul accesează pagina cu furnizori
- Utilizatorul deschide lista de facturi de la un furnizor (spre exemplu: facturile de curent)
- Alege factura pe care dorește să o plătească
- Introduce suma pe care dorește să o plătească
- Se actualizează balanța utilizatorului
- Se salvează tranzacția
- Se modifică suma rămasă de plată pentru factura respectivă.
- Utilizatorul este întors pe pagina cu facturi.

O secvență de plată a unei facturi se consideră un succes în momentul în care se salvează cu succes tranzacția efectuată, iar balanța unui utilizator cât și suma rămasă de plată este actualizată corect.

Unul din motivele pentru care o secvență de plată nu se va executa cu succes poate fi produsă de încercarea plății unei sume care nu este disponibilă în contul curent. În acest caz se va afișa un mesaj corespunzător, iar utilizatorul poate ulterior să încarce o sumă de bani pe platformă iar apoi să reia procesul de plată a facturii.

După cum se poate observa și în figura 7.7, utilizatorul interacționează cu aplicația MVC, iar aceasta comunică prin apeluri sincrone (dar care pot fi paralelizate), cu microserviciile care se ocupă de datele utilizatorului, datele despre facturile furnizorului cât și de tranzacții. Apelurile între microservicii sunt apeluri de tip HTTP. În diagrama 7.7 nu este reprezentat și stratul de persistență, dar putem considera ca fiecare apel la un microserviciu determină o interogare pe baza de date.

