

Universitatea POLITEHNICA din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Arhitecturi pentru sisteme auto-scalabile: microservicii

Lucrare de cercetare

**Prezentată ca cerință parțială pentru obținerea
titlului de *Master***

**în domeniul *Electronică, Telecomunicații și Tehnologia Informației*
programul de studii *Tehnologii Software Avansate pentru Comunicații***

Conducător științific
Conf. Dr. Ing. Eduard Popivici

Absolvent
Andrei Mihăescu

Anul 2016

Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul *Arhitecturi pentru sisteme auto-scalabile: microservicii*, prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității “Politehnica” din București ca cerință parțială pentru obținerea titlului de *Master* în domeniul Inginerie Electronică și Telecomunicații/ Calculatoare și Tehnologia Informației, programul de studii *Tehnologii Software Avansate pentru Comunicatii* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate. Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare. Declar că toate rezultatele simulărilor, experimentelor și măsurărilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, Iunie 2016.

Absolvent: Andrei Mihăescu

.....

Cuprins

Lista figurilor	iii
Lista tabelelor	v
Lista acronimelor	vi
1. Introducere	1
1.1. Ce este o arhitectura software?	1
1.2. De ce este arhitectura importantă ?	2
1.3. Obiectivele arhitecturii software	2
1.4. Principii arhitecturale cheie	3
2. Principii fundamentale ale arhitecturii software	4
2.1. Principii de proiectare	5
2.1.1. Standarde de proiectare	6
2.1.2. Nivelele aplicației	6
2.1.3. Componente, module și funcții	7
3. Tipare și stiluri arhitecturale	8
3.1. Ce este un stil arhitectural?	8
3.1.1. Sumar al stilurilor arhitecturale	9
3.1.2. Combinarea stilurilor arhitecturale	9
3.2. Arhitectura client/server	10
3.3. Arhitectura bazată pe componente	11
3.4. Arhitectura stratificată	13
3.5. Arhitectura bazată pe bus de mesaje	15
3.6. Arhitectura orientată pe obiecte	17
3.7. Arhitectura orientată pe servicii	18
4. Microservicii	20
4.1. Caracteristici ale arhitecturii microservicii	21
4.1.1. Separarea prin intermediul serviciilor	21
4.1.2. Organizare în jurul nevoilor de business	22
4.1.3. Produse, nu proiecte	23
4.1.4. Puncte termnale inteligente și bus-uri simple	23
4.1.5. Guvernare distribuită	24
4.1.6. Managementul distribuit al datelor	24
4.1.7. Automatizarea infrastructurii	25
4.1.8. Proiectare pentru dezastru	26
4.1.9. Design evolutiv	27
4.2. Avatanje și dezavantaje ale microserviciilor	29
4.2.1. Avantaje	29

4.2.2.	Dezavantaje	29
4.3.	Construirea microserviciilor: folosirea unui API gateway	31
4.3.1.	Introducere	31
4.3.2.	Comunicare directă între client și microservicii	32
4.3.3.	Folosirea unui API Gateway	32
4.3.4.	Avantaje și dezavantaje ale folosirii unui API Gateway	33
4.3.5.	Implementarea unui API Gateway	33
4.3.5.1.	Perfomanță și scalabilitate	33
4.3.6.	Folosirea unui model de programare reactiv	34
4.3.7.	Invocarea serviciilor	34
4.3.8.	Descoperire serviciilor	34
4.3.9.	Tratarea penelor parțiale	34
4.4.	Construirea microserviciilor: Comunicarea inter-process în cadrul microserviciilor	36
4.4.1.	Introducere	36
4.4.2.	Stiluri de interacțiune	36
4.4.3.	Definirea API-urilor	38
4.4.4.	API-uri evolutive	38
4.4.5.	Gestionarea penelor parțiale de serviciu	38
4.4.6.	Tehnologii IPC	39
4.4.7.	Comunicarea asincronă bazată pe mesaje	40
4.5.	Identificarea serviciilor	41
4.5.1.	Identificarea serviciilor de către client	41
4.5.2.	Identificarea serviciilor de către server	42
4.5.3.	Registrul serviciilor	43
4.5.4.	Metode de înregistrare a serviciilor	44
4.5.4.1.	Auto-înregistrarea	44
4.5.4.2.	Înregistrarea terță	45
5.	Implementare practică	46
5.1.	Descrierea componentelor	47
5.1.1.	Microserviciile de tip depozit	48
5.1.2.	Service Registry	52
5.1.3.	API Gateway	54
5.1.4.	Baza de date	55
5.2.	Funcționarea unui caz de utilizare	56
5.3.	Șablonul circuit breaker	58
6.	Bibliografie	60

Lista figurilor

2.1. Organizarea pe "arii de responsabilitate"	4
3.1. Client/server	10
3.2. Arhitectură bazată pe componente	12
3.3. Arhitectură stratificată	13
3.4. Arhitectură bazată pe bus de mesaje	16
3.5. Arhitectură orientată pe servicii	18
4.1. Arhitecturii monolit si microservicii	20
4.2. Legea lui Conway	22
4.3. Echipe responsabile de servicii	22
4.4. Model distribuit al datelor	25
4.5. Pipeline	26
4.6. Instalarea modulelor diferă	26
4.7. Diagramă a serviciilor	31
4.8. Folosirea unui API gateway	32
4.9. Comunicare inter-process	36
4.10. Exemplu de interacțiune servicii	37
4.11. Consumarea resurselor unei aplicații	39
4.12. Identificarea serviciilor în cloud	42
4.13. Identificarea serviciilor de către client	43
4.14. Identificarea serviciilor de către server	43
4.15. Auto-înregistrare	45
5.1. Arhitectura aplicației	47
5.2. pom.xml - SpringBoot	49
5.3. Lansare instanță microserviciu	49
5.4. Lansare instanță microserviciu	49
5.5. Lansare instanță microserviciu	50
5.6. Lansare instanță microserviciu	50
5.7. Clasă de tip entitate	50
5.8. Configurarea DDL	51
5.9. Configurarea DDL	51
5.10. Configurarea DDL	51
5.11. Operațiuni legate de baza de date	52
5.12. Expunerea serviciilor prin intermediul unui REST API	52
5.13. Cod pentru server Eureka	53
5.14. Intefată web pentru server Eureka	53
5.15. API Gateway - clasa depozit	54
5.16. Livrarea aplicației front-end	55
5.17. Injectare RestTemplate	55
5.18. Fișier configurare docker-compose	56
5.19. Caz de utilizare: crearea unui utilizator	56

5.20. Pagină creare utilizator	57
5.21. Pagină comentarii - serviciu funcțional	58
5.22. Configurare Hystrix	59
5.23. Pagină comentarii - serviciu în pană	59

Lista tabelelor

3.1. Categori și stiluri arhitecturale	8
3.2. Sumar al stiluri arhitecturale	9

Lista acronimelor

BDFU - Big Design Upfront
COBRA - Common Object Request Broker Architecture
COM - Common Object Model
DCOM - Distributed Common Object Model
DDD - Domain-Driven Design
DRY - Don't repeat yourself
EJB - Enterprise Java Beans
ESB - Enterprise Service Bus
IDL - Interface Definition Language
IPC - Inter-process communication
ISB - Internet Service Bus
JIT - Just in time
LOB - Line-of-business
MVC - Model View Controller
P2P - Peer-to-Peer
QA - Quality Assurance
RPC - Remote Procedure Call
SOA - Service Oriented Architecture
UML - Unified Modelling Language
URI - Uniform Resource Identifiers
YAGNI - You ain't gonna need it

Capitolul 1

Introducere

1.1 Ce este o arhitectura software?

Arhitectura software reprezintă procesul de definire a unei soluții structurate care îndeplinește toate cerințele tehnice și operaționale, totodată optimizând metrici comune de calitate precum performanța, securitatea și facilitatea de gestionare. Aceasta presupune o serie de decizii bazate pe o gamă largă de factori fiecare din aceștia având un impact considerabil asupra calității, performanței, gestionabilității și bunei funcționării a aplicației.

Philippe Kruchten, Grady Booch, Kurt Bittner și Rich Reitman au derivat și rafinat definiția arhitecturii software bazându-se pe munca lui Mark Shaw și David Garlan (Shawn and Garlan 1996). Definiția lor este următoarea:

”Arhitectura software înglobează setul deciziilor semnificative legate de organizarea unui sistem software ce includ selectarea elementelor structurale și a interfețelor din care sistemul este compus; comportamentul așa cum reiese din interacțiunea acestor elemente; compunerea acestor elemente structurale și comportamentale în subsisteme mai mari; și un stil arhitectural care guvernează această organizare. Arhitectura software implică constrângeri și compromisuri legate de funcționalitate, utilitate, robustețe, performanță, reutilizare, inteligibilitate, economie, tehnice și estetice.”

În cartea *”Patterns of Enterprise Application Architecture”*, Martin Fowler evidențiază câteva teme recurente explicând conceptul de arhitectură. El identifică aceste teme după cum urmează: ”Descompunerea de nivel înalt a unui sistem în părți componente; deciziile care sunt dificil de schimbat; există multe arhitecturi într-un sistem; ceea ce este arhitectural important se poate schimba de-a lungul ciclului de viață al sistemului; și, la final, arhitectura se rezumă la lucrurile importante.”

În cartea *”Software Architecture in Practice (2nd edition)”* Bass, Clements, and Kazman definesc arhitectura astfel: ”Arhitectura software a unui program sau a unui sistem de calcul reprezintă structura sau structurile, ce înglobează elementele software, proprietățile lor vizibile către exterior și relația între acestea. Arhitectura se preocupă cu partea publică a interfețelor; detaliile private ale elementelor - cele ce sunt strict legate de implementarea internă - nu sunt legate de arhitectură.”

1.2 De ce este arhitectura importantă ?

Ca orice structură complexă, software-ul trebuie construit pe o bază solidă. Neluarea în considerare a anumitor scenarii cheie, cât și ignorarea anumitor probleme comune de design pot pune aplicația în pericol. Uneltele și platformele moderne ajută la construirea aplicațiilor, însă nu pot înlocui nevoia unei proiectări atente a aplicației, bazată pe scenarii și cerințe. Riscurile pe care le presupune o arhitectură slab gândită sunt instabilitatea, inabilitatea de a susține cerințele actuale și viitoare de business sau dificultatea de instalare și gestiune într-un mediu de producție.

Un sistem ar trebui proiectat luând în considerare utilizatorul, infrastructura și obiectivele de business. Pentru fiecare din aceste arii, trebuie gândite scenarii cheie, identificate proprietățile relevante și arii cheie de satisfacție și desatisfacție. Unde este posibil, este indicat să se stabilească metrici precise care vor putea fi evaluate pentru a măsura succesul fiecărei arii.

Cel mai probabil vor exista compromisuri și un echilibru trebuie găsit între cerințele concurente din aceste trei arii. De exemplu experiența utilizatorului, este adesea o funcție de business și infrastructură și schimbările într-una din zone o poate drastic afecta. În mod similar, schimbările la nivelul experienței de utilizare pot avea impact la nivelul infrastructurii și business. Performanța ar putea fi importantă din punct de vedere business și al utilizatorului, dar administratorul de sistem poate nu avea mijloacele financiare pentru a atinge obiectivele în 100% din timp. Un compromis ar fi să atingă obiectivele 80% din timp.

Rolul arhitecturii este acela de a defini modul în care elemente majore și componente din cadrul aplicației sunt folosite sau interacționează între ele. Selectarea structurilor de date și a algoritmilor, cât și detaliile de implementare specifice fiecărei componente nu sunt de interes pentru proiectare. Problemele de arhitectură și proiectare de multe ori se intercalează. În unele cazuri, deciziile țin mai mult de arhitectură. În altele, în schimb țin mai mult de proiectare și de cum acestea contribuie la realizarea arhitecturii.

1.3 Obiectivele arhitecturii software

Arhitectura software urmărește să găsească un compromis între specificațiile de business și cele tehnice înțelegând cazurile de utilizare și apoi găsind căi de implementare. Obiectivele arhitecturii sunt acelea de a găsi cerințele care influențează structura aplicației. O arhitectură bine realizată reduce riscurile de business asociate cu construirea unei soluții tehnice. Un design bun este suficient de flexibil ca să poată gestiona devierile de la tehnologiile software și hardware care pot apărea în timp, cât și cerințele utilizatorilor. Un arhitect trebuie să ia în considerare efectul global al deciziilor de proiectare, cât și compromisurile inerente între factorii de performanță, dar și cele cu privire la utilizator, infrastructură și cerințele de business.

1.4 Principii arhitecturale cheie

Pentru proiectarea unei arhitecturii următoarele principii cheie ar trebui luate în calcul:

- **Arhitectura trebuie să fie concepută pentru a suporta schimbări, nu pentru a rămâne neschimbată.** Întotdeauna trebuie luat în calcul cum aplicația s-ar putea modifica de-a lungul timpului pentru a putea răspunde noilor cerințe și provocări.
- **Modelarea trebuie făcută pentru a reduce riscul.** Este indicată folosirea uneltelor de proiectare, a sistemelor de modelare precum **UML** - *Unified Modelling Language* și a vizualizare unde este cazul pentru a putea evidenția cerințele și a deciziile arhitecturale și de proiectare, analizându-le impactul.
- **Folosirea modelelor și a uneltelor de vizualizare pentru comunicare și colaborare.** Comunicarea eficientă a designului, a deciziilor luate și schimbărilor necesare este necesară unei arhitecturi bune. Este recomandată folosirea modelelor, a vederilor și a altor mijloace de vizualizare a arhitecturii pentru a comunica și împărtăși eficient ideile cu clienții, rezultând astfel într-o comunicare rapidă a schimbărilor de proiectare.
- **Identificarea deciziilor tehnice cheie.** Este esențială înțelegerea deciziilor tehnice cheie pentru evitarea greșelilor comune. Investirea în luarea acestor decizii bine de prima dată este importantă pentru a obține un design flexibil și foarte puțin expus riscului de a fi afectat de viitoare schimbări.

