

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

1	Introducere	3
2	Concepte de bază	4
2.1	Arhitectura unei aplicații soft	4
2.2	Șabloane de proiectare	4
2.3	Serviciu Software	5
3	Aplicații Monolit	6
3.1	Ce este o aplicație monolit?	6
3.2	Avantaje și dezavantaje	7
4	Arhitectura bazată pe servicii	8
4.1	Fundamentele arhitecturii bazate pe servicii	8
4.2	Principiile arhitecturii bazate pe servicii	9
4.2.1	Autonomia	9
4.2.2	Cuplajul slab	9
4.2.3	Abstractizarea	9
4.2.4	Contractul Formal	9
5	Sabloane de proiectare bazate pe microservicii	10
5.1	Sabloane de implementare a microservicilor	10
5.1.1	Agregator	10
5.1.2	API Gateway	10
5.2	Sabloane arhitecturale de proiectare	10
5.2.1	Service bus	10
5.2.2	Procesare asincronă	10
5.2.3	Agregarea proceselor	10
5.2.4	Alte Sabloane	10
6	Studiu de Caz	11
6.1	Introducere	11
6.2	Tranziția către microservicii în compania Netflix	11
6.2.1	Arhitectura aplicației de azi	11
6.2.2	Motive	13
6.2.3	Avantaje și dezavantaje, costuri și sacrificii	14

6.2.4	Descrierea tranziției către microservicii	16
7	Aplicație Practică	18
7.1	Cerința	18
7.2	Specificații	18
7.3	Arhitectura aplicației	18
7.4	Implementare	18
7.5	Sablone bazate pe microservicii în practică	18
8	Concluzii	19
	Bibliografie	20

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 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.”((Hilliard, 2000))

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 problema 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” ((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, 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, aceasta tranzacție (verificarea dacă suma introdusă este validă, 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 unități 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 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 în vedere numărul mare de întâlniri din interiorul firmei, aceștia doresc o aplicație în care să poată rezerva o sală 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).

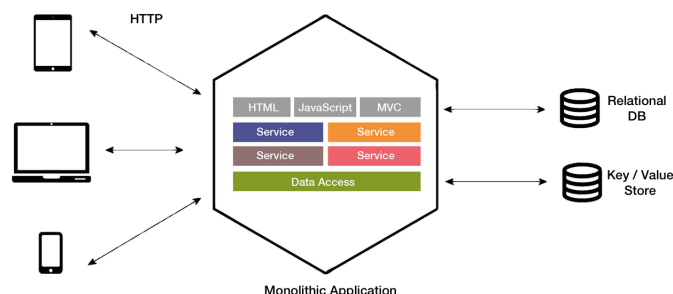


Figura 3.1: Arhitectură Aplicație Monolit(imagine : <http://bits.citrusbyte.com>)

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.
- Aplicația este ușor de testat, automatizând procesul de testare a codului folosind o combinație de unit tests 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 ca 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 în 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.((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 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 apărut.

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

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 8 principii, primele 4 sunt cele mai importate, acestea reprezentând fundația arhitecturii bazate pe servicii. Toate cele 8 principii se susțin între 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 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 ele 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.

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
- Autonomie pură

4.2.2 Cuplajul slab

4.2.3 Abstractizarea

4.2.4 Contractul Formal

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 în 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 Meshenberg
- Adrian Cockcroft
- Josh Evans