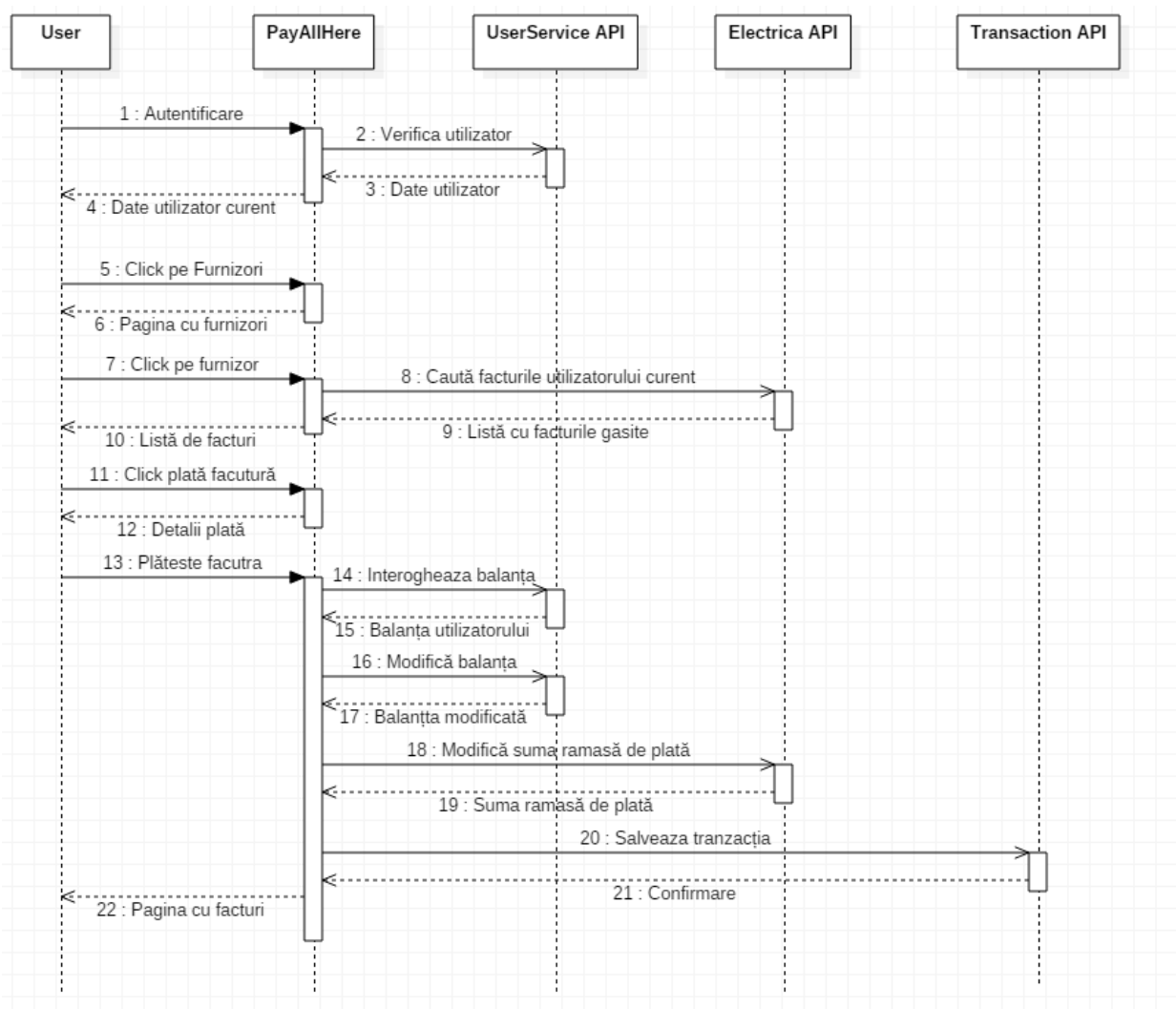


Figura 7.7: Diagrama de secvență pentru plata unei facturi

7.4.2 Generarea unui raport

Pentru generarea unui raport se va efectua un apel din afara aplicației (din aplicația furnizorului), aceasta va fi recunoscută de platformă, identificând furnizorul de la care vine apelul, va genera un raport sub forma unui fisier excel, iar acesta va fi trimis ca raspuns. Pentru completarea cu succes al acestei operații se vor efectua urmatoarele evenimente:

- Furnizorul face un apel catre aplicație
- Aplicația selecteaza toate tranzacțiile între cele doua date selectate
- Aplicația generează un document excel
- Aplicația transmite un raspuns cu documentul atașat

O secvență executată cu succes se consideră atunci când generarea raportului in excel este reușită. O diagramă de secvență pentru acest caz este prezentată în figura 7.8.

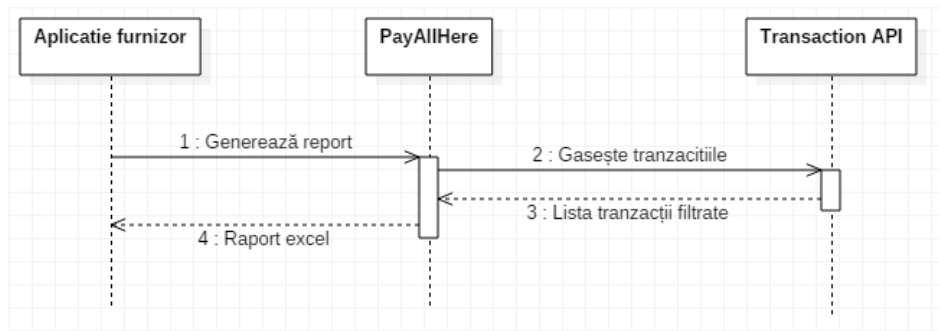


Figura 7.8: Diagrama de secvență pentru generarea unui raport

7.5 Specificații tehnice

Tehnologiile folosite pentru dezvoltarea platformei sunt:

- ASP .NET Core 2.1
- MongoDB 4.0
- JQuery

Pentru implementare s-a folosit mediul de dezvoltare Microsoft Visual Studio 2019, iar pentru vizualizarea bazei de date s-a folosit Robo 3T.

Platforma are o singura poartă de intrare, aceasta fiind reprezentată de aplicația MVC, iar procesarea datelor sa face în microservicii, acestea fiind aplicații Web API care au acces la baze de MongoDB. Interfața grafică prezentată utilizatorilor este construită cu ajutorului Razor View-urilor (.cshtml). În aceste view-uri s-a html, css, javascript, JQuery și JQuery

Pentru soluția prezentată a fost introdus si suport pentru Docker, deoarece lansarea microserviciilor in execuție este ușor de facut cu ajutorul containerelor. Dacă se dorește rularea aplicație in modul de dezvoltare, este nevoie de toate elementele de mai sus să fie instalate: Microsoft Visual Studio 2019, .NET Core SDK 2.1, MongoDB (trebuie rulată o instanță pe localhost pe portul de bază 27017) și Docker (nu este obligatoriu, doar dacă se dorește lansarea in execuție cu ajutorul lui).

7.6 Analiza arhitecturii

Pentru a putea analiza arhitectura aplicației și beneficiile pe care aceasta le aduce, vom reaminti că aplicația are la baza o arhitectură bazată pe microservicii și utilizează un șablon de proiectare numit „Gateway design pattern”. acest șablon subliniază ideea cum că accesul către microservicii trebuie să se facă printr-o singură poartă de acces pentru fiecare tip de aplicație (web, mobilă, etc.). În platforma construită, poarta de acces cât și aplicația web sunt reprezentate de aplicația MVC.