Pentru rafinarea arhitecturii este recomandată o abordare incrementală și iterativă. Se pornește de la o arhitectură de bază pentru a avea o imagine de ansamblu, după care crează arhitecturii derivate pe măsură ce aceasta este testată și îmbunătățită. Modelul incipient nu trebuie să acopere toate nevoile, ci ar trebui să reprezinte un prim design pe care să poată fi testate cerințele de business. În mod iterativ vor fi adăugate detaliile care vor putea o primă imagine de ansamblu corectă, pentru ca mai târziu să fie adăugate și detaliile de finețe. O greșeală foarte des întâlnită este aceea de a se concentra pe detaliile neesențiale încă din faze incipiente și de a urma direcții greșite făcând presupuneri incorecte sau eșuând în a evalua eficient arhitectura.

Capitolul 2

Principii fundamentale ale arhitecturii software

Acest capitol va aborda principiile fundamentale de proiectare ale unei arhitecturii software. Aceasta este adesea descrisă ca organizarea sau structura unui sistem, care reprezintă o colecție de componente ce îndeplinesc o funcție sau un set de funcții specifice. Cu alte cuvinte, arhitectura se concentrează pe organizarea componentelor ce vor oferi o anumită funcționalitate. Organizarea funcțională a componentelor implică grupare acestora în "arii de interes", după cum se poate vedea în schemă de mai jos.

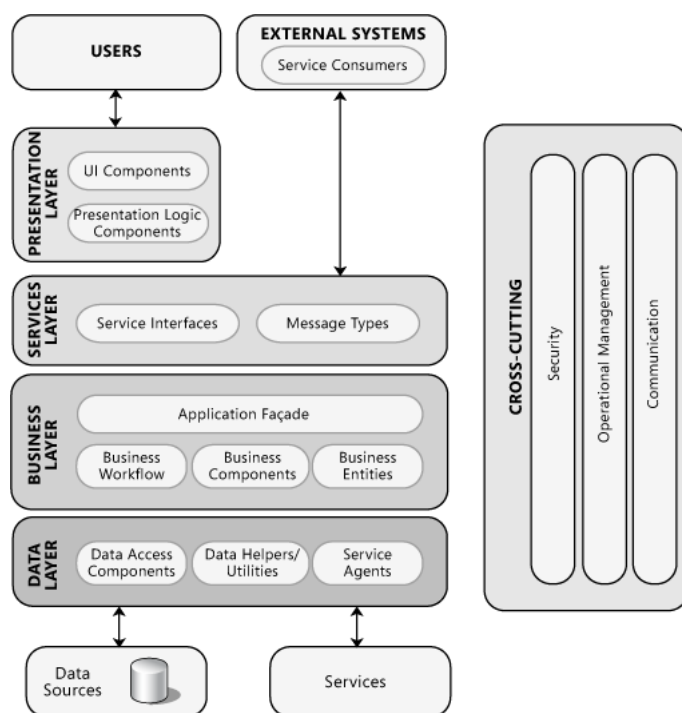


Figura 2.1: Organizarea pe "arii de responsabilitate"

Pe lângă gruparea pe componente, alte arii de interes se concentrează pe interacțiunea între acestea și pe cum ele funcționează împreună.

2.1 Principii de proiectare

La începerea procesului de proiectare trebuie avute în vedere principiile care vor contribui la crearea unei arhitecturii care aderă la practici consacrate, reduce costurile și care încurajează ușurința de folosire și posibilitățile de extindere. Acestea sunt:

- **Separarea atribuțiilor.** Aplicația trebuie divizată în funcționalități distincte cu o suprapunere cât mai mică între acestea. Factori importanți pentru a obține aceasta o reprezintă un nivel mare de coeziune și o cuplare slabă între componente, însă separarea greșită funcționalităților poate duce la o cuplare strânsă între acestea chiar dacă fiecare atribuțiile fiecărei componente nu se suprapun.
- **Principiul responsabilității unice.** Fiecare componentă sau modul ar trebui să îndeplinească o singură funcționalitate sau o agregare coerentă de funcționalități.
- **Principiul cunoașterii minime (cunoscut și sub numele de Legea lui Demeter).** O componentă sau un obiect nu ar trebui să cunoască detaliile interne de implementare ale altor componente sau obiecte.
- **Nu te repeta (eng. Don't repeat yourself - DRY)** Funcționalitatea ar trebui să fie implementată într-un singur loc. De exemplu, în cadrul proiectării unei aplicații, funcționalitate ar trebui implementată într-o singură componentă și nu ar mai trebui duplicată și în alta.
- **Minimizarea proiectării anticipate.** Proiectarea ar trebui să se rezume doar la ceea ce este necesar. În unele cazuri însă, atunci când costurile de dezvoltare sau al uni eșec în design sunt foarte mari, proiectarea anticipată este necesară. În altele, în special în cadrul dezvoltării agile, aceasta (eng. **BDUF - big design upfront**) poate fi evitată. Dacă cerințele aplicației sunt neclare sau există posibilitate unor evoluții a designului în viitorul apropiat, evitarea eforturilor de proiectare prematură este indicată. Acest principiu se mai numește și **YAGNI (eng. "You ain't gonna need it")**.

În cadrul proiectării unei aplicații sau a unui sistem, obiectivul unui arhitect este să minimizeze complexitatea separând designul în mai multe arii de responsabilitate. De exemplu, interfața grafică, procesele de business și accesul la date reprezintă arii diferite. În cadrul fiecăreia, componentele proiectate ar trebui să fie focusate pe responsabilități diferite, fără a conține cod din cadrul alteia. De exemplu, interfața grafică nu va conține cod care accesează direct sursele de date; în schimb va folosi componente specializate pentru obținerea datelor.

O evaluare cost/beneficiu va fi folosită pentru determinarea investiției necesare. În unele cazuri poate fi necesară o simplificare a structurii, permițând legarea interfeței grafice la un set de date. În general, separarea funcționalităților va fi făcută ținând cont și de partea de business. Cele ce urmează vor prezenta factori care afectează ușurința de design, implementare, testare și mentenanță a aplicației.

2.1.1 Standarde de proiectare

La nivelul fiecărui nivel logic, acolo unde este posibil, proiectare componentelor ar trebui să fie consistentă în cadrul unei operații. De exemplu, dacă se folosește tiparul "Table Data Gateway" pentru a crea un obiect care să reprezintă punctul de acces al tabelelor dintr-o bază de date, nu ar mai trebui folosit un altul (ex. "Repository") care folosește o altă paradigmă pentru accesarea datelor și inițializarea entităților de business. Însă, este posibilă folosirea altor tipare pentru operațiuni al unui alt modul care are o varietate mai largă de cerințe, cum ar fi o aplicație care necesită tranzacții business și rapoarte.

Funcționalitatea nu trebuie duplicată în cadrul aceleiași aplicații; o singură componentă va oferi această funcționalitate, care nu va mai exista într-o alta. Aceasta va duce la componente coerente și va ușura optimizarea acestora dacă o anumită funcționalitate va suferi modificări. În caz contrar, duplicarea funcționalității va îngreuna implementarea schimbărilor, scădea claritatea și va introduce potențiale inconsistențe.

Atunci când este posibil, compoziția este preferată în defavoarea moștenirii pentru re folosirea funcționalității deoarece crește dependența între părinte și clase care moștenesc. Aceasta de asemenea reduce ierarhiile de moștenire, ceea ce poate deveni dificil de gestionat.

Stabilirea unui stil de codare și a unei convenții de denumire pentru dezvoltare va oferi un model consistent care va facilita revizuirea codului de către membrii ai echipei care nu l-au scris, ceea ce duce la o administrare îmbunătățită.

Mentținerea calității sistemului se poate face folosind tehnici automatizate de QA pe parcursul dezvoltării, precum scrierea de teste unitare, analiza de dependențe și analiza statică a codului pe parcursul dezvoltării. Define clear behavioral and performance metrics for components and sub-systems, and use automated QA tools during the build process to ensure that local design or implementation decisions do not adversely affect the overall system quality.

Proiectarea componentelor aplicației și a sub-sistemelor cu o înțelegere clară a nevoilor operaționale specifice fiecăreia va ușura semnificativ instalarea și mentenanța acestora. În vederea eficientizării acestor două operațiuni metrice și date operaționale sunt necesare echipei responsabile de infrastructură. Pentru aceasta folosirea unetelor automatizate pentru asigurarea calității pot fi folosite.

2.1.2 Nivelele aplicației

Pentru a pune în practică principiul "ariilor de responsabilitate" aplicația este împărțită în blocuri funcționale a căror funcționalitate se suprapune cât mai puțin. Mare avantaj al acestei abordări este că fiecare bloc funcțional va putea fi optimizat independent față de restul aplicației. În plus, dacă unul încetează să funcționeze, acesta nu va cauza și defectarea altora, iar procesele pot rula independent unul față de celălalt. Această abordare contribuie și la facilitarea înțelegerii și proiectării aplicațiilor și simplifică administrarea sistemelor complexe interdependente.

Permițând fiecărui nivel să comunice cu celălalte sau să fie dependent de alte nivele va crește gradul de complexitate al aplicației și o va face mai greu de înțeles și administrat, așadar regulile de interacțiune între acestea trebuie foarte bine explicitate, astfel încât fluxul datelor în aplicație să fie foarte limpede.

Implementarea couplării slabe între nivele se poate implementa prin folosirea abstractizării, definindu-se astfel componente-interfața precum "façade" cu intrări și ieșiri bine cunoscute care se traduc prin cereri într-un format cunoscut înțeles de nivelul respectiv. În plus, se pot folosi tipurile Interface sau clase de bază abstracte pentru a implementa o interfață comună.

Separarea tipurilor de component în cadrul aceluiași nivel logic se poate obține prin identificarea diferitelor arii de responsabilitate și apoi prin gruparea componentelor asociate fiecăruia

pe nivele logice. De exemplu, interfața grafică nu ar trebui să conțină componente legate de procesare business, însă ar trebui să conțină elemente care permit utilizatorului să introducă date.

Amestecând formatele datelor în cadrul unui nivel sau a unei component va face ca aplicația să devină mai dificil de implementat, extins și administrat, așadar formatul trebuie să fie consistent. În caz contrar, de fiecare dată când este necesară trecerea de la un format la altul, o operație de traducere trebuie efectuată ceea ce atrage după sine efort suplimentar.

2.1.3 Componente, module și funcții

O componentă sau un obiect nu trebuie să depindă de detaliile interne ale unei alte componente sau obiect. Fiecare componentă sau obiect ar trebui să apeleze o metodă a unui alt obiect sau component, iar aceasta să aibă informații despre cum să proceseze cererea și eventual cum să o redirecționeze către altele.

Funcționalitatea unei componente nu trebuie supraîncărcată. De exemplu, o componentă a interfeței grafice nu va conține cod specific accesului datelor și nici nu va încerca să ofere o funcționalitate suplimentară. Adesea componentele supraîncărcate au multe funcții și proprietăți care furnizează funcționalități de business amestecate cu funcționalități transverse precum logare și tratarea excepțiilor. Resultatul este un design foarte predisus erorilor și dificil de menținut. Aplicarea principiului responsabilității unice și a separării ariilor de responsabilitate contribuie la evitarea acestui lucru.

Înțelegerea felului în care componentele comunică între ele presupune înțelegerea cazurilor de utilizare pentru care aplicația a fost concepută. Trebuie stabilit dacă toate componentele vor rula în cadrul aceluiași proces sau dacă comunicarea dincolo de barierele fizice sau de proces este necesară, implementând interfețe de comunicare.

Codul funcționalităților auxiliare trebuie separat de logica de business. Acest cod este cel responsabil pentru securitate, comunicare și management operațional precum logare și instrumentație. Amestecarea codului care implementează aceste funcții cu logica de business poate duce la un design dificil de extins și menținut. Schimbările acestui cod necesită implică alterarea logicii de business. Pentru a rezolva această problemă se recomandă folosirea librăriilor și tehnicilor, precum programarea orientate pe aspecte.

Componentele, modulele și funcțiile ar trebui să definească un contract sau o interfață care descrie în mod explicit utilizare și comportamentul acestora. Un contract conține o descriere explicând cum celălalte componente funcționalitățile componente, modulului sau funcției alături de comportamentul acesteia înainte de apelare, după apelare, efecte adverse ale acesteia, excepții, caracteristici de performanță și alți factori.

Capitolul 3

Tipare și stiluri arhitecturale

În acest capitol voi prezenta tipare și principii de nivel înalt des întâlnit în aplicațiile. Acestea sunt adesea numite stiluri arhitecturale și includ tipare cum ar fi client/server, arhitectura stratificată, arhitectură bazată pe componente, arhitectură cu bus de mesaje și arhitectura orientată pe obiecte (SOA). Pentru fiecare stil, voi prezenta o imagine de ansamblu, caracteristicile principale, avantajele și informații legate de ce stil se potrivește cărei aplicații. Este important de înțeles că stilurile descriu diferite aspecte ale aplicațiilor. De exemplu, unele stiluri arhitecturale descriu tipare de instalare, altele descriu probleme legate de structuri și design, iar altele descriu soluții de comunicare. Prin urmare o aplicație tipică va folosi o combinație a mai mult de un stil din cele descrise.

3.1 Ce este un stil arhitectural?

Un stil arhitectural, adesea numit și tipar arhitectural, reprezintă un set de principii care crează un cadru abstract pentru o familie de sisteme. Un stil arhitectural îmbunătățește partiționarea și încurajează reutilizarea designului oferind soluții la probleme recurente. Tiparele și stiluri arhitecturale pot fi privite ca și principii care conturează forma unei aplicații. Garlan și Shaw definesc un stil arhitectural : *”...o familie de sisteme în termenii unui tipar de organizare structurală. Mai exact, un stil arhitectural determină vocabularul de componente și conectori care pot fi utilizați în cadrul său, împreună cu un set de constrângeri care explică cum acestea pot fi combinate. Acestea pot include constrângeri legate de topologie (ex. fără cicluri). Alte constrângeri ar putea face parte din definiția stilului.”*

Înțelegerea stilurilor arhitecturale oferă mai multe avantaje. Cel mai important ar fi că oferă un limbaj comun. Oferă de asemenea posibilitate unor discuții agnostice cu privire la tehnologie, care facilitează discuții la nivel înalt incluzând tipare și principii, fără a intra în detalii. De exemplu, folosind stiluri arhitecturale, se poate discuta despre client/server versus n-niveluri. Stilurile arhitecturale pot fi organizate în funcție de aria lor cheie. Următoarea lista enumeră zonele majore de interes alături de stilul lor corespunzător.