6.2.1 Arhitectura aplicației de azi

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

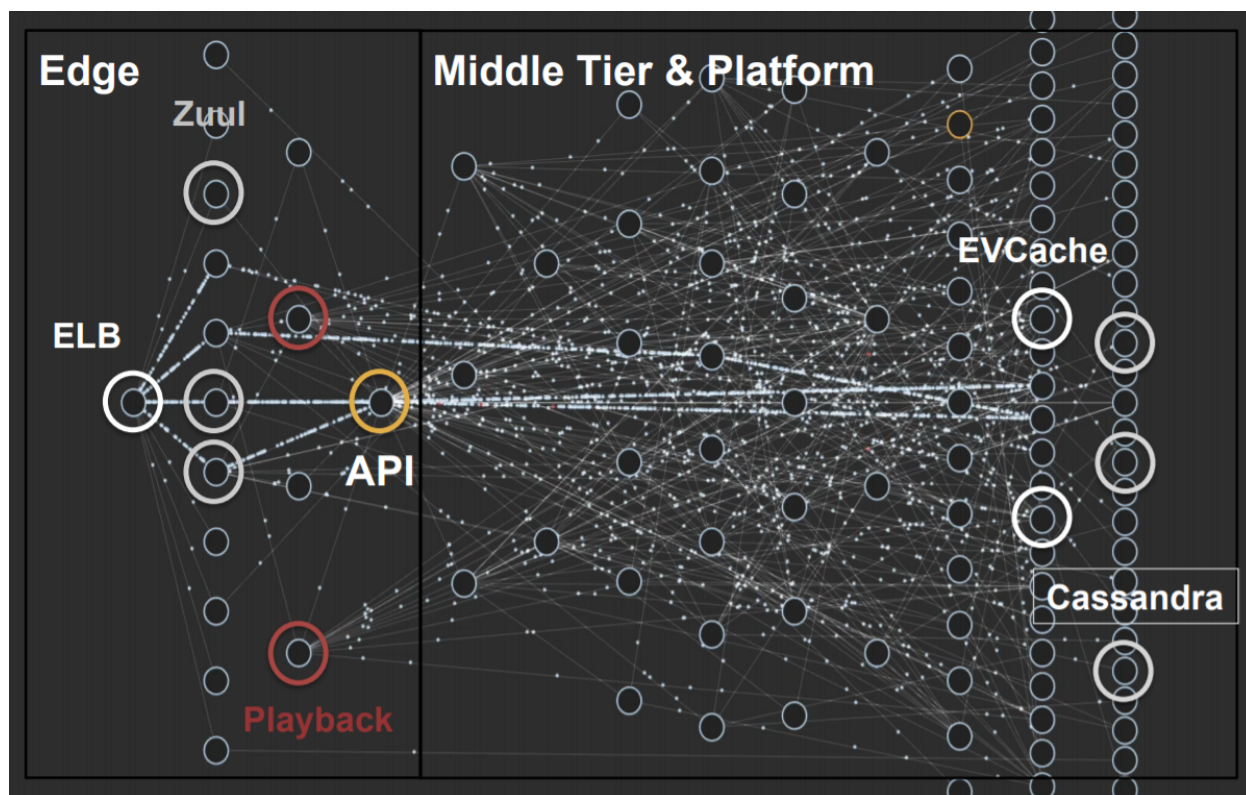


Figura 6.1: Arhitectura Aplicației Netflix()

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 clientilor. În partea a 2-a a diagramei, este un strat destul de stratificat de microservicii care conțin mai multă logică și servesc componentelor 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.

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.

500+ microservices



Figura 6.2: Arhitectura Aplicației Netflix()

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 că 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 că 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 fără stare

(în engleză „stateless”), acestea 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

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.

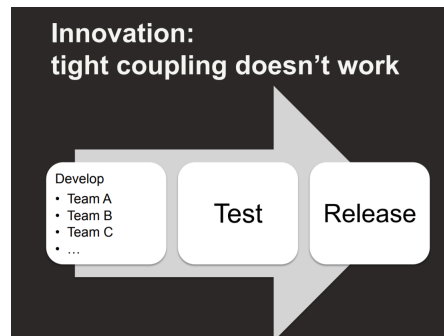


Figura 6.4: Procesul de dezvoltare într-o aplicație monolit

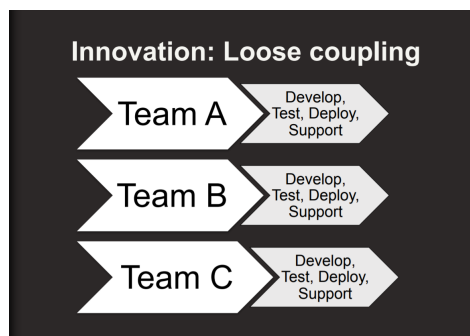


Figura 6.5: Procesul de dezvoltare într-o aplicație bazată pe microservicii

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 relațiile oamenilor din echipe. Printre acestea se pot număra următoarele:

- În primul rând, costul de dezvoltare și mentenanță a aplicației 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 noilor 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.

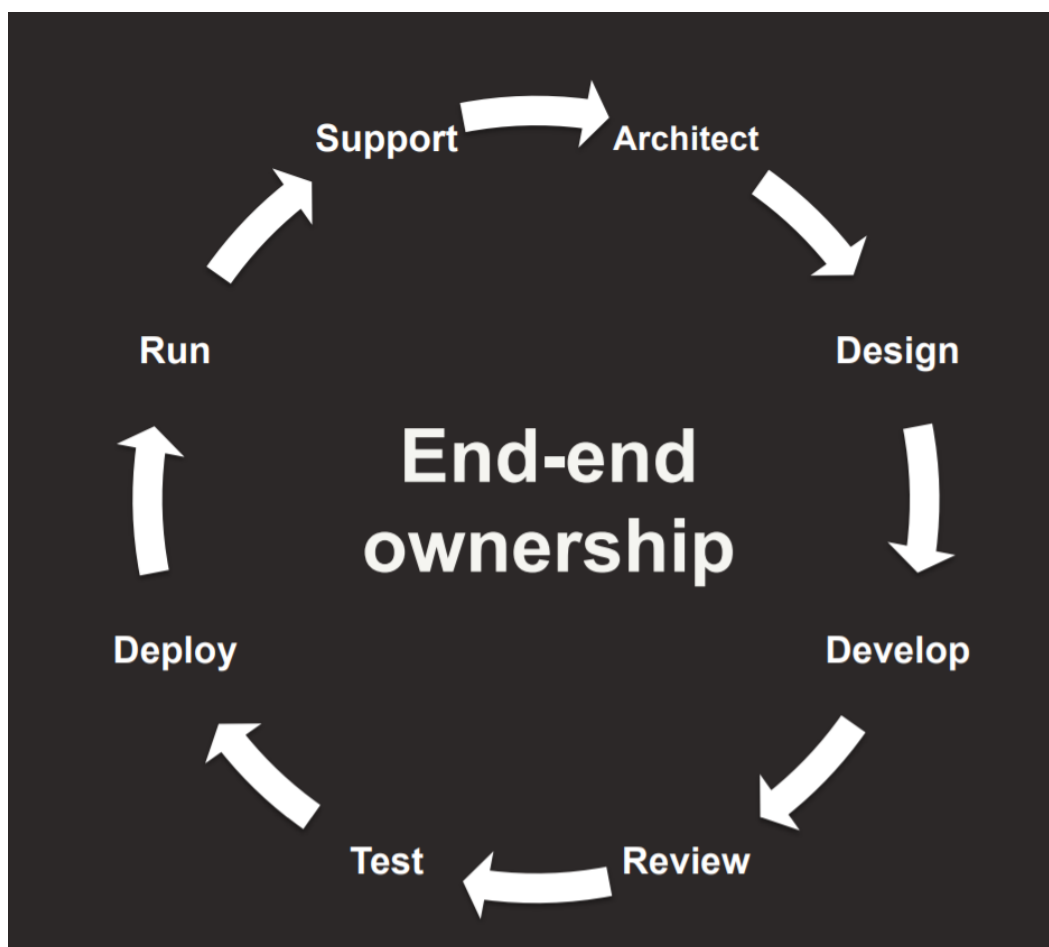


Figura 6.6: Ciclul de viață al unui microserviciu

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.

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 căutare. 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 ca este open source(în engleza „open source”), 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 în mare parte lui Adrian Cockcroft, care este un vizionar, reușind tranziția Netflix catre o arhitectura bazată pe microservicii in cloud, într-un moment în care nimeni nu dorea să creadă ca cloud-ul ar putea să fie o soluție(6.7).

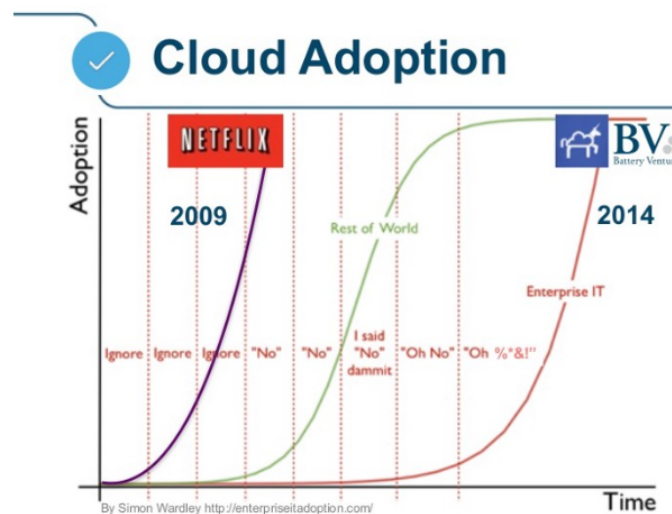


Figura 6.7: Migrarea catre cloud a aplicațiilor

Capitolul 7

Aplicație Practică

7.1 Cerința

7.2 Specificații

7.3 Arhitectura aplicației

7.4 Implementare

7.5 Sablone bazate pe microservicii în practică

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. [interview cu] Derrick Harris. 2015.
- Cockcroft, Adrian. 2014.** Migrating to Microservices by Adrian Cockcroft. 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>.