Așadar unul din marile avantaje pe care le aduce arhitectura bazată pe microservicii este faptul că unitățile logice au un grad mic de cuplare. Acest lucru oferă posibilitatea de a cunoaște platforma bucată cu bucată. Din acest motiv un programator nou va putea dezvolta o anumită parte din aplicație fără să cunoască în detaliu celelalte componente.

Pentru a reprezenta un alt avantaj adus de arhitectura bazată pe microservicii, ne vom baza pe un exemplu. Putem presupune că pe viitor se dorește să se introducă un algoritm de inteligență artificială, în așa fel încât în momentul în care un utilizator nou va dori să se înregistreze, acesta doar va face o poză buletinului lui, iar datele personale se vor completa automat. În primul rând este bine de știu că platforma .NET este abia la început cu suportul pe care îl oferă în materie de inteligență artificială, așadar, partea de IA ar putea să fie făcută în python. Acest lucru este posibil doar datorită arhitecturii bazate pe microservicii, deoarece este foarte simplu de construit un microserviciu wrapper scris în C# peste o aplicație python. Un astfel de algoritm ar avea nevoie de o putere de procesare mai mare, ceea ce poate fi oferită ușor de scalarea unui singur microserviciu. De exemplu acest lucru nefiind capabil într-o aplicație monolit.

În figura 7.9 putem observa o analiză făcută asupra soluției de către Visual Studio. Cele mai relevante date care susțin ideea folosirii unei arhitecturi bazate pe microservicii, sunt datele din coloana „Maintainability Index”. Acesta este calculat în funcție de complexitatea ciclomatică, volumul Halstead și numărul liniilor de cod [14]. Formula fiind prezentată în figura 7.10. Având în vedere că numere din coloana „Maintainability Index”, au o valoare ridicată, înseamnă că aplicația poate fi ușor îmbunătățită, complexitate din fiecare microserviciu nefiind prea mare. Acest lucru implică costuri mai mici.

Hierarchy ^		Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling
▷ BuildingBlocks\Common (Debug)	■	97	148	2	12
▲ Client\PayAllHere (Debug)	■	81	324	4	214
▷ AspNetCore	■	81	224	4	125
▷ PayAllHere	■	82	7	1	35
▷ PayAllHere.Controllers	■	76	38	3	58
▷ PayAllHere.Models	■	95	4	1	0
▷ PayAllHere.Service	■	67	39	1	29
▷ PayAllHere.Service.Contracts	■	100	12	0	15
▲ Microservices\Electrica.API (Debug)	■	88	69	2	96
▷ Electrica.API	■	84	7	1	31
▷ Electrica.API.Controllers	■	75	10	2	30
▷ Electrica.API.Models	■	100	28	1	3
▷ Electrica.API.Properties	■	82	5	1	10
▷ Electrica.API.Repository	■	75	7	1	29
▷ Electrica.API.Repository.Contracts	■	100	5	0	7
▷ Electrica.API.Service	■	86	6	1	11
▷ Electrica.API.Service.Contracts	■	100	1	0	2
▷ Microservices\EONGaz.API (Debug)	■	89	64	2	86
▲ Microservices\Transaction.API (Debug)	■	86	35	2	70
▷ Transaction.API	■	84	7	1	31
▷ Transaction.API.Controllers	■	72	7	2	25
▷ Transaction.API.Models	■	100	14	1	1
▷ Transaction.API.Repository	■	81	3	1	20
▷ Transaction.API.Repository.Contract	■	100	2	0	6
▷ Transaction.API.Service	■	84	2	1	3
▲ Microservices\UserService.API (Debug)	■	85	60	2	77
▷ UserService.API	■	84	7	1	31
▷ UserService.API.Controllers	■	75	11	2	24
▷ UserService.API.Models	■	100	24	1	2
▷ UserService.API.Repository	■	75	10	1	26
▷ UserService.API.Repository.Contract	■	100	6	0	5
▷ UserService.API.Service	■	82	2	1	4

Figura 7.9: Analiza asupra codului, generată de Visual Studio 2019

Maintainability Index = MAX(0,(171 - 5.2 * log(Halstead Volume) - 0.23 * (Cyclomatic Complexity) - 16.2 * log(Lines of Code))*100 / 171)

Figura 7.10: Formula de calcul pentru „Maintainability Index”

Capitolul 8

Concluzii

Având în vedere toate aspectele care s-au discutat în această lucrare putem afirma că o arhitectură bazată pe microservicii aduce următoarele avantaje:

- Costul scalabilității scade deoarece puterea de procesare poate fi crescută doar pe microserviciile care au nevoie.
- Nu este nevoie ca un programator să cunoască toată aplicația în detaliu pentru a putea dezvolta o funcționalitate nouă.
- Productivitatea în cadrul echipei nu va fi afectată de dimensiunile aplicației.
- Aplicația nu este limitată la folosirea unui singur limbaj de programare.
- Aplicația va avea un stil aranjat, cu domenii de interes bine definite, acest lucru ducând la un cod lizibil și de calitate.
- Aplicația poate fi găzduită cu ușurință în cloud.

Toate aceste avantaje se aplică asupra aplicațiilor care au la bază o aplicație pe microservicii. Deși o astfel de arhitectură aduce atât de multe avantaje, există câteva tipuri de aplicații în care aceasta arhitectură nu ar fi una ideală, spre exemplu aplicații care primesc un flux de date constant.

Ca o concluzie personală, consider că arhitectura bazată pe microservicii este un stil arhitectural care ar trebui luat în calcul mereu pentru dezvoltarea unei noi aplicații. Nu este o arhitectură la fel de simplistă ca o arhitectură de tip monolit, dar avantajele pe care le poate oferi pot face o diferență enormă, acest fapt fiind exemplificat și în studiul de caz. În același timp, această arhitectură favorizează o calitate ridicată a codului, care, din punctul meu de vedere, este unul din cele mai importante lucruri. Așadar, arhitectura bazată pe microservicii are multe avantaje dar și

cateva dezavantaje, dar în momentul de față, aceasta arhitectură este o soluție fiabilă pentru multe aplicații și mereu ar trebui luată în calcul. Așadar, pentru o aplicație care intermediază mai mulți furnizori cu scopuri asemănătoare, cum este o platformă de plată a facturilor, arhitectură bazată pe microservicii este o alegere foarte bună.

Capitolul 9

Bibliografie

- [1] Paulo SC Alencar, Donald D Cowan, and Carlos José Pereira de Lucena. A formal approach to architectural design patterns. In *International Symposium of Formal Methods Europe*, pages 576–594. Springer, 1996.
- [2] Adrian Cockcroft. Migrating to microservices, 2014. URL: <https://www.youtube.com/watch?v=1wiMLkXz26M>.
- [3] Thomas Erl. The principles of service-orientation part 1 of 6: Introduction to service-orientation. URL: <https://searchmicroservices.techtarget.com/tip/The-principles-of-service-orientation-part-1-of-6-Introduction-to-service-orientation>.
- [4] Thomas Erl. The principles of service-orientation part 2 of 6: Introduction to service-orientation. URL: <https://searchmicroservices.techtarget.com/tip/The-principles-of-service-orientation-part-2-of-6-Service-contracts-and-loose-coupling>.
- [5] Thomas Erl. The principles of service-orientation part 5 of 6: Introduction to service-orientation. URL: <https://searchmicroservices.techtarget.com/tip/The-principles-of-service-orientation-part-5-of-6-Service-autonomy-and-statelessness>.
- [6] Thomas Erl. *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2006.
- [7] Josh Evans. Mastering chaos - a netflix guide to microservices. URL: <https://www.youtube.com/watch?v=CZ3wIuvmHeM>.

- [8] Martin Fowler. Microservicepremium, 2014. URL: <https://martinfowler.com/bliki/MicroservicePremium.html>.
- [9] Martin Fowler. Microservices, 2014. URL: <https://martinfowler.com/articles/microservices.html>.
- [10] Derrick Harris. Talking microservices with the man who made netflix s cloud famous, 2015. URL: <https://medium.com/s-c-a-l-e/talking-microservices-with-the-man-who-made-netflix-s-cloud-famous-1032689afed3>.
- [11] Rich Hilliard. Ieee-std-1471-2000 recommended practice for architectural description of software-intensive systems. *IEEE*, <http://standards.ieee.org>, 12(16-20):2000, 2000.
- [12] Ruslan Meshenberg. Microservices at netflix scale. URL: https://gotocon.com/dl/goto-amsterdam-2016/slides/RuslanMeshenberg_MicroservicesAtNetflixScaleFirstPrinciplesTradeoffsLessonsLearned.pdf.
- [13] Sam Newman. *Building microservices: designing fine-grained systems*. ” O’Reilly Media, Inc.”, 2015.
- [14] Paul Oman and Jack Hagemester. Metrics for assessing a software system’s maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–344. IEEE, 1992.
- [15] Dmitry I Savchenko, Gleb I Radchenko, and Ossi Taipale. Microservices validation: Mjolnirr platform case study. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 235–240. IEEE, 2015.
- [16] Andy Singleton. The economics of microservices. *IEEE Cloud Computing*, 3(5):16–20, 2016.
- [17] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.