Categorie	Stil arhitectural
Comunicare	Arhitectură orientată pe servicii, Bus de mesaje
Instalare	Client/Server, N-Niveluri
Structură	Arhitectură stratificată

Tabela 3.1: Categoriile și stiluri arhitecturale

3.1.1 Sumar al stilurilor arhitecturale

Următorul tabel enumeră stilurile arhitecturale descrise în cadrul acestei lucrări, împreună cu o scurtă descriere a fiecăruia.

Stil arhitectural	Descriere
Client/server	Împarte sistemul în două aplicații, în care clientul lansează cereri către server. În multe cazuri, serverul este o bază de date cu logică implementată în proceduri stocate.
Arhitectură bazată pe componente	Împarte aplicația în componente funcționale sau logice reutilizabile care expun interfețe de comunicare bine cunoscute.
Arhitectură stratificată	Partiționează aplicația în grupuri funcționale numite niveluri.
Bus de mesaje	Un stil arhitectural care implică folosirea unui sistem software care poate primi și trimite mesaje folosind unu sau mai multe canale de comunicare, astfel încât aplicațiile să poată interacționa fără a cunoaște detalii specifice.
N-niveluri/3-niveluri	Împarte aplicația în blocuri funcționale la fel ca arhitectura stratificată însă fiecare segment este localizat pe o mașină fizică diferită.
Obiect orientată	O paradigmă bazată pe împărțirea responsabilităților unei aplicații sau sistem în componente individuale reutilizabile și obiecte, fiecare conținând datele și comportamentul relevante.
Arhitectură orientată pe servicii (SOA)	Se referă la aplicații care expun și consumă funcționalități prin intermediul serviciilor folosind contracte și mesaje.

Tabela 3.2: Sumar al stilurilor arhitecturale

3.1.2 Combinarea stilurilor arhitecturale

Arhitectura unui sistem software nu este aproape niciodată limitată la doar un stil arhitectural, însă este adesea o combinație a mai multor stiluri care formează un sistem complet. De exemplu am putea avea un sistem cu design SOA a cărui servicii au fost dezvoltate folosind o arhitectura stratificată și una orientată pe obiecte.

O combinație de stiluri este de asemenea utilă dacă se dorește construirea unei aplicații web, unde se poate obține o separare eficientă a funcționalității folosind arhitectura stratificată. Aceasta va separa logica de afișare de logica de business și de cea de acces la date. Din motive de securitate poate fi impusă o instalare a aplicației pe 3 niveluri sau chiar pe mai multe. Nivelul de prezentare poate fi instalat în partea demilitarizată a rețelei companiei. În cadrul acestui nivel se poate folosi un tipar separat pentru prezentare, cum ar fi Model-View-Controller (MVC), pentru modelul de interacțiune. Se poate folosi și SOA pentru implementarea unei comunicații orientată pe mesaje între serverul web și serverul aplicativ.

În cazul unei aplicații desktop, clientul va trimite cereri către server. Ceea ce se pretează în acest caz este arhitectura client/server împreună cu abordarea orientată pe componente pentru

a descompune mai departe designul în module care expun interfețele adecvate de comunicare. Folosirea designului orientat pe obiecte ar îmbunătăți reutilizabilitatea, testarea și flexibilitatea.

Mulți factori pot influența alegerea stilului. Aceștia includ și capacitatea de design și implementare a organizației; capacitatea și experiența dezvoltatorilor; infrastructura și constrângerile organizaționale.

3.2 Arhitectura client/server

Arhitectura client/server descrie sistemele distribuite care implică separarea clientului și serverului și conectarea acestora prin rețea. Cea mai simplă implementare a unui sistem client/server implică un server care este accesat direct de mai mulți clienți, adesea numit stil arhitectural pe 2 niveluri.

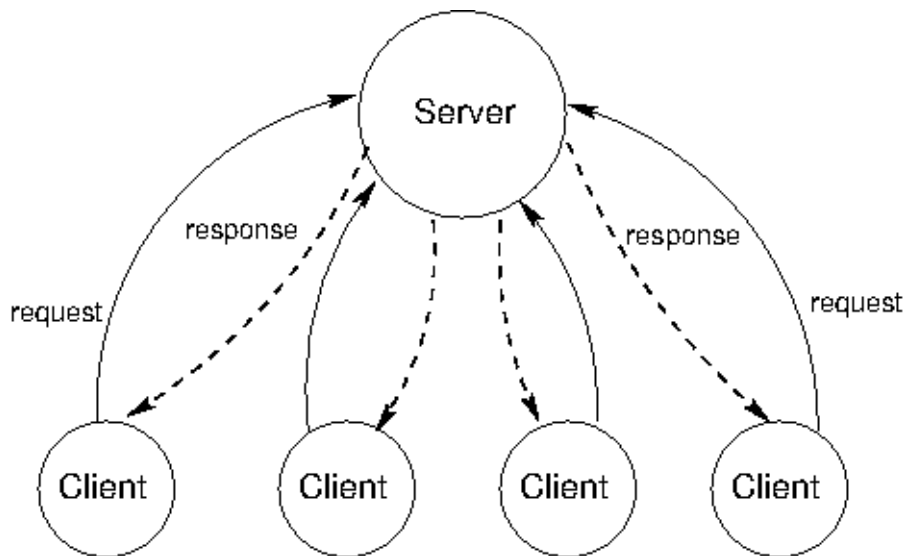


Figura 3.1: Client/server

În mod tradițional, arhitectura client/server era implementată printr-o aplicație desktop cu interfață grafică ce comunica cu un server de baze de date conținând majoritatea logicii de business în proceduri stocate. Mai generic, însă, arhitectura client/server descrie relația între un client și unul sau mai multe servere, clientul inițiind una sau mai multe cereri folosind interfața grafică, așteptând răspunsuri și executând procesările la recepție. Serverul în mod tipic autorizează utilizatorul și apoi declanșează procesele necesare pentru generarea rezultatului. Serverul poate să trimită răspunsurile folosind o varietate de protocoale și formate de date pentru a comunica informația clientului.

Astăzi exemple ale arhitecturii client/server includ programele bazate pe browserele Web ce rulează pe Internet sau intranet; aplicații ale sistemului de operare care accesează servicii de date prin intermediul rețelei; aplicații care accesează surse de date distante; unelte și utilitare pentru manipularea sistemelor la distanță.

Alte variațiuni ale stilului client/server includ:

- **Sisteme client-coadă-client.** Această abordare permite clienților să comunice cu alți clienți printr-o coadă aflată pe un server. Clienții pot citi și pot trimite date către un server care se comportă precum o simplă coadă pentru stocarea informațiilor. Asta permite clienților să distribuie și să sincronizeze fișiere și informații. Aceasta se mai numește uneori și arhitectura cozii pasive.

- Aplicații peer-to-peer (P2P). Dezvoltat plecând de la stilul precedent, P2P permite clientului și serverului să-și inverseze rolurile cu scopul de a distribui și sincroniza informații și fișiere pe mai mulți clienți. Extinde stilul client/server prin generarea mai multor răspunsuri la cereri, date partajate, descoperirea resurselor și redundanță față de pierderea unor noduri.
- Servere aplicative. Un stil arhitectural în care serverul găzduiește și execută aplicații și servicii pe care un client lejer le accesează prin intermediul unui browser sau a unui program specializat. Un astfel de exemplu este un client care execută o aplicație care rulează bazându-se pe servicii terminal.

Principalele beneficii ale acestui stil sunt:

- Securitate sporită. Toate datele sunt stocate pe server, ceea ce în general oferă un control al securității mai bun decât mașinile client.
- Acces centralizat la date. Deoarece datele sunt stocate numai pe server, accesul și actualizările datelor sunt mult mai ușor de administrat decât în orice alt stil arhitectural.
- Mentenanță facilă. Rolurile și responsabilitățile unui sistem de calcul sunt distribuite între mai multe servere care sunt cunoscute prin intermediul rețelei. Aceasta presupune transparență pentru client vizavi de posibile reparații, actualizări sau relocări ale unui server.

Acest stil se pretează foarte bine dacă aplicația ce se dorește a fi dezvoltată este: un server care va deservi mulți client, o aplicație Web, aplicația conține procese de business care vor fi utilizate în interiorul organizației sau se dorește crearea unor servicii care vor fi consumate de alte aplicații. Stilul arhitectural client/server este deasemenea adecvat atunci când se dorește centralizarea datelor, crearea redundanței și a funcțiilor de administrare sau când aplicația trebuie să deservească diverse tipuri de clienți și dispozitive.

Totuși, tradiționalul stil client/server pe 2-Niveluri are numeroase dezavantaje cum ar fi tendința de corelarea strânsă a datelor aplicației și a logicii de business pe server, ceea ce poate impacta în mod negativ scalabilitatea și dependența de serverul central, ceea ce afectează și fiabilitatea sistemului. Pentru a rezolva această problemă stilul arhitectural a evoluat într-unul ceva mai generic pe 3-Nivele (N-Nivele), descris în cele ce urmează și care reușește să înlăture dezavantajele inerente ale modelului pe 2 nivele, dar păstrează avantajele acestuia.

3.3 Arhitectura bazată pe componente

Ahitectura bazată pe componente descrie o abordare a inginerii software pentru proiectare și dezvoltarea sistemelor. Se concentrează pe descompunerea designul pe componente funcțional individuale sau logice care expun interfețe de comunicare bine definite conținând metode, evenimente și proprietăți. Aceasta crează un nivel mai înalt de abstractizare decât principiile obiect orientării și nu se axează pe probleme cum ar fi protocoale de comunicare și partajarea stărilor.

Principalele trăsături ale acestui stil sunt:

- Reutilizarea. Componentele sunt de obicei proiectate pentru a fi reutilizate în diferite scenarii în aplicații diferite. Totuși, unele sunt proiectate pentru sarcini specifice.
- Substituția. Componentele poate fi subsituite cu componente similare.

- Nespecifice contextului. Componentele sunt proiectate pentru a funcționa în medii și contexte diferite. Informații specifice, cum ar fi datele de stare, ar trebui transmise componentei în loc să fie incluse sau accesate de către aceasta.
- Extensibilitatea. O componentă poate fi extinsă pornind de la componente existente pentru a oferi noi funcționalități.
- Encapsularea. Componentele expun diferite interfețe care permit apelantului să-i acceseze funcționalitatea și care nu divulgă detalii legate de procesele sau variabilele interne.
- Independența. Componentele sunt proiectate pentru a avea dependențe minime față de alte componente. Prin urmare acestea pot fi instalate în orice mediu fără a afecta alte componente sau sisteme.

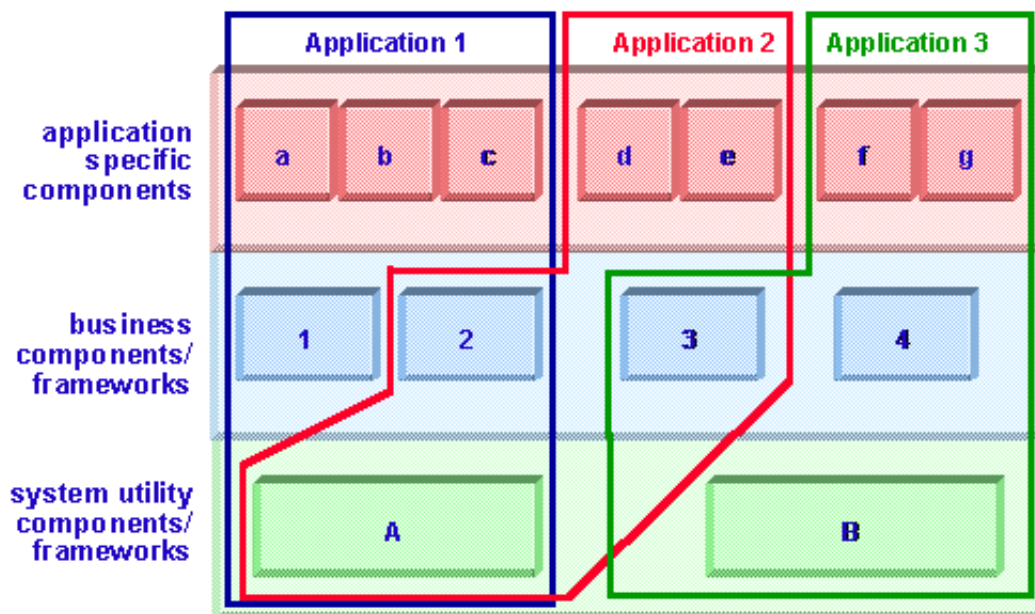


Figura 3.2: Arhitectură bazată pe componente

Tipuri comune de componente utilizate în aplicații includ componente grafice cum ar fi butoane și grile și componente ajutătoare sau utilitare care expun un set de funcții specifice folosite de către alte componente. Alte tipuri de componente sunt acelea care sunt strâns legate de resurse, nu așa de des accesate și care trebuie activate folosind abordarea *just-in-time (JIT)* - des întâlnită în cadrul componentelor comandate la distanță ; și componente bazate pe cozi ale căror apeluri de metode pot fi executate asincron folosind mesaje de așteptare.

Componentele depind de un mecanism din cadrul platformei care oferă un mediu în care ele să poată fi executate. Exemplele includ : *component object model (COM)* și *distributed component object model (DCOM)* pentru Windows; și *Common Object Request Broker Architecture (CORBA)* și *Enterprise Java Beans (EJB)* pentru alte platforme. Aceste mecanisme administrează localizarea componentelor și a interfețelor acestora, transmiterea mesajelor și a comenzilor între componente și în unele cazuri menținerea stării.

Principalele avantaje ale acestui stil sunt:

- Ușurința instalării. Pe măsură ce noi versiuni devin disponibile, acestea pot înlocui versiunile existente fără a impacta vreo altă componentă sau sistemul ca un întreg.
- Cost redus. Utilizare componentelor terțe permite reducerea costurilor de dezvoltare și administrare.

- Ușurința dezvoltării. Componentele implementează interfețe bine cunoscute pentru a oferi funcționalitate definită, permițând dezvoltarea fără a impacta alte părți ale sistemului.
- Reutilizarea. Folosirea componentelor reutilizabile înseamnă că acestea pot împărtăși costul de dezvoltare pe mai multe aplicații și sisteme.
- Atenuarea complexității tehnice. Componentele reduc complexitatea prin folosirea cadrului componentei și a serviciilor. Exemple de servicii includ activarea, gestionare timpului de viață, metode de așteptare, eveniment și tranzacții.

Tipare de proiectare cum ar fi Dependency Injection sau Service Locator pot fi utilizate pentru gestionare dependențelor între componente și pentru a încuraja slaba cuplare și reutilizarea. Aceste tipare sunt adesea folosite pentru a construi aplicații composite ce combină și reutilizează componente în cadrul mai multor aplicații.

Se dorește folosirea acestei arhitecturii dacă: există deja componente adecvate sau dacă există acces la componente terțe; aplicația va conține predominant funcții procedurale, sau poate puține date; se folosesc mai multe limbaje de programare. De asemenea, acest stil se pretează a fi folosit în cazul în care se dorește crearea unei arhitecturii modulare sau composite care să permită schimbarea și actualizarea facilă a componentelor individuale.

3.4 Arhitectura stratificată

Arhitectura stratificată se concentrează pe gruparea funcționalităților conexe din cadrul unei aplicații pe niveluri diferite care sunt grupate orizontal unele deasupra celorlalte. Funcționalitate în cadrul fiecărui nivel este dominată de un rol sau o responsabilitate comună. Comunicarea între niveluri este explicită și slab corelată. Stratificarea corectă ajută la separarea responsabilității care sporește flexibilitate și ușurința administrării.

Acest stil a fost descris ca fiind o piramidă întoarsă în care fiecare nivel agregă responsabilitățile și abstractizările nivelului imediat inferior. Folosind stratificarea strictă, componentele din cadrul unui nivel pot interacționa doar cu componente din cadrul aceluiași nivel sau cu componente din nivelul imediat inferior. O stratificare mai lejeră permite componentelor dintr-un nivel să interacționeze cu componente din același nivel sau orice componente aparținând nivelurilor inferioare.

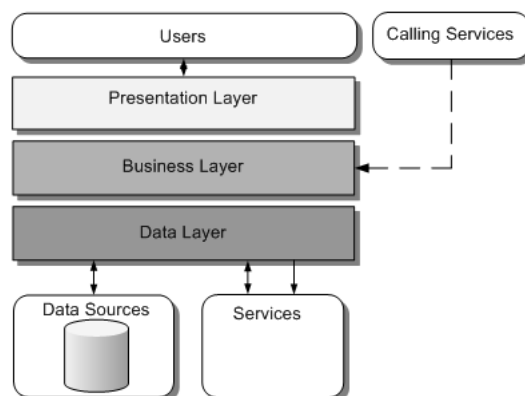


Figura 3.3: Arhitectură stratificată

Straturile aplicației pot fi aceeași mașină fizică (pe același nivel) sau pot fi distribuite pe mașini diferite (N-Nivele) și componentele din același nivel comunica cu componente din alte nivele prin interfețe bine definite. De exemplu o aplicație web tipică este alcătuită dintr-un

nivel de prezentare (tot ceea ce ține de interfața grafică), un nivel de logică business și un nivel de acces de date.

Principalele trăsături ale acestui stil includ:

- **Abstractizarea.** Arhitectura stratificată abstractizează imaginea de ansamblu a sistemului ca și întreg în vreme ce oferă destule detalii pentru a înțelege rolurile și responsabilitățile fiecărui nivel individual și relația între ele.
- **Encapsularea.** Nici o presupune nu trebuie făcută cu privire la tipurile de date, metode și proprietăți sau implementare în timpul proiectării, deoarece acestea nu vor fi expuse public.
- **Nivele funcționale clar definite.** Separarea funcționalității pe fiecare nivel este clară. Nivele superioare cum ar fi cel de prezentare trimit comenzi celor inferioare, cum ar fi cele de business și acces de date și poate reacționa la evenimente din aceste nivele, permițând fluxuri de date în ambele sensuri.
- **Nivel ridicat de coeziune.** Limitele bine definite ale responsabilității fiecărui nivel și asigurarea faptului ca fiecare nivel conține funcționalitatea strict legată de sarcinile acelui nivel contribuie la maximizare nivelului de coeziune din acel nivel.
- **Reutilizarea.** Nivele inferioare nu depinde de cele superioare, permițând o posibilă reutilizare a acestora în cadrul altor scenarii.
- **Cuplare slabă.** Comunicarea între nivele este bazată pe abstractizare și evenimente pentru a oferi o cuplare slabă între acestea.

Exemple de aplicații stratificate includ aplicații de tip line-of-business (LOB) precum sisteme de contabilitate și administrarea clienților; aplicații web de tip enterprise și site-uri web și aplicații desktop sau agenți inteligenți cu un server central pentru logica de business.

Tiparele de proiectare care susțin acest stil arhitectural sunt numeroase. De exemplu tiparele "Separated Presentation" conțin o varietate largă de tipare pentru manipularea interacțiunilor cu interfața grafică, logica de prezentare și business și datele aplicației cu care utilizatorul lucrează. Acesta permite designerilor să lucreze la interfața grafică în vreme ce dezvoltatorii lucrează la codul care va orchestra totul. Împărțind funcționalitatea astfel crește șansele de a test comportamentul individual al rolurilor. Principiile cheie ale acestui tipar sunt :

- **Separare responsabilității.** Acest tipar împarte procesare interfeței grafice în roluri distincte; de exemplu MVC are 3 roluri diferite: modelul, view-ul și controller-ul. Modelul reprezintă datele; View-ul reprezintă interfața grafică; Controller manipulează cererile, modelul și efectuează alte operații.
- **Notificări bazate pe evenimente.** Tiparul observator este folosit pentru a trimite notificări View-ului când datele administrate de model se schimbă.
- **Procesare delegată a evenimentelor.** Controller-ul procesează evenimente lansate din interfața grafică.

Principalele beneficii ale acestui stil și tipar sunt :

- **Abstractizarea.** Nivelele permit schimbări la nivel abstract. Se poate crește sau scădea nivelul de abstractizare al fiecărui nivel din stivă.

- Izolarea. Permite izolarea actualizărilor la nivele individuale pentru a reduce riscul și a minimiza impactul asupra sistemului ca și întreg.
- Administrarea. Separarea responsabilităților cheie ajută la identificarea dependențelor și organizează codul în blocuri mult mai gestionabile.
- Performanța. Distribuirea nivelelor pe mai multe mașini fizice poate crește scalabilitatea, fiabilitatea și performanța.
- Reutilizarea. Rolurile promovează reutilizarea. De exemplu, în MVC, Controller-ul poate adesea fi refolosit de către alte View-uri pentru a crea un rol specific sau un view personalizat bazat pe aceeași funcționalitate și date.
- Testarea. Posibilitatea sporită de testare rezultă din interfețe bine definite, cât și din posibilitatea de a trece de la o implementare la alta a interfeței nivelului. Tiparele de tip "Separated Presentation" permit construirea obiectelor de test pentru a simula comportamentul obiectelor concrete cum ar fi modelul, controller-ul sau view-ul în timpul testării.

Acest stil este adecvat cazului în care există nivele care poate fi reutilizate în alte aplicații, dacă există deja aplicații care expun procese de business prin interfețe de serviciu sau dacă aplicația este complexă și design-ul de nivel înalt impune o separare astfel încât echipele să se poată concentra pe arii diferite. Acest stil este potrivit și în cazul în care aplicația trebuie să fie disponibilă pe tipuri diferite de clienți sau dispozitive sau dacă se dorește implementarea unor reguli și procese business complexe și configurabile.

Tiparul prezentat mai sus sporește posibilitatea de testare a aplicației și simplifică mentenanța funcționalității interfeței grafice și oferă posibilitatea separării sarcinilor de concepere a interfeței grafice de dezvoltarea logicii de business.

3.5 Arhitectura bazată pe bus de mesaje

Arhitectura bazată pe bus de mesaje descrie principiul folosirii unui sistem software care primește și trimite mesaje folosind unul sau mai multe canale de comunicare, astfel încât aplicațiile pot interacționa fără a fi nevoite să cunoască detalii. Este un stil de proiectare a aplicațiilor a caror interacțiune se realizează prin transmitere de mesaje (de obicei asincron) prin intermediul unui bus comun. Implementările tipice ale acestui stil folosesc fie un ruter de mesaje fie tiparul "Publish/Subscribe" și sunt adesea implementate folosind sisteme de mesagerie de tip "cozi de mesaje". Multe implementări sunt alcătuite din aplicații individuale care comunică folosind o schemă comună și o infrastructură partajată pentru primirea și trimiterea de mesaje. Un bus de mesaje oferă posibilitatea de a gestiona :

- Comunicații orientate pe mesaje. Toate comunicațiile între aplicații se bazează pe mesaje care folosesc scheme cunoscute.
- Logică complexă de procesare. Operațiile complexe poate fi executate folosind un set de operațiuni mai mici, fiecare realizează o sarcină specifică, ca parte a unui proces cu mai multe etape.
- Modificări ale logicii de procesare. Deoarece interacțiunea cu bus-ul se bazează pe scheme și comenzi comune, se pot insera și scoate aplicații din bus pentru a schimba logica care este folosită pentru a procesa mesajele.

- Integrarea cu diverse medii. Folosind comunicarea orientată pe mesaje bazate pe standarde comune se pot crea interacțiuni între medii diferite cum ar fi Microsoft .Net și Java.

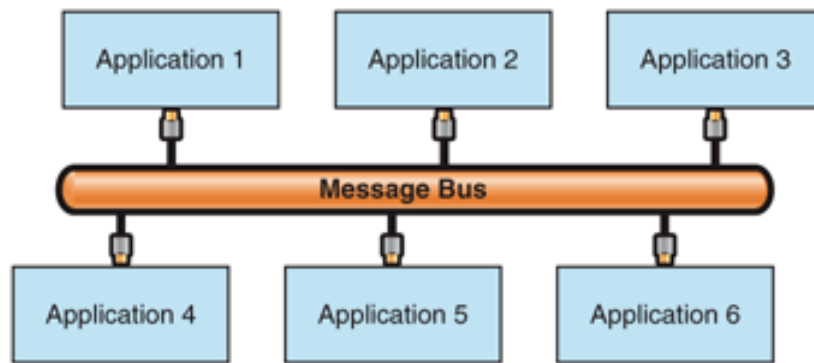


Figura 3.4: Arhitectură bazată pe bus de mesaje

Arhitecturile bazate pe bus de mesaje au fost folosite în cadrul procesărilor complexe. Acest design oferă o arhitectură ce permite inserarea aplicațiilor în proces și îmbunătățește scalabilitatea prin posibilitatea de a atașa numeroase instanțe ale aceleiași aplicații în bus. Variații ale acestui stil sunt:

- Enterprise Service Bus (ESB). Bazată pe design cu bus de mesaje, ESB folosește servicii pentru comunicarea între bus și componentele atașate la bus. De obicei oferă servicii pentru convertirea mesajelor între diferite formate, permițând clienților să folosească mesaje incompatibile pentru a comunica între ei.
- Internet Service Bus (ISB). Similar cu precedentul doar că în acest caz aplicațiile se află în cloud și nu pe rețeaua companiei. Un concept esențial al ISB reprezintă folosirea URI-urilor (Uniform Resource Identifiers) și a politicilor pentru controlarea rutării informației din cadrul proceselor aplicațiilor și serviciilor din cloud.

Principalele avantaje ale acestui stil sunt:

- Extensibilitate. Aplicațiile pot fi adăugate sau înlăturate de pe bus fără a avea impact asupra aplicațiilor existente.
- Complexitate redusă. Complexitatea aplicațiilor este redusă, deoarece fiecare aplicație nu trebuie să cunoscă decât modul de interacțiune cu busul.
- Flexibilitate. Setul de aplicații care alcătuiesc un proces complex sau tiparele de comunicare între aplicații, pot fi ușor schimbate pentru a răspunde nevoilor de business, pur și simplu prin modificarea configurării sau a parametrilor care controlează rutarea.
- Cuplare slabă. Atât timp cât aplicațiile expun o interfață adecvată pentru comunicarea cu busul de mesaje, nu există nici o dependență față de aplicația în sine, aceasta permițând schimbări, actualizări și înlocuiri cu aplicații ce expun aceeași interfață.
- Scalabilitate. Mai multe instanțe ale aceleiași aplicații pot fi atașate busului pentru a prelucra mai multe cereri în același timp.
- Simplitatea aplicației. Deși implementarea unui bus de mesaje adaugă complexitate infrastructurii, fiecare aplicație nu trebuie să asigure decât o singură conexiune la bus în locul mai multora, către celelalte aplicații.

Acest stil arhitectural este potrivit cazurilor în care deja există aplicații care funcționează împreună pentru a realiza anumite funcții sau care combină mai multe sarcini într-o singură operație. Acest stil este adecvat dacă se dorește implementarea unei sarcini care necesită interacțiune cu aplicații externe sau aplicații găzduite pe medii diverse.

3.6 Arhitectura orientată pe obiecte

Arhitectura orientată pe obiecte este o paradigmă bazată pe divizarea responsabilităților dintr-o aplicație sau sistem pe obiecte individuale reutilizabile, fiecare conținând datele și comportamentul relevant obiectului. Un design orientat pe obiecte privește un sistem ca o serie de obiecte cooperând, în loc de un set de rutine sau instrucțiuni procedurale. Obiectele sunt discrete, independente și slab coupleate; ele comunică prin interfețe, apelând metode sau accesând proprietăți ale obiectelor, sau trimițând și primind mesaje. Principiile cheie ale stilului arhitectural obiect orientat sunt următoarele:

- **Abstractizare.** Aceasta permite reducerea unei operațiuni complexe într-o generalizare ce reține caracteristicile de bază ale acesteia. De exemplu, o interfață abstractă poate fi o definiție bine cunoscută ce expune operațiuni legate de accesul la date folosind metode simple cum ar fi "get" sau "update". O altă formă de abstractizare ar putea fi metadatele folosite pentru a crea o mapare între două formate ce conțin date structurate.
- **Compoziție.** Obiectele pot asamblate pornind de la alte obiecte și pot ascunde obiectele interne din alte clase sau să le expună ca simple interfețe.
- **Moștenire.** Obiectele pot moșteni alte obiecte și folosi funcționalitatea din obiectul de bază sau să o suprascrie pentru a implementa un nou comportament. În plus, moștenirea facilitează mentenanța și actualizările, deoarece schimbările în obiectul de bază propagă schimbările și în obiectele care îl moștenesc.
- **Encapsularea.** Obiectele expun funcționalitate numai prin metode, proprietăți și evenimente, dar ascund detalii interne cum ar fi starea sau variabile din alte obiecte. Aceasta ușurează actualizarea și înlocuirea obiectelor, atât timp cât interfețele lor sunt compatibile, fără a afecta celălalte obiecte.
- **Polimorfism.** Aceasta permite suprascrierea comportamentului din tipul de bază care susține operații din cadrul aplicației implementând tipuri de noi care sunt interschimbabile cu obiectul existent.
- **Decuplare.** Obiectele pot fi decuplate de la consumator prin definirea unui interfețe abstracte pe care obiectele o pot implementa și pe care consumatorul o poate înțelege. Aceasta permite oferirea implementărilor alternative fără a afecta consumatorilor interfeței.

Utilizări tipice ale acestui stil includ definirea unui obiect model care realizează operații științifice complexe sau financiare, și a obiectelor care reprezintă artefacte din lumea reală dintr-un domeniu de business.

Beneficiile majore ale stilului orientat pe obiecte ar fi:

- **Inteligibilitatea.** Mapează aplicația la obiecte mult mai apropiate de lumea reală, ceea ce o face mai inteligibilă.
- **Reutilizarea.** Oferă posibilitatea reutilizării codului prin intermediul polimorfismului și a abstractizării.

- Testarea. Posibilități sporite de testare prin intermediul encapsulării.
- Extensibilitatea. Folosirea encapsulării, a polimorfismului și a abstractizării izolează interfețele pe care obiectul le expun de schimbările asupra datelor.
- Nivel înalt de coeziune. Punând în obiect doar metode și funcționalități corelate și folosind diferite obiecte pentru diferite funcții se obține un nivel înalt de coeziune.

Acest stil se potrivește în cazul în care se dorește modelarea unui aplicații bazat pe obiecte și acțiuni din lumea reală, sau dacă există deja obiectele și acțiunile care să se potrivească cu cerințele de design și operaționale. El mai este adecvat și în cazul în care logica de business trebuie encapsulată alături de date în componente reutilizabile sau există logică de business complexă ce necesită abstractizare și comportament dinamic.

3.7 Arhitectura orientată pe servicii

Arhitectura orientată pe servicii crează posibilitatea expunerii funcționalității unei aplicații ca un set de servicii și crearea de aplicații care să folosească aceste servicii. Serviciile sunt slab cuplate deoarece ele folosesc interfețe standard ce pot fi invocate, publicate și descoperite. Serviciile din cadrul SOA (Service Orientated Architecture) se concentrează pe oferirea unei interacțiuni bazate pe scheme și mesaje cu o aplicație prin interfețe care au scopul restrâns la aceasta și care nu sunt orientate pe componente sau obiecte. Un serviciu care respectă SOA nu ar trebuie să fie considerat precum o componentă a unui furnizor de servicii.

Stilul SOA poate să includă procese de business în servicii interoperabile folosind o gamă largă de protocoale și formate de date pentru comunicarea informațiilor. Clienții și alte servicii pot accesa servicii local rulând la același nivel, sau pot accesa serviciile la distanță prin intermediul rețelei.

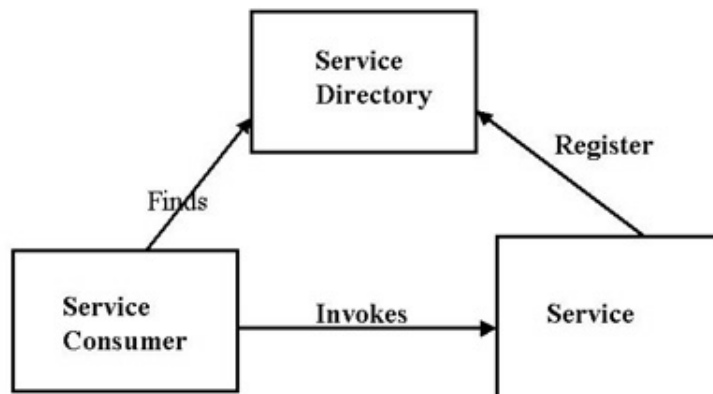


Figura 3.5: Arhitectură orientată pe servicii

Principiile cheie ale stilului SOA sunt:

- Serviciile sunt autonome. Fiecare serviciu este menținut, dezvoltat, instalat și versionat în mod independent.
- Serviciile pot fi distribuite. Serviciile pot fi localizate oriunde pe rețea, local sau la distanță, atât timp cât rețeaua dispune de protocoalele necesare de comunicare.

- Serviciile sunt slab cuplate. Fiecare serviciu este independent față de celălalte și poate fi înlocuit și actualizat fără a întrerupe celălalte aplicații care îl folosesc atât timp cât interfețele sunt compatibile.
- Serviciile împarte schema și contractul, nu clasa. Serviciile partajează contracte și scheme atunci când comunică, și nu clase interne.
- Compatibilitatea se bazează pe politici. Politici în acest caz înseamnă definirea caracteristicilor precum transport, protocol și securitate.

Exemple tipice de aplicații orientate pe servicii includ cele legate de partajarea informației, manipularea proceselor multi-etapă precum sisteme de rezervare sau magazine online, ce expun date sau servicii specifice industriei prin intermediul unei rețele, creând colecții de informații din mai multe surse.

Beneficiile principale ale stilului SOA sunt:

- Alinierea la domeniu. Reutilizarea serviciilor comune cu interfețe standard crește oportunitățile de business și tehnologice și reduce costurile.
- Abstractizarea. Serviciile sunt autonome și accesate printr-un contract formal, ce oferă cuplare slabă și abstactizare.
- Detectabilitatea. Serviciile pot expune descrieri care să permită altor aplicații și servicii să le localizeze și să construiască automat interfața.
- Interoperabilitatea. Deoarece protocoalele și formatele datelor sunt baze pe standarde din industrie, furnizorul și consumatorul serviciului pot fi construiți și instalați pe platforme diferite.
- Raționalizarea. Serviciile pot fi granulare pentru a oferi o funcționalitate specifică, mai degrabă decât să dublice funcționalitatea în cadrul mai multor aplicații.

Acest stil merită luat în considerare atunci când este disponibil accesul la serviciile care se doresc a fi reutilizate; când serviciile pot fi cumpărate de la o companii terțe; când se dorește construirea aplicațiilor compuse dintr-o serie de servicii cu o singură interfață grafică; sau când se dorește crearea unei aplicații de tipul Software plus Services (S+S), Software as a Service (SaaS), sau bazate pe cloud. Stilul SOA este adecvat atunci trebuie asigurată comunicarea între segmentele unei aplicații și expuse funcționalități într-o manieră independentă de platformă, când se dorește utilizarea serviciilor centralizate precum autentificare sau expunerea serviciilor detectabile prin registre sau care pot fi utilizate de clienți care nu cunosc în prealabil interfețele.

Capitolul 4

Microservicii

Stilul arhitectural ”micro servicii” reprezintă o abordare ce constă în dezvoltarea unei aplicații ca o suită de mici servicii, fiecare rulând un proces independent și comunicând prin mecanisme simple, adesea resurse expuse prin API-uri via HTTP. Aceste servicii sunt construite în jurul nevoilor de business și instalate în mod automat. Există un minim necesar pentru o administrare centralizată a acestor servicii, care pot fi scrise folosind limbaje de programare și soluții de stocare diferite.

Pentru a explica mai bine microserviciile este utilă comparația cu stilul monolit: o aplicație construită ca o singură unitate. Aplicațiile de tip enterprise sunt adesea construite din trei părți: o interfață grafică către client (compusă din pagini HTML și javascript ce rulează în browser-ul utilizatorului), o bază de date (constituită din mai multe tabele dintr-un sistem de gestiune al bazei de date) și o parte de server. Aplicația server se ocupă de cererile HTTP, execută logica de business, extrage și actualizează datele din baza de date, selectează și populează paginile HTML care vor fi trimise browser-ului. Partea server este un *monolit* - un singur executabil logic. Orice schimbare în sistem presupune construirea și instalarea unei noi versiune a aplicației.

O astfel de aplicație monolit este calea firească pentru construirea unui sistem. Toată logica pentru manipularea cererilor rulează în cadrul unui singur proces, permițând prin intermediul limbajului de programare împărțirea aplicației în clase, funcții și namespace-uri. Aceasta poate fi rulată și testată pe laptopul unui dezvoltator și instalată pe un mediu de producție folosind procese automatizate care să permită testarea acestora. Aceasta poate fi scalată pe orizontală rulând mai multe instanțe în spatele unui load-balancer.

Aplicațiile monolit pot fi un real succes, însă un număr tot mai mare de oameni le constată limitările, odată cu creșterea numărului de servicii oferite în cloud. Ciclurile de schimbări sunt strâns legate - o schimbare făcută unei mici părți a aplicației, necesită recompilarea și instalarea întregului monolit. În timp păstrarea unei modularități bune devine dificilă, împiedicând astfel izolarea modificărilor. Scalarea implică scalarea întregii aplicații, nu a părților care necesită resurse în plus.

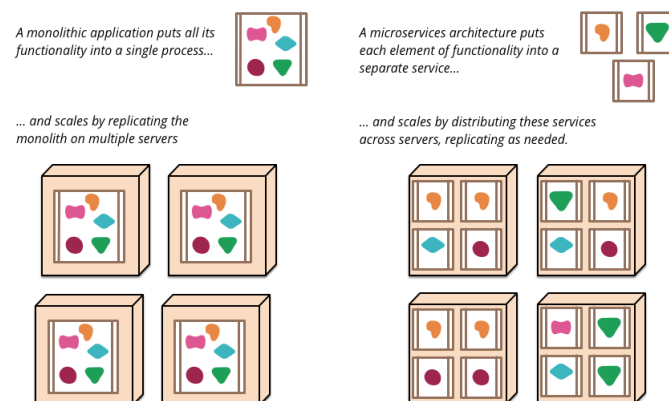


Figura 4.1: Arhitecturii monolit si microservicii

Aceste considerente au dus la arhitectura microservicii: construirea aplicațiilor ca o suită

de servicii. Pe lângă faptul ca serviciile pot fi instalate și scalate în mod independent, fiecare crează o limită fermă a modulului, permițând chiar și folosirea limbajelor de programare diferite, acestea putând fi gestionate de echipe diferite.

Acest stil arhitectural nu este o noutate, având rădăcini în principiile de proiectare ale Unix-ului, însă nu este îndeajuns de răspândit.

4.1 Caracteristici ale arhitecturii microservicii

Deși nu există o definiție formală a stilului arhitectural microservicii, însă se poate face o grupare a trăsăturilor comune arhitecturilor care se încadrează în acest tipar. Ca în cazul oricărei definiție care enumeră o serie de trăsături comune, aceste nu se vor regăsi în toate arhitecturile ce folosesc microservicii, dar este de așteptat ca majoritatea arhitecturilor să prezinte o mare parte a acestora.

4.1.1 Separarea prin intermediul serviciilor

De-a lungul industriei software, a existat întotdeauna dorința de a construi sisteme prin conectarea diverselor componente, așa cum se întâmplă în lumea reală. În ultimele decenii a existat un progres substanțial în domeniul dezvoltării bibliotecilor reutilizabile. Când vorbim despre componente ne lovim de dificultatea de a defini ce este o componentă. Definiția pe care o putem da la acest moment este că o **componentă** este o parte software care poate fi înlocuită și actualizată în mod independent.

Arhitecturile bazate pe microservicii folosesc librării, însă modalitatea principală prin care se face componentizarea este împărțirea pe servicii. Definim **bibliotecile** ca fiind componente ce funcționează împreună în cadrul unui program și deci au memorie comună, în vreme ce **serviciile** sunt componente ce rulează în cadrul unor procese diferite și care comunică prin mecanisme cum ar fi cereri de tip web sau **RPC** (Remote Procedure Call).

Unul din motivele pentru folosirea serviciilor ca și componente (mai degrabă decât biblioteci) este faptul că serviciile pot fi instalate în mod independent. În cazul unei aplicații constituită din mai multe biblioteci aparținând unui singur proces, o schimbare a oricărei component ar rezulta în reinstalarea întregii aplicații. Dacă însă aplicația este descompusă în mai multe servicii, este de așteptat ca modificările pe servicii să nu necesite decât reinstalări ale unui singur serviciu. Acest lucru nu este universal valabil, unele modificări ar putea impacta interfețe ale serviciilor necesitând o sincronizarea la nivelul mai multor servicii, însă ținta unei arhitecturi bazate pe microservicii este minimizarea acestui impact prin precizarea funcționalităților în cadrul contractelor.

O altă consecință a folosirii serviciilor ca și componente este o intefățare mult mai explicită. Majoritatea limbajelor nu au un mecanism bun de definire a unei intefete publice. Adesea, este doar documentația și disciplina utilizatorului care previn distrugerea încapsulării unei componente. Serviciile simplifică acest lucru folosind mecanisme explicite pentru apeluri la distanță.

Folosirea serviciilor în acest fel are și dezavantaje. Apelurile la distanță sunt mult mai costisitoare decât apleurile efectuate în cadrul aceluiasi proces și de aceea procedurile trebuie să fie mai puțin granulare, ceea ce implică și o dificultate de folosire. Schimbarea responsabilităților dintre componente este mai dificilă de făcut atunci când trebuie făcut trans-proces.

La o primă aproximare, se poate observa ca serviciile se mapează pe procese locale, dar această nu este decât o primă aproximare. Un serviciu poate fi constituit din mai multe procese care vor fi întotdeauna dezvoltate și instalate împreună, ca un proces de aplicație sau o o bază de date care va fi folosită doar de acel serviciu.

4.1.2 Organizare în jurul nevoilor de business

Când se dorește împărțirea unei aplicații în componente, adesea divizarea echipei se face în funcție de nivelul tehnologic, astfel formându-se echipe responsabile pentru interfața grafică, logica de server și baza de date. Când această împărțire este realizată, chiar și cele mai simple schimbări pot duce la proiecte realizate de mai multe echipe ceea ce consumă timp și bani. În 1967, Melvyn Conway afirma : ”Orice organizație ce proiectează un sistem va produce un design care este o copie a structurii de comunicare a organizației”.

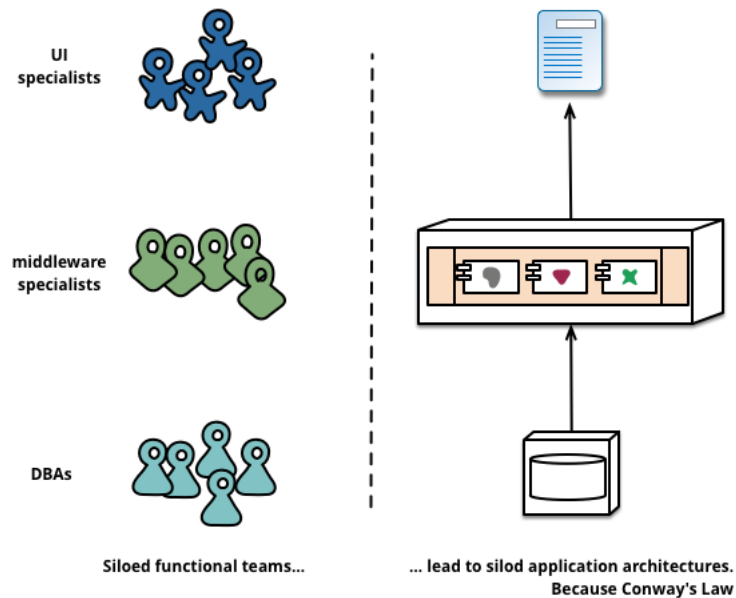


Figura 4.2: Legea lui Conway

Abordarea separării componentelor diferă în cadrul microserviciilor, aceasta făcându-se ținând cont nevoile de business. Astfel de servicii preiau întreaga stivă tehnologică ce include interfața grafică, stocarea și orice colaborare externă. În consecință echipele sunt transfuncționale, cuprinzând o plajă largă de cunoștințe necesare pentru dezvoltare: proiectarea interfeței grafice, baze de date și project management.

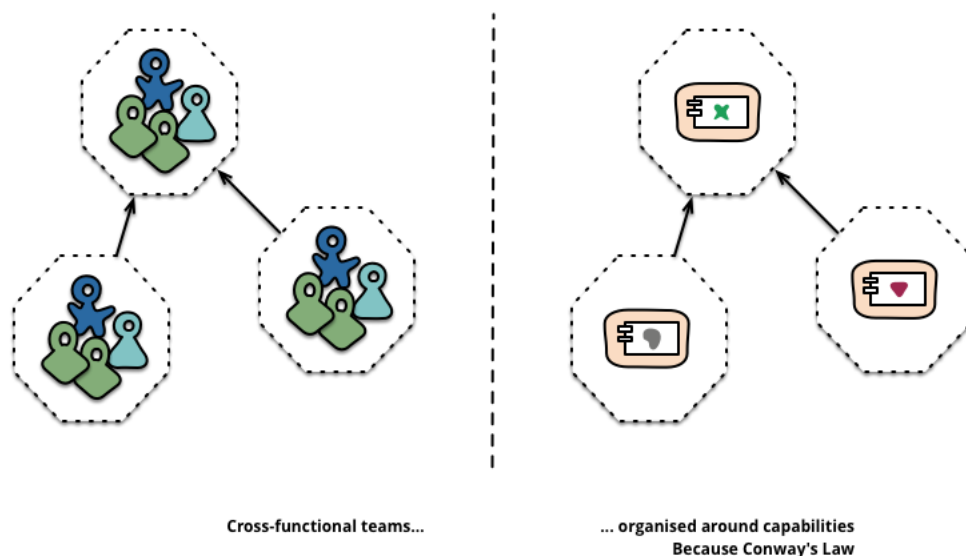


Figura 4.3: Echipe responsabile de servicii

Aplicațiile mari monolit pot fi întotdeauna modularizate în jurul nevoilor de business, însă aceasta nu se întâmplă foarte des. Cu siguranță este de dorit ca o echipă mare ce construiește o aplicație monolit să fie împărțită în echipe mai mici concentrare pe liniile de business. Una din probleme care pot apărea aici este împărțirea liniilor de business în funcție de prea multe criterii. Dacă monolitul se extinde peste prea multe module poate deveni dificil ca membrii echipei să-și însușească aplicația. În plus se observă ca modularitatea implică un nivel mai mare de disciplină. Separarea mult mai explicită necesară componentelor bazate pe servicii sporește claritatea limitelor responsabilității echipei.

4.1.3 Produse, nu proiecte

Majoritatea eforturilor de dezvoltare pe care le vedem în cadrul unui proiect sunt concentrate pe livrarea unui program care tinde să fie perceput ca fiind complet. Odata cu terminarea dezvoltării aplicația este predată unei echipe responsabile pentru mentenanță și echipa de proiect este desființată.

Susținătorii microserviciilor tind să evite acest model, preferând în schimb noțiunea ca o echipă să fie responsabil de produs pe întreaga sa viață. O sursă de inspirație pentru aceasta este sintagma din cadrul companiei Amazon: "you build it, you run it" unde echipa de dezvoltare își asumă responsabilitatea pentru mentenanța pe timpul producției. Aceasta îi aduce pe dezvoltatori în contact cu modul în care aplicația se comportă în producție și îmbunătățește contactul cu utilizatorii, preluând astfel o parte din efortul de mentenanță.

Această mentalitate, se leagă și de dependența de nevoile de business. Mai degrabă decât să privim aplicațiile ca un set de funcționalități complete, există o relație permanentă unde întrebarea se pune cum poate soft-ul să ajute utilizatorii pentru a sporii capacitățile de business.

Nu există nici un motiv pentru care această abordare nu s-ar putea aplica și în cadrul aplicațiilor monolit, dar granularitatea sporită oferită de servicii facilitează crearea relațiilor personale între dezvoltatori și utilizatori.

4.1.4 Puncte terminale inteligente și bus-uri simple

Pentru construirea unor structuri de comunicare între diferite procese există multe procese și abordări care se concentrează pe implementarea inteligenței în mecanismul de comunicare în sine. Un exemplu foarte bun este Enterprise Service Bus (**ESB**), aceste mecanisme oferind capacități complexe pentru rutarea mesajelor și aplicarea regulilor de business.

Microserviciile implică o altă abordare: puncte terminale inteligente și bus-uri simple (eng. "smart endpoints and dumb pipes"). Aplicațiile ce urmează aceste principii tind să fie cât mai decuplate, primind o cerere, aplicând logica adecvată și producând un răspuns. Acești pași sunt corelați folosind protocoale simple de tip REST mai degrabă decât cele complexe precum WS-Choreography, BPEL sau orchestrare prin intermediul unei uneelte centrale.

Cele mai folosite două protocoale sunt HTTP cerere-răspuns folosind API-uri pentru accesarea resurselor și mecanisme simple de trimiterea mesajelor. Echipele care lucrează cu microservicii folosesc principiile și protocoalele pe baza caror a fost construit și web-ul.

Al doilea mecanism folosit este cel al mesajelor transmise prin intermediul unui bus simplu. Infrastructura aleasă este adesea simplă (nu are decât rolul de a ruta mesajele) - implementări simple precum RabbitMQ sau ZeroMQ nu fac mai mult decât să pună la dispoziție un canal de comunicare asincron - inteligența rămânând în punctele terminale ce produc și consumă mesaje- în microservicii.

Într-un monolit, componentele execută instrucțiunile în cadrul aceluiași proces, iar comunicare între ele se face prin invocarea unei metode sau apelul unei funcții. Cea mai mare provocare

în trecerea de la monolit la microservicii constă în schimbare șablonului de comunicare. O conversie de la apeluri în cadrul procesului la apeluri de tip RPC conduce la o comunicare excesivă care nu oferă o performanță bună. În schimb trebuie înlocuită comunicarea granulară cu o comunicare mai puțin selectivă.

4.1.5 Guvernare distribuită

Una din consecințele unei guvernări centralizate este tendința de a standardiza tehnologia folosită. Experiența arată că aceasta abordează limitează flexibilitatea - "nu fiecare problemă este un cui și nu fiecare soluție este un ciocan". Se dorește folosirea uneltelor adecvate pentru a rezolva o problemă și în vreme aplicațiile monolit pot beneficia într-o oarecare măsură de folosirea mai multor limbaje de programare, nu este ceva comun.

Împărțirea componentelor unui monolith pe servicii implică alegerea limbajului de programare și a uneltelor adecvate pentru construirea fiecăruia. Bineînțeles, doar pentru că avem posibilitatea a construi sisteme eterogene din punct de vedere al tehnologiilor nu înseamnă că este ceva obligatoriu.

Echipele care construiesc microservicii preferă o abordare diferită de cea standard. În loc să se folosească un set de standarde scrise, se preferă crearea de unelte pe care ceilalți dezvoltatori le pot folosi pentru a rezolva probleme similare cu cele cu care s-au confruntat. Acestea sunt distribuite în cadrul unor grupuri mai largi, dar nu în mod exclusiv folosindu-se modelul open-source. Popularitatea sporită a git și github a contribuit la răspândirea practicilor open source în cadrul companiilor.

Netflix este un exemplu foarte bun de organizație care urmează această filozofie. Distribuind cod util și foarte bine testat sub forma unor librării încurajează alții dezvoltatori să rezolve probleme similare, însă lasă ușa deschisă pentru abordări diferite în caz de nevoie. Bibliotecile partajate tind să se concentreze pe probleme comune cum ar fi stocarea datelor, comunicarea inter-proces și după cum vom discuta în cele ce urmează automatizarea infrastructurii.

În cadrul comunității ce practică microservicii, întârzierile sunt în mod particular neatractive. Aceasta nu implică o lipsă de importanță acordată contractelor de servicii, chiar dimpotrivă din moment ce ele tind să fie mai numeroase. Ele sunt doar privite dintr-o perspectivă diferită de administrare. Șabloane precum "Tolerant Reader" și "Consumer-Driver Contracts" sunt adesea aplicate microserviciilor. Acestea facilitează o evoluție independentă a acestora. Folosirea contractelor orientate pe nevoile clienților ca și parte a proiectării crește nivelul de încredere și oferă un răspuns rapid asupra funcționării serviciului. Serviciul este apoi construit în vederea satisfacerii contractului. Aceste tehnici și apariția uneltelor adiacente, limitează nevoia centralizării contractelor prin scăderea cuplării temporale a serviciilor.

4.1.6 Managementul distribuit al datelor

Management distribuit al datelor se poate face în mai multe feluri. La nivelul cel mai abstract, înseamnă că modelul conceptul va diferi de la un sistem la altul. Acesta este un scenariu tipic în cadrul aplicațiilor de tip enterprise, perspectiva departamentului de vânzări asupra datelor va fi diferită față de cea a echipei de suport. În acest caz informațiile despre un anumit client pot diferi.

Aceasta este o problemă tipică care poate apărea la integrarea diverselor aplicații, însă poate apărea și la nivelul aceleiași aplicații. Un mod în care poate fi rezolvată această problemă este folosind șablonul "Bounded Design" din Domain-Driven Design (DDD). DDD împarte un domeniu complex în mai multe domenii limitate și mapează relațiile între acestea. Acest proces este util atât în cazul aplicațiilor monolit cât și în cel al microserviciilor, însă există o corelație naturală între serviciul și granițele contextului ce ajută la clarificarea și întărirea separării.

Optarea pentru un model conceptual distribuit implică și un model distribuit pentru soluțiile de stocare. În vreme ce aplicațiile monolitice folosesc o singură bază de date pentru persistența datelor, în companiile mari se preferă o singură bază de date pentru o suită de aplicații - o parte a acestei decizii fiind legată de costul licenței. În cadrul microserviciilor se preferă folosirea propriei baze de date de către fiecare serviciu în parte, fie instanțe diferite ale aceluiași sistem fie sisteme complet diferite - o abordare care se numește "Polyglot Persistence". Această abordare poate fi folosită și în cadrul aplicațiilor monolit, însă apare mai frecvent în cadrul microserviciilor.

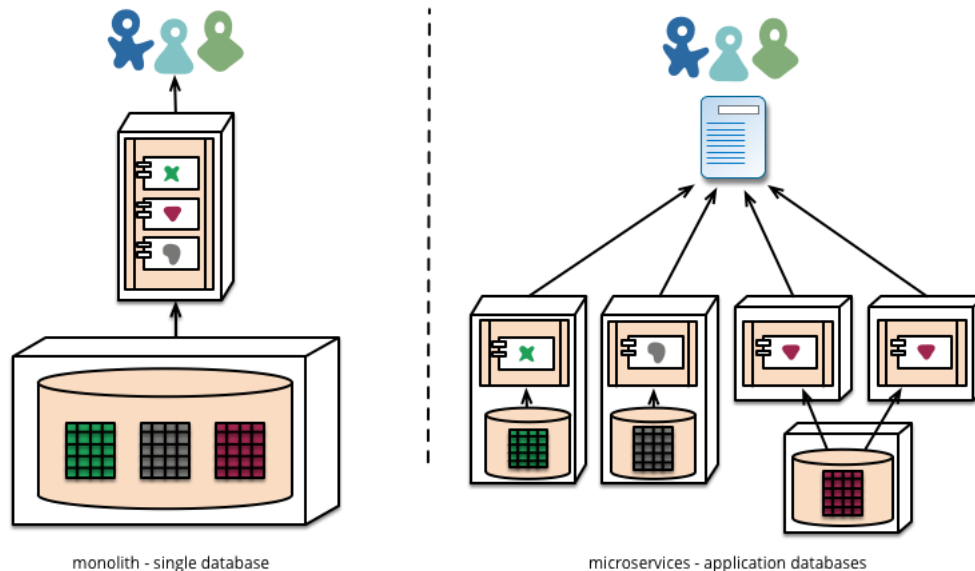


Figura 4.4: Model distribuit al datelor

Folosirea unui model distribuit pentru date implică și împărțirea responsabilității pentru mentenanța sistemelor. Abordarea comună pentru gestionarea actualizărilor a fost folosirea tranzacțiilor pentru asigurarea consistenței la nivelul mai multor resurse. Această abordare este folosită adesea în cadrul aplicațiilor de tip monolit.

Astfel tranzacțiilor contribuie la asigurarea consistenței datelor, dar impune și o cuplare temporală, ceea ce este problematic în cazul folosirii mai multor servicii. Tranzacțiile distribuite sunt foarte greu de implementat și de aceea microserviciile se axează pe o coordonarea fără tranzacții între servicii, cu mențiunea că în acest caz consistența nu este garantată și că această problemă va fi adresată prin mecanisme de compensare.

Faptul de a gestiona astfel inconsistențele este o nouă provocare pentru multe echipe de dezvoltare, dar este cea care adesea se pliează cel mai bine pentru cazurile de business. Adesea business-ul operează cu un anumit nivel de inconsistență pentru a răspunde repede la cereri, având la dispoziție mecanisme pentru trarea erorilor. Acest compromis rentează atât timp cât costul reparării greșelilor este mai mic decât costul pierderii anumitor informații de business.

4.1.7 Automatizarea infrastructurii

Tehnicile pentru automatizarea infrastructurii au evoluat foarte mult în ultimii ani - evoluția clodului și AWS - Amazon Web Services - în mod particular a redus foarte mult complexitatea operațională pentru gestionarea microserviciilor.

Multe dintre produsele și sistemele folosind microservicii sunt construite de echipe cu experiență în Continuous Delivery și predecesorul său, Continuous Integration. Echipele ce construiesc astfel aplicațiile folosesc în mod extensiv tehnici de automatizare a infrastructurii.

Acest lucru, denumit în literatura de specialitate ”**pipeline**”, este ilustrat mai jos:

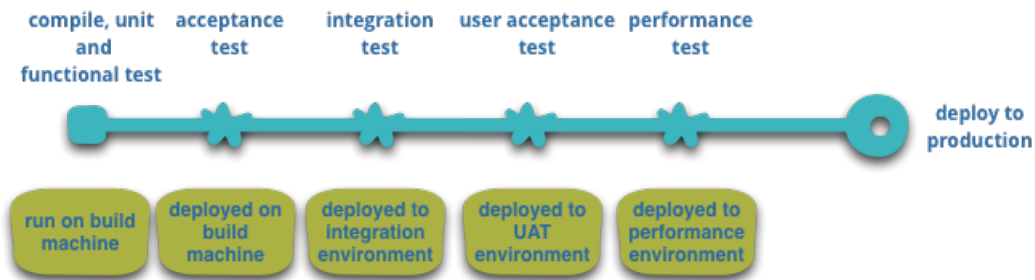


Figura 4.5: Pipeline

Câteva aspecte-cheie ale procesului de livrare continuă (e.g. Continuous Delivery) sunt: încrederea cât mai mare în faptul calitatea software-ului dată de o suită foarte mare de teste automate; trecerea dintr-un mediu în altul în cadrul pipeline-ului înseamnă instalarea automată pe fiecare nou mediu.

O aplicație monolit va fi compilată, testată și trecută prin aceste medii fără prea multe probleme. S-a constatat că investirea în automatizarea procesului de instalarea a unei aplicații în producție crește încrederea echipelor de a livra mult mai des pe acest mediu. Unul din obiectivele livrării continue este banalizarea procesului de instalare, astfel încât nu contează că se livrează una sau mai multe aplicații, lucrurile rămân la fel de simple.

O altă arie în care se observă folosirea extensivă a automatizării infrastructurii este administrarea microserviciilor în producție. În contrast cu afirmația precedentă legată de banalizarea instalării aplicațiilor în producție, peisajul operațional este complet diferit.

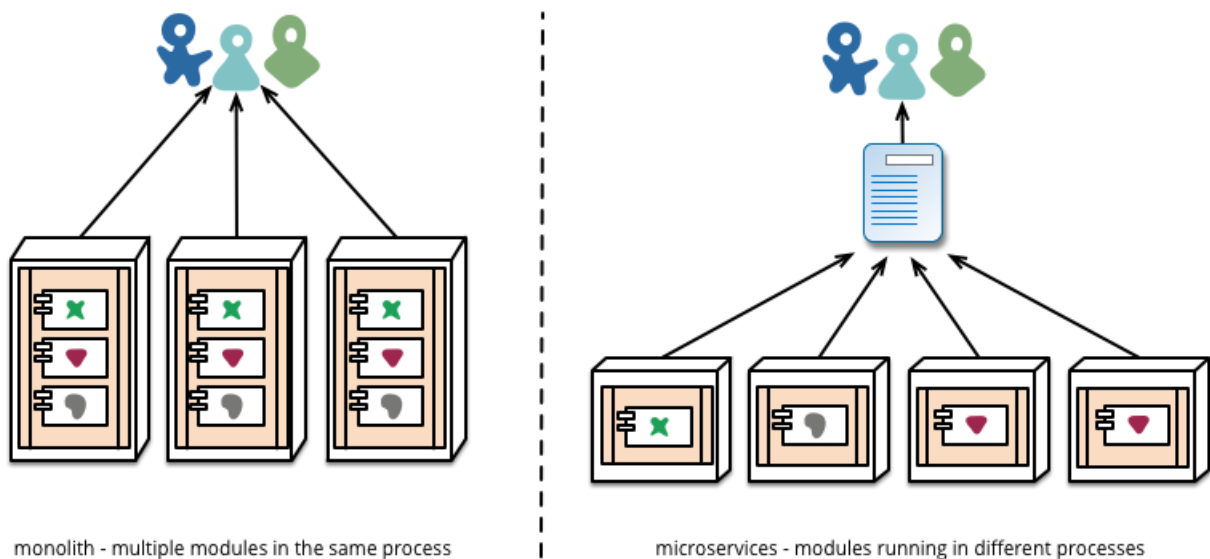


Figura 4.6: Instalarea modulelor diferă

4.1.8 Proiectare pentru dezastru

O consecință a folosirii serviciilor ca și componente este faptul ca aplicațiile trebuie proiectate pentru a putea tolera pene de servicii (indisponibilitatea acestora). Orice apel al unui serviciu poate eșua datorită indisponibilității furnizorului de serviciu, clientul ar trebuie deci să răspundă

într-un mod cât mai elegant. Acesta reprezintă un dezavantaj față de folosirea unui design monolit deoarece introduce complexitatea adițională. Consecința este că echipele responsabile pentru microservicii trebuie în permanență să gestioneze modul în care o pană a unui serviciu va afecta experiența utilizatorului. Adesea companiile efectuează teste prin care simulează indisponibilitatea serviciilor și uneori chiar a centrelor de date pentru a verifica comportamentul aplicațiilor.

Acest tip de teste automate realizate în producție ar fi indeajuns pentru a oferi echipelor operaționale datele necesare pentru a constata comportamentul aplicațiilor și pentru a face teste de performanță. Acest lucru nu implică faptul că aplicațiile monolitice nu pot suporta scenarii de test complexe, însă este mai puțin obișnuit.

Din moment ce serviciile pot fi întrerupte la orice moment, este important ca paneele să fie detectate cât mai repede, și eventual serviciul să fie repornit în mod automat. Aplicațiile bazate pe microservicii pun foarte mult accentul pentru monitorizarea în timp real, atât a parametrilor tehnicii (numărul de cereri pe secundă care trimise către o bază de date sau către serverul aplicativ) cât și cele de business (câte comenzi sunt procesate pe minut). În acest mod se echipele de dezvoltare au la dispoziție un sistem de monitorizare ce îi poate avertiza dacă ceva nu funcționează în parametrii.

Acest aspect este foarte important pentru o arhitectură bazată pe microservicii datorită folosirii extensive a orchestrării și a colaborării bazate pe evenimente ce duc la comportamente neprevăzute. În vreme ce descoperirea comportamentelor neprevăzute este lăudată de unii, adevărul este că de multe ori este un lucru rău. Monitorizarea este vitală pentru descoperirea acestora și remedierea lor într-un timp cât mai scurt.

Aplicațiile monolit ar putea fi construite la fel de transparente ca un microserviciu și în fapt chiar așa ar trebui să fie. Diferența este că absolut necesar să se știe când serviciile ce rulează pe o mașină distantă devin indisponibile. Folosind librării locale acest tip de transparență nu este foarte folositoare.

Echipele care lucrează cu microservicii se așteaptă să vadă soluții complexe de monitorizare și logare pentru fiecare serviciu în parte: pagini care arată statutul serviciilor și o multitudine de metrice operaționale și de business.

4.1.9 Design evolutiv

Adeptii microserviciilor privesc împărțirea unei aplicații pe servicii ca un pas natural prin care dezvoltatorii pot controla mult mai bine schimbările fără a îngreuna procesul. Controlarea schimbărilor nu înseamnă neapărat reducerea numărului de schimbări - cu atitudinea și uneltele potrivite se pot face schimbări frecvente, rapid și bine controlate.

În împărțirea unui sistem pe componente apare problema luării deciziei creării serviciilor, respectiva care sunt principiile ce trebuie folosite pentru a secționa aplicația în mai multe subsisteme. Proprietatea cheie a unei componente este independența interschimbării și a actualizării ceea ce faptul că vom cauta acele caracteristici ce vor permite rescrierea acesteia fără a afecta consumatorii. Există totuși grupuri care duc această idee mai departe și consideră că serviciile vor fi mai degrabă înlăturate și înlocuite cu altele, decât menținute pe termen lung.

Un exemplu foarte bun al unei aplicații gândite ca și monolit, dar care a fost evoluată în direcția microservicii este site-ul publicației The Guardian. Deși monolitul rămâne în continuare nucleul, noile funcționalități au fost realizate folosind microservicii ce apelează API-ul monolitului. Această abordare este deosebit de practică pentru funcționalitățile cu scop temporal limitat, cum ar fi paginile despre evenimente sportive. Astfel de pagini pot fi create repede folosind limbaje de programare ce permit dezvoltări rapide și șterse în momentul în care evenimentul s-a încheiat. O abordare similară poate fi întâlnită și în cadrul instituțiilor

financiare când noi servicii sunt adăugate pentru o oportunitate a pieței și decomisionate câteva săptămâni mai târziu.

Accentul pe schimbare este un caz special al unui principiu mult mai general al design-ului modular. Este indicat ca lucrurile ce se schimbă în același timp să rămână în același modul. Părți ale sistemului care se schimbă foarte rar ar trebui să fie în servicii diferite față de cele care se schimbă foarte des. Dacă adesea apare nevoia de a schimba două servicii în același timp, atunci acestea ar trebui să formeze un singur serviciu.

Adunarea componentelor în cadrul unui serviciu oferă oportunitatea de a crea planuri de instalare mult mai granulare. În cazul unui monolit orice schimbare implică o recompilare și reinstalarea a întregii aplicații. Folosind microservicii, este necesară doar reinstalarea serviciului modificat. Acesta poate simplifica și grăbi procesul de instalare. Dezavantajul este că apare riscul nesatisfacerii unora dintre consumatorii serviciului. Abordarea clasică pentru rezolvarea acestor probleme este versionarea, însă în cadrul microservicii aceasta este considerată ca o ultimă alternativă. Versionarea poate fi evitată printr-o proiectare a serviciilor cât mai tolerantă la schimbările furnizorilor.

4.2 Avatanje și dezavantaje ale microserviciilor

4.2.1 Avantaje

Șablonul arhitectural al microserviciilor prezintă un număr important de avantaje. În primul rând, rezolvă problema complexității, prin descompunerea a ceea ce altfel ar fi fost un aplicație monolit foarte mare într-un set de servicii. În vreme ce ansamblu funcționalităților rămâne neschimbat, aplicația a fost împărțită în grupuri de servicii mult mai ușor de administrat. Fiecare serviciu conține un set bine definit de funcționalități expuse folosind RPC sau un API. Acest șablon oferă un grad de modularitate care în practică ar fi extrem de greu de obținut în cadrul unei aplicații monolitice. În consecință serviciile individuale sunt mult mai rapid de dezvoltat și mult mai simplu de înțeles și menținut.

În al doilea rând, această arhitectură permite fiecărui serviciu să fie dezvoltat în mod independent de către o altă echipă. Dezvoltatorii au posibilitatea de a alege orice tehnologie consideră optimă pentru îndeplinirea cerințelor, în condițiile în care reușesc să expună în mod corect API-ul stabilit. Desigur, majoritatea organizațiilor doresc să evite potențialele probleme care pot apărea datorită folosirii prea multor tehnologii și limitează alegerile. Totuși, această liberate presupune faptul că dezvoltatorii nu mai sunt obligați să folosească aceleași tehnologii învechite care au existat la începutul proiectului. Când dezvoltă un serviciu nou, au ocazia să folosească tehnologii actuale. În plus, serviciile fiind relativ mici, devine fezabilă rescrierea unui serviciu vechi folosind tehnologie actuală.

În al treilea rând acest șablon permite fiecărui microserviciu să fie instalat în mod independent. Dezvoltatorii nu mai au nevoie să coordoneze instalarea modificărilor care sunt specifice serviciului lor. Aceste genuri de modificări pot fi instalate imediat ce au fost testate. De exemplu, echipa responsabilă de interfața grafică poate să efectueze rapid teste și să implementeze schimbări. Această arhitectură face posibilă integrarea continuă.

În ultimul rând, microserviciile permit fiecărui serviciu să fie scalat în mod individual. Pot fi instalate doar numărul necesar de instanțe pentru fiecare serviciu care satisface nevoile de capacitate și disponibilitate. În plus se poate folosi un suport fizic care să corespundă nevoilor serviciilor. De exemplu un serviciu de procesare a imaginilor, care solicită foarte mult procesorul, poate fi instalat pe o instanță EC2 Compute-Optimized în vreme ce o bază de date în memorie poate fi instalată pe o instanță EC2 Memory-Optimized.

4.2.2 Dezavantaje

Ca orice tehnologie, arhitectura microservicii prezintă dezavantaje. Un prim dezavantaj ar fi numele în sine. Numele microservicii pune extrem de mult accentul pe dimensiunea serviciului. De fapt, sunt foarte muți dezvoltatorii care pun accentul pe construirea serviciilor cu o granulație foarte fină. În vreme ce serviciile mici sunt preferate, este important de avut în vedere că ele reprezintă calea spre rezolvarea unei probleme și nu sunt un scop în sine. Scopul microserviciilor este să împartă aplicația îndeajuns de mult pentru a facilita dezvoltarea și instalarea flexibilă.

Un alt dezavantaj major este complexitatea care rezultă din faptul ca o aplicație ce folosește microservicii este un sistem distribuit. Dezvoltatorii trebuie să aleagă și să implementeze sunt mecanism de comunicare inter-proces bazat fie pe mesaje fie pe RPC. În plus, trebuie să scrie cod care să se gestioneze situațiile în care furnizorul de serviciu este încet sau indisponibil. În vreme ce niciuna din aceste probleme nu este foarte complexă, este considerabil mai complexă decât o aplicație monolit un module comunică prin apeluri de proceduri la nivel de limbaj.

O altă provocare a microserviciilor o reprezintă arhitectura distribuită a bazei de date. Tranzacțiile de business care modifică entități multiple sunt ceva comun. Acest tip de tranzacții

sunt trivial de implementat în cadrul unei aplicații monolit deoarece există o singură bază de date. Într-o arhitectură bazată pe microservicii multiplele baze de date ale serviciilor trebuie actualizate. Folosirea unui sistem distribuit de tranzacții nu este în mod normal o opțiune, și nu doar datorită Teoremei CAP - denumită și Teorema lui Brewer, care afirmă că este imposibil ca un sistem informatic să ofere simultan constrângerile următoare: consistență, disponibilitate și toleranță parțială. Ele pur și simplu nu sunt suportate de sistemele de baze de date NoSQL și nici de brokerii de mesaje. Compromisul este folosirea unei abordări bazate pe consistență, care este mai problematică pentru dezvoltatori.

Testarea microserviciilor este mult mai complexă. De exemplu, folosind o bibliotecă modernă precum Spring Boot este foarte simplu de scris o clasă de test care pornește o aplicație web monolitică și care îi testează API-ul. În contrast cu aceasta, un test similar pentru un serviciu ar necesita lansarea serviciului și a serviciilor de care depinde, sau simularea acestora. Încă odată, acestea nu sunt foarte complexe de realizat însă este importantă estimarea efortului necesar pentru a face aceasta.

O altă provocare majoră legată de șablonul arhitectural al microserviciilor este implementarea schimbărilor care se întind pe mai multe servicii. Considerând cazul implementării unei cerințe care necesită modificări la nivelul serviciilor A, B și C, unde A depinde de B și B depinde de C. Într-o aplicație monolitică am putea pur și simplu implementa și integra modificările la nivelul modulelor corespunzătoare și apoi instala pe toate odată. Prin comparație, în cazul microserviciilor trebuie să planificăm și coordonăm cu grijă schimbările fiecărui serviciu. În cazul de față trebuie actualiza serviciul C, urmat de serviciul B și abia apoi serviciul A. Din fericire totuși, majoritatea schimbărilor vizează un singur serviciu, iar scenariile multi-serviciu care să necesite coordonare sunt destul de rare.

Instalarea unei aplicații bazate pe microservicii este mult mai complexă. O aplicație monolit este pur și simplu instalată pe un set de servere identice ce se află în spatele unui load-balancer tradițional. Fiecare instanță este configurată cu locațiile serviciilor de infrastructură cum ar fi baza de date și broker-ul de mesaje. Pe de altă, o aplicație bazată pe microservicii este alcătuită dintr-un număr mare de servicii. De exemplu, Netflix (platformă de vizualizare de conținut video on-line) folosește peste 600 de servicii. Fiecare serviciu are mai multe instanțe. Aceasta implică o complexitate crescută datorată nevoii de configurare, instalare, scalare și monitorizare a fiecăruia dintre servicii. În plus, este necesară implementarea unui mecanism de descoperire a serviciilor care să permită acestora să obțină adresa și portul altor servicii cu care ar avea nevoie să comunice. Modul tradițional de operare folosind tichete și operațiuni manuale nu scalează bine la acest nivel de complexitate. În consecință, instalarea cu succes a unei aplicații cu microservicii necesită un grad ridicat de control asupra procesului din partea dezvoltatorilor și un nivel înalt de automatizare.

O soluție pentru obținerea automatizării este folosirea unui furnizor de servicii de tip "platforma ca serviciu" (eng. Platform as a Service - PaaS). Furnizorii de PaaS oferă dezvoltatorilor o modalitate simplă de instalare și administrare a microserviciilor. Acest îi izolează de probleme cum ar fi cumpărarea și configurarea de resurse IT. În același timp, specialiștii de sisteme și rețele care configurează PaaS se pot asigura că toate bunele practici au fost respectate. O altă soluție este dezvoltarea a propriului PaaS. Un punct de plecare ar putea fi o soluție de clustering, precum Mesos sau Kubernetes folosit împreună cu tehnologii precum Docker.

4.3 Construirea microserviciilor: folosirea unui API gateway

În vederea realizării unei aplicații folosind microservicii o modalitatea de interacțiune între aceasta și utilizatori trebuie aleasă. În cazul unei aplicații monolitice există doar un set de puncte terminale. Într-o arhitectura de tip microservicii, fiecare serviciu expune un set de funcționalități. Una din posibilele metode prin care se poate realiza comunicarea client aplicație este prin intermediul unui API gateway.

4.3.1 Introducere

Pornind de la cazul practic al unei aplicații mobile a unui magazin online. Cel mai probabil aceasta va conține o pagină care va afișa detalii legate de produse. Deși este aplicație mobilă, pagina de detalii a produsului oferă multe informații. De exemplu, nu doar că există informații de bază despre produs, dar aceasta include și: numărul de produse din coș, istoricul comenzilor, recenziile altor clienți, opțiuni de livrare.

În cazul unei aplicații monolit, un client de mobil ar putea obține datele printr-un simplu apel REST către aplicație: `GET api.company.com/productdetails/productId`. Un load balancer ar ruta cererea către unul din cele N servere identice. Aplicația ar lansa diverse interogări ale bazei de date care ar întoarce un răspuns către client.

Pe de altă parte, folosind microservicii, datele afișate pe pagină aparțin mai multor microservicii. O posibilă asociere între informații și servicii ar fi următoarea:

- Shopping Cart Service - numărul de obiecte din coș
- Order Service - istoricul comenzilor
- Catalog Service - informații de bază despre produs, cum ar fi numele, imaginea și prețul
- Review Service - recenziile clienților
- Shipping Service - opțiuni de livrarea, termeni limită și costul de livrare pentru fiecare furnizor

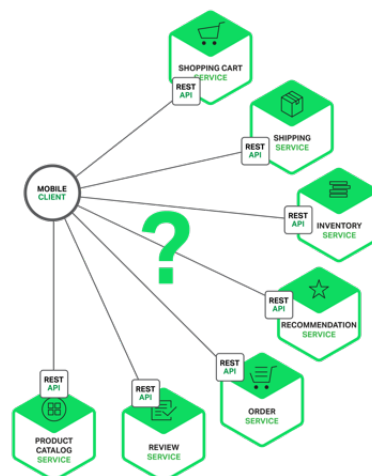


Figura 4.7: Diagramă a serviciilor

În acest caz o strategie de comunicare între clienți și servicii trebuie aleasă.

4.3.2 Comunicare directă între client și microservicii

Teoretic, un client ar lansa cereri directe către fiecare microserviciu. Fiecare microserviciu ar avea un punct terminal public de tipul: `https://serviceName.api.company.name`. Acest URL s-ar mapa la load balancer-ul microserviciului, care ar distribui cererile serverelor din back-end. Pentru obținerea detaliilor legate de produs, clientul ar lansa cereri spre fiecare din serviciile listate mai sus.

Din nefericire, există un număr de provocări și limitări în cazul acestei soluții. O primă problemă ar fi nepotrivirea dintre nevoile clientului și API-ul foarte granular expus de fiecare microserviciu. Clientul în acest exemplu va trebui să lanseze 7 cereri separate. În cazul unei aplicații mai complexe ar putea fi mult mai multe. În vreme ce un client ar putea lansa un număr mare de cereri într-o rețea locală ar fi probabil foarte inefficient în cazul internetului și în mod definitiv nepractic peste o rețea mobilă. Această abordare complică și codul care rulează la client.

O altă problemă cu această abordare ar fi că în unele cazuri microserviciile ar putea folosi protocoale care nu se pretează a fi folosite peste internet. Un serviciu ar putea folosi o librărie de tip RPC în vreme ce altul ar folosi protocol de mesagerie AMQP. Nici unul nu se pretează a fi folosit într-un browser și nici să traverseze un firewall și este foarte bine folosit în intern. O aplicație ar trebui să folosească protocoale precum HTTP sau WebSocket în afara firewall-ului.

Un dezavantaj al acestei abordări este că refactorizarea devine dificilă. Pe de-a lungul vieții sistemului este posibil ca partiționarea serviciilor să fie schimbată. De exemplu, două servicii ar putea unite, altul ar putea fi împărțit în două sau mai multe. Dacă, însă, clienții comunică direct cu serviciile, atunci aceste operațiuni devin extrem de dificile.

Datorită acestor tipuri de probleme în foarte puține cazuri clienții discută direct cu microserviciile.

4.3.3 Folosirea unui API Gateway

În general o abordare mult mai bună este folosirea a ceea ce se numește un API gateway. Un API gateway reprezintă un server care este punctul de intrare în sistem. Este similar cu șablonul "Facade" programarea orientată pe obiecte. API Gateway-ul încapsulează arhitectura internă a sistemului și expună un API care poate fi adaptat pentru fiecare client. S-ar mai putea ocupa și de alte funcții precum autentificare, monitorizare, load balancing, caching, administrarea și modelarea cererilor.

Sistemul precedent ar putea fi regândit pentru a include un API Gateway:

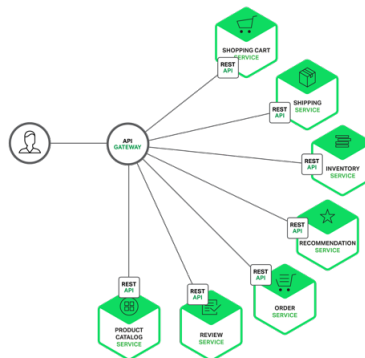


Figura 4.8: Folosirea unui API gateway

API Gateway-ul este reponsabil pentru rutarea cererilor, compoziția și translația protocoalelor. Toate cererile de la clienți trec întâi prin API Gateway, care apoi le rutează către

microserviciul adecvat. De cele mai multe ori, o cerere venită către gateway implică apelarea mai multor microservicii și agregarea rezultatelor. Pe lângă acestea, poate efectua tranzlații între protocoale web precum HTTP, WebSocket și protocoale care nu sunt orientate pe web, folosite în intern.

Printre funcționalitățile sale se numără și expunerea unui API pentru fiecare client, în special pentru clienții mobile. Luând de exemplu cazul enunțat mai sus al detaliilor unui produs. API Gateway-ul poate expune un punct terminal (`/productdetails?productid=xxx`) care va permite clientului mobile să obțină toate datele necesare prin intermediul unei singure cereri. Acesta va trata cererea invocând diverse servicii - informații de produs, recomandări, recenzii etc - și va combina rezultatele.

Un exemplu foarte bun de API Gateway este cel al firmei Netflix. Serviciul de streaming Netflix este disponibil într-o multitudine de formate : televizoare, box-uri, telefoane mobile, sisteme de gaming, tablete, etc. Inițial, Netflix a încercat să expună un API care să răspundă tuturor acestor formate. Mai târziu, au descoperit ca acesta nu va funcționa bine datorită varietății mari de dispozitive și nevoilor aferente. Astăzi folosesc un API Gateway care expune un API dedicat rulării codului ce implementează șablonul de tip adaptor pentru fiecare tip de dispozitiv. Un adaptor rezolvă fiecare cerere prin invocarea în medie a șase șapte servicii de back-end.

4.3.4 Avantaje și dezavantaje ale folosirii unui API Gateway

Așa cum este de așteptat, folosirea unui API Gateway are avatanje și dezavantaje. Un avantaj major ar fi încapsularea întregii structurii interne a aplicației. Clientul discută cu această entitate, mai degrabă decât cu fiecare serviciu în parte. API Gateway-ul expune fiecărui tip de client un API specific. Aceasta reduce numărul de apeluri între client și aplicație, dar duce și la o simplificare a codului client.

Printre dezavantaje se numără faptul ca este încă o componentă care trebuie dezvoltată, instalată și menținută. Există de asemenea riscul ca aceasta să devină un punct de aglomerare. Dezvoltatorii trebuie să actualizeze API Gateway-ul pentru a expune fiecare punct terminal al microserviciilor. Este important ca acest proces să fie cât mai simplu cu putință, altfel dezvoltatorