UNIVERSITATEA „BABEŞ-BOLYAI”
FACULTATEA DE MATEMATICĂ ŞI 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

Capitolul 1

Introducere

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 imbunatații 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 sa fie usor de dezvoltat, pentru ca o imbunătățire sa poată fi implementată fără prea mult efort, este important ca aplicația sa aibă la baza o structură bine definită. Acest lucru face ca o aplicație să fie ușor de intreținut în decursul timpului.

Având in 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.”((Hilliard, 2000))

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

2.2 Șabloane de proiectare

Un șablon de proiectare este reprezentat ca fiind o solutie cunoscuta pentru o problema recurenta.

„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” ((Alencar, Cowan, & Lucena, 1996)). 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, in 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 impementat codul si 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 fucturile de curent, gaz, etc.. În momentul în care utilizatorul efectuează o plată, browser-ul va apela un serviciu care va procesa, în spate, aceasta tranzactie(verificarea daca suma introdusă este validă, dacă suma este disponibilă, etc.)

Sistemele ce folosesc mai multe servicii, similar cu exemplul expus mai sus, sunt considereate 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 in cadrul unei singure unitați structurale. Componentele din care este alcătuită aplicația sunt gândite in așa fel incât sa funcționeze împreună și să se folosească de acelasi spatiu de memorie si de aceleași resurse.

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

Spre exemplu, să spunem că o firmă are un sediu destul de mare care are 5 săli de conferință, având in 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 intr-o anumită zi. O astfel de aplicatie se poate realiza usor si rapid, sub forma unei aplicații web de tip server client(3.1).

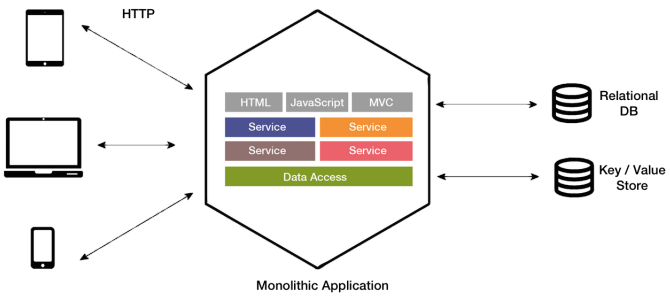

Figura 3.1: Arhitectură Aplicație Monolit(imagine : http://bits.citrusbyte.com)

3.2 Avantaje și dezavantaje

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

- Faza de dezvoltare durează mai puțin timp, IDE-urile moderne reușind să genereze mult cod automat, ceea ce implică scădearea costurilor de producție.
- Aplicația este usor de testat, automatizând procesul de testare a codului folosind o combinație de unit tests și integration test.
- Procesul de lansare în producție a unei aplicatii nu este complicat, deoarece este nevoie de rularea unei singure aplicații per server.

Pe de altă parte, in momentul în care aplicația va creste 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 ca aceștia vor intampina anumite impedimente:

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

Acestea sunt doar câteva dintre avantajele si dezavantajele unei aplicații monolit, dar sunt suficiente incât 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 parțiala sau totală a aplicației sau cel puțin este nevoie de o refactorizare care sa diminueze dezavantajele create de stilul arhitectural folosit. Majoritatea aplicațiilor încep ca și aplicații tip monolit, iar mai târziu se pot transaforma in aplicații cu arhitecturi bazate pe microservicii de exemplu.((Thönes, 2015))

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 evolutia tehnologică rapidă și numărul tot mai ridicat de persoane care aveau acces la internet.

Arhitectura bazată pe servicii (în engleză „Servici 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, si in paralel pe cât posibil.

Unul din motivele principale pentru apariția acestui stil arhitectural s-a datorat dorințe producătorilor ca o aplicație să poată sa fie folosită de pe diferite platforme. Dacă pentru fiecare platforma 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 parți mult mai mici din aplicație pentru ca aceasta să poată functiona pe diferite tipuri de platforme. Odată cu această decizie, un nou stil arhitectural a apărut.

Pentru început, s-a incercat enunțarea a 8 principii, care ar trebui respectate intr-o arhitectură bazată pe servicii.

Acestea sunt:

- Serviciile sunt autonome.
- Serviciile sunt slab cuplate
- Serviciile trebuie să abstractizeaze 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 8 principii, primele 4 sunt cele mai importate, acestea reprezentând fundația arhitecturii bazate pe servicii. Toate cele 8 principii se susțin intre ele, dar primele 4 principii induc și restul.

4.2 Principiile arhitecturii bazate pe servicii

4.2.1 Autonomia

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

Odată cu cresterea nivelului de control al unui serviciu asupra propriului context de executie, se reduc dependințele din serviciu, ceea ce ne duce spre un alt principiu, cel al cuplajului slab. Chiar dacă exclusivitatea asupra logicii pe care un serviciu o encapsulează nu poate să fie deplină, principalul obiectiv este atribuirea unui nivel rezonabil de control asupra oricărei logici pe care o prezintă intr-un moment al executiei.

Având in vedere că pot exista deferite metode de a masura autonomia, este bine facem distincție intre 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 encapsulează 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 ca intreaga logică de care se folosește un serviciu, îi serveste exclusiv lui.

Cu siguranță, se dorește ca un număr cât mai mare sa fie de servicii pur autonome, deoarece acestea ar rezolva usor problemele de concurență și ar impinge 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 servcii trebuie să fie rescrisă, iar partea de lansare în execuție a serviciului ar avea nevoie de atenție sporită. Așadar efortul si 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 știi evoluția pentru automatizarea soluților, integrarea sau schimbarea lor, acestea fiind influențate de cauze din afara mediului IT. Având in vedere faptul ca în cazul unei modificăre neprevazută programatorii trebuie să fie gata să răspundă într-o manieră eficientă, un lucru necesar este lucrul cu forme abstracte ale serviciilor si mesajelor. Acest fapt, susține agilitatea de a compune o soluție folosind serviciile disponibile.

Putem spune că cuplajul între doua unități logice și structurale poate fi văzut ca o măsurătoare a dependințelor dintre ele. Ceea ce înseamnă ca un număr mare de dependințe poate fi definit ca și un cuplaj mare. În cazul în care nu exista dependințe între două servicii, putem spune ca ele sunt decuplate. Prin implementarea consisteantă 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 in noi compoziții, și care pot fi întreținute ușor, fără a influența alte servicii.

4.2.3 Abstractizarea

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

Dacă ar fi să discutăm despre cantitatea de logică pe care un serviciu o poate expune, putem spune ca nu există nici o limitare din acest punct de vedere. Un serviciu poate fi creat cu scopul de a servii 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 in care un contract este conceput. Cu cât mai multă informație este expusa prin contract, cu atat mai puțină informație poate fi abstractizată. Cu cât un contract este mai generic, cu atat mai mult crește capacitatea lui de a fi reutilizat.

4.2.4 Contractul Formal

În mod normal, serviciile trebuie definite in 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 ajuta 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 sa definească urmatoarele lucruri:

- Punctele de acces in 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 si caracteristicile serviciului.

Definirea contractelor necesită o atenție sporită pentru ca acestea sunt distribuite printre servicii. După momentul in care sunt lansate in execuție, definiția contractelor s-ar putea să devină o dependința pentru o sursa 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 baza intr-o arhitectura bazată pe servicii. În plus acest concept susține alte principii despre care am discutat, aceastea fiind: abstractizarea și cuplajul slab, dar și alte principii care nu au fost abordate in detaliu precum: compozabilitatea si descoperirea serviciilor.

4.2.5 Șabloane de proiectare într-o arhitectură bazată pe servicii

Capitolul 5

Sabloane de proiectare bazate pe microservicii

5.1 Sabloane de implementare a microservicilor

5.1.1 Agregator

5.1.2 API Gateway

5.2 Sabloane arhitecturale de proiectare

5.2.1 Service bus

5.2.2 Procesare asincrona

5.2.3 Agregarea proceselor

5.2.4 Alte Sabloane

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 si seriale. Netflix fiind un gigant in industria televiziunii online. Aplicația netflix este o aplicație la scară mondială, care în momentul in care firma a horărat schimbarea arhitecturii avea un trafic de 8 milioane de utilizatori, ajungând la finalul anul 2018 la 139 de milioane de utilizatori.

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

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

6.2 Tranziția către microservicii în compania Netflix

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

- Ruslan Meshemberg
- Adrian Cockcroft
- Josh Evans

6.2.1 Arhitectura aplicației de azi

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

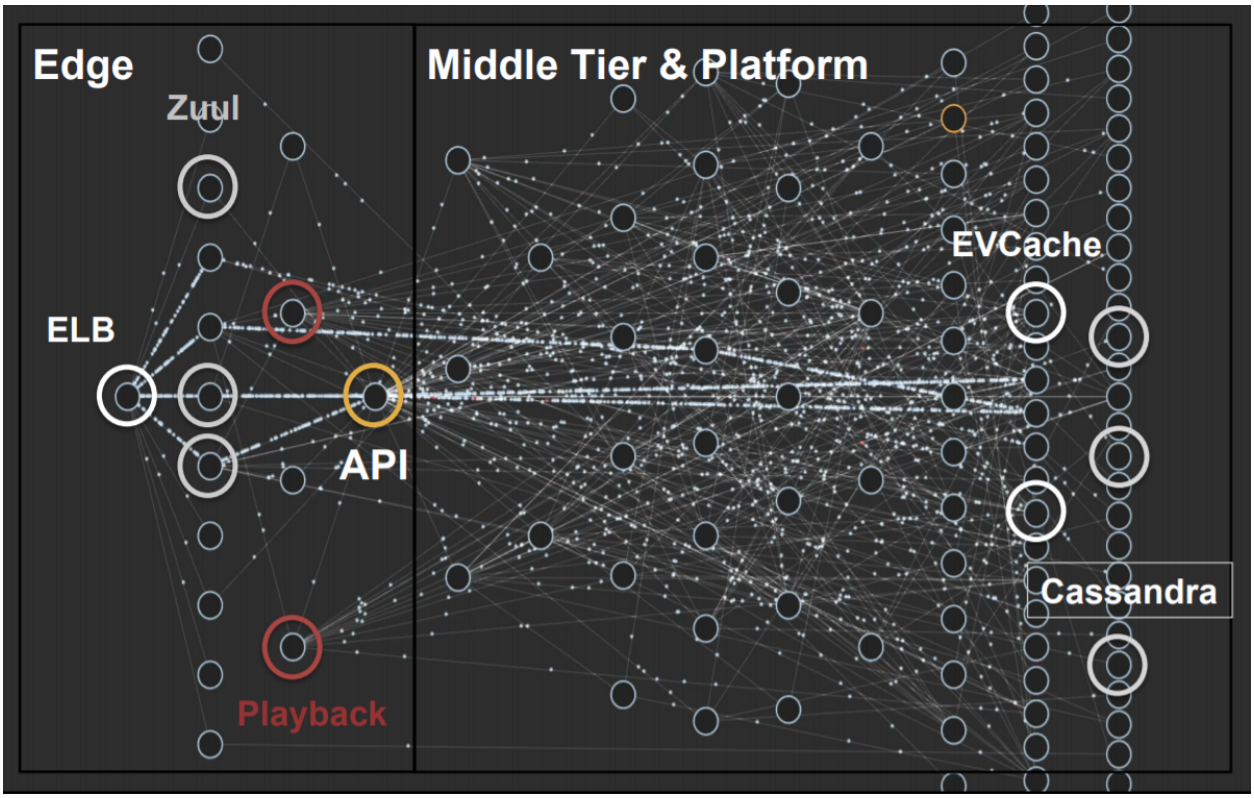

Figura 6.1: Arhitectura Aplicației Netflix()

Făcând o scurtă analiză asupra diagramei, putem să observam că prezentarea este una stratificată, nodul notat „ELB” rebrezintă un balansier de capacitate elastic (Elastic Load Balancer), care se o ocupă cu distrubuirea uniforma a cererilor catre microservicii. Stratul in care avem nodul notat „Zuul”, este un strat de tip proxy care ajută la oprirea atacurilor cibernetice. Iar zona dintre acest strat si „Middle Tier” este reprezentată de componente accesibile clientilor. În partea a 2-a a diagramei, este un strat destul de stratificat de microservicii care conțin mai multa logică si serversc componentelor din prima parte a diagramei. Ultimul strat, cel care contine si nodul notat cu „Cassandra” pe diagramă, reprezintă stratul de acces la bazele de date, datele fiind salvate separat in baza de date NoSQL. Iar intre ultimele 2 straturi explicate, avem un strat care se ocupa de caching.

O diagramă mai reprezentativă pentru arhitectura bazată pe microservicii a Netflix, care contine mai mult de 500 de microservicii este numita si diagrama de arhitectura „Death Star”(6.2).

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

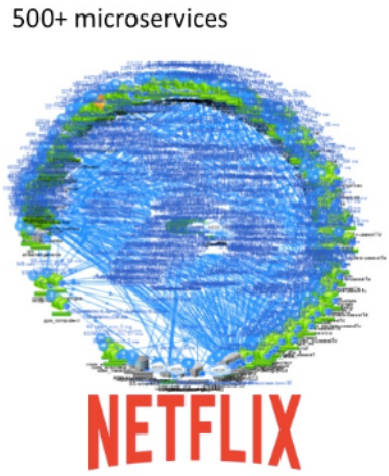

Figura 6.2: Arhitectura Aplicației Netflix()

6.2.2 Motive

Unul dintre primele motive care au înpins compania netflix să își mute aplicația catre cloud și odata cu asta, inceperea tranziției către noua arhitectură, a fost o criză întampinată în 2008 când baza lor de date a fost coruptă, iar acest lucru a dus la intreruperea activitații timp de 3 zile.

Al doilea motiv a fost ritmul alert în care compania crestea si numarul tot mai mare de utilizatori. Toată lumea era conștientă că numărul de utilizatori va creste si mai mult iar aplicația trebuia scalată din ce in ce mai mult.

În momentul în care vorbim de scalabilitate, putem sa ne gândim la asta in 2 feluri, scalare pe veriticala (termenul în engleză „scale up") sau scalare pe orizontală (termenul în engleză „scale out"). Scalarea pe verticală poate fi văzută ca imbunătățirea constantă a sistemului, nevoia suplimentară de memorie, spatiu pe disk, puterea de procesare mai mare. Acest lucru este posibil doar pană la un anumit punct, ajungându-se la atingearea limitei tehnologice. În plus scalarea pe verticală devenind din ce in 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 lucreaza împreuna. Așadar, în momentul în care aplicația trebuie scalată, este nevoie doar de introducerea unei noi componente in sistem si crearea unei noi instanțe pentru serviciul pentru care se doreste scalarea, iar având in vedere ca Netflix migra in același timp înspre cloud, activarea unei noi unitați si 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 sa nu existe posibilitatea ca o eroare sa se propage, generând in cascadă și alte erori ale sistemului. Soluția agreeată pentru această problemă a fost crearea de servicii fara stare

(în engleză „stateless”), aceestea având proprietatea că anumite date de intrare vor fi procesate si transformate în date de ieșire in același mod indiferent de instanța serviciului. Iar ca si testare a acestei idei, cei de la Netflix folosesc un tool numit „Chaos Monkey”, care aleator distruge cate o instanța 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

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 dezovolare a produsului. Acest lucru, probabil fiind datorat faptului că fiind primul care aduce o imbunătățire sau un lucru nou, este favorizat față de cei care il urmează. Așadar, microserviciile au permis o redistribuire a echipelor, iar noile echipe având responsabilitați separate, conceptul de a astepta dupa o altă echipă pentru a putea sa iti faci treaba, incepe sa dispară.

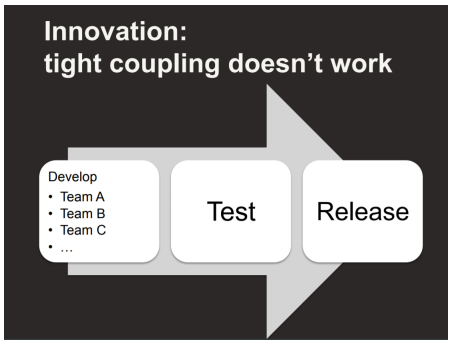

Figura 6.4: Procesul de dezovoltare intr-o aplicație monolit

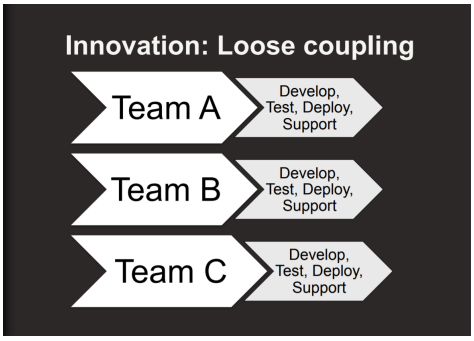

Figura 6.5: Procesul de dezovoltare intr-o aplicație bazată pe microservicii

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

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

- În primul rând, costul de dezvoltare si mentenanță a aplicație a crescut. Motivul fiind următorul: tranziția a fost una de durată, iar in timp ce se crea aplicația cu noua arhitectură, aplicația monolit trebuia sa ramână in picioare. Și pe de alta parte, datele trebuiau sa fie replicate si salvate atat in aplicația noua cât si in cea curentă.
- În al doilea rând, tehnologiile din noua aplicație s-au schimbat partial fața de aplicația monolit, iar compania trebuie sa suporte toate tehnologiile existente din ambele proiecte. Un exemplu este legat de bazele de date, in aplicația monolit existând baze de date relaționale, iar bazele de date din microservicii au devenit baze de date NoSQL. Mai târziu un evoluția noi arhitecturi vor aparea microservicii scrise si in NodeJS, Phyton si alte limbaje de programare.
- În al treilea rand, un alt compromis, poate nu atat de mare, a fost în momentul în care compania a ales să traiască intr-o zonă hibridă, asta insemnând ca au inceput sa exite servicii scrise in diferite limbaje de programare cum are fi NodeJS, Phyton si 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 si echipe specializate in lansarea aplicației in mediul online, s-a ajuns la echipe mai mici care să se ocupe de intreg ciclul de viată al unui microserviciu(6.6), De la implementare, pâna la lansarea lui in execuție.

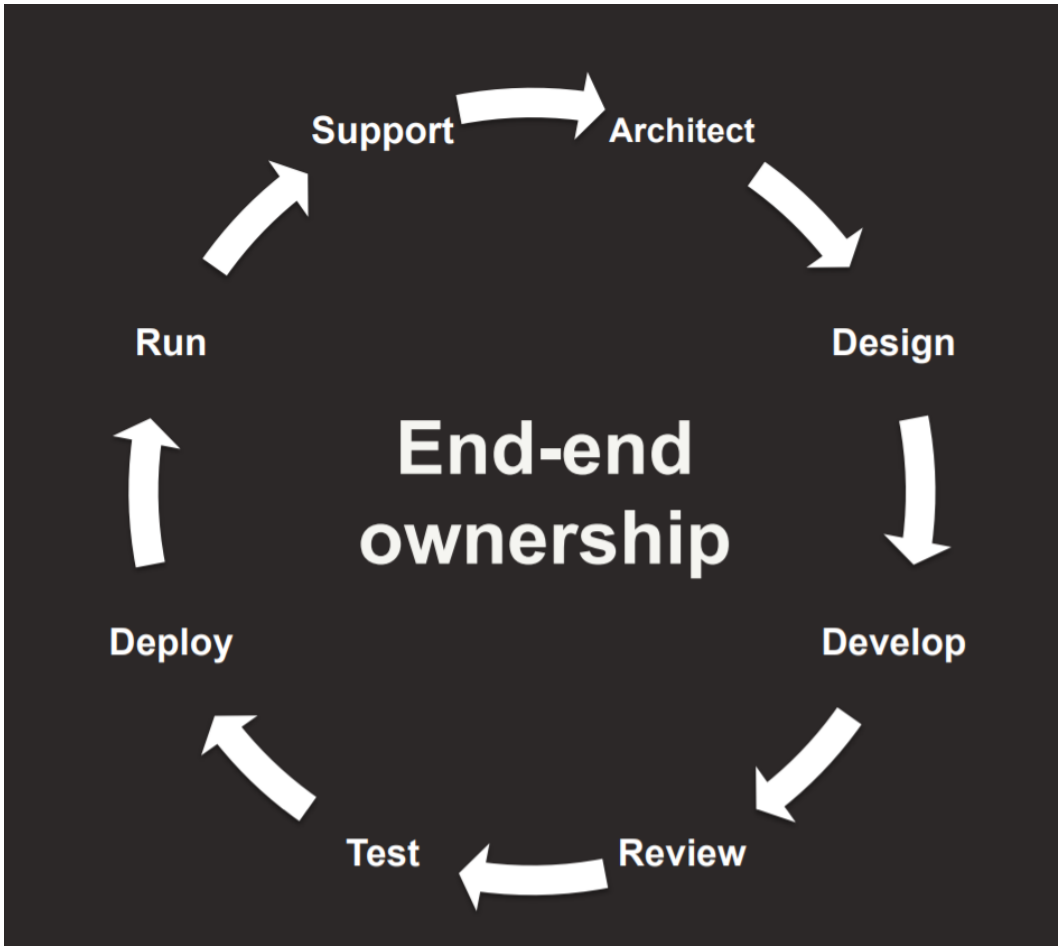

Figura 6.6: Ciclul de viață al unui microserviciu

6.2.4 Descrierea tranziției catre microservicii

Unul din cei mai importanți oameni care a ajutat la tranziția aplicație catre microservicii si mutarea acesteia in cloud, este specialistul în scalabilitate de la Netflix, Adrian Cockcroft. Dupa cum am amintit și mai sus, posibilitățile de scalare erau pe verticală si pe orizontală. Dupa cum Adrian Cockcroft a susținut in 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 aceași tema, a glumit, spunând ca banii respectivi ar fi mai bine folosiți daca Netflix ar cumpăra inca un sezon din „House Of Cards"(acesta fiind un serial TV). Soluția plauzibilă care a rămas, fiind scalarea pe orizontală.

Primul pas fiind de test, s-a încercat mutarea unui serviciu in cloud, iar alegerea nu a fost intâmplătoare, a fost ales un serviciu care să nu fie in prima linie pentru utilizator, adică un serviciu care mai mult procesează cantități mari de date in spate. Spre exemplu, algoritmul folosit pentru auto completarea câmpului de cautare. După ce acest pas a fost realizat cu succes, si 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 ca este open source(în engleza „open sourse”), Netflix a dezvoltat un feature care ajuta la replicarea usoară a datelor.

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

Așadar succesul Netflix se datorează in mare parte lui Adrian Cockcroft, care este un vizionar, reusind tranziția Netflix catre o arhitectura bazată pe microservicii in cloud, intr-un moment in care nimeni nu dorea să creadă ca cloud-ul ar putea să fie o soluție(6.7).

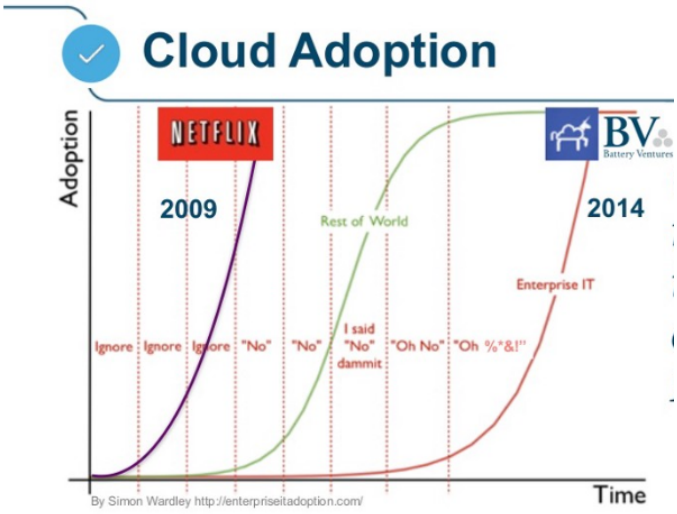

Figura 6.7: Migrarea catre cloud a aplicațiilor

Capitolul 7

Aplicație Practica

7.1 Cerința

7.2 Specificații

7.3 Arhitectura aplicatiei

7.4 Implementare

7.5 Sablone bazate pe microservicii in practica

Capitolul 8

Concluzii

Bibliografie

Alencar, P. S., Cowan, D. D., & Lucena, C. J. P. d. (1996). A formal approach to architectural design patterns. In *International symposium of formal methods europe* (pp. 576–594).

Hilliard, R. (2000). Ieee-std-1471-2000 recommended practice for architectural description of software-intensive systems. *IEEE, http://standards. ieee. org*, *12*(16-20), 2000.

Thönes, J. (2015). Microservices. *IEEE software*, *32*(1), 116–116.

Cockcroft, Adrian. 2015. Talking microservices with the man who made Netflix's cloud famous. [interviu cu] Derrick Harris. 2015.

Cockroft, Adrian. 2014. Migrating to Microservices by Adrian Cockroft. YouTube. [Interactiv] 2014. https://www.youtube.com/watch?v=1wiMLkXz26M.

Erl, Thomas. 2009. SOA Design Patterns. Crawfordville Indiana : Prentice Hall, 2009.

Fowler, Martin. 2014. Microservices, 2014,. MartinFowler.com. [Interactiv] 2014. https://martinfowler.com/articles/microservices.html.

OASIS Foundation. 2006. Reference Model for Service Oriented Architecture. [Interactiv] 2006. https://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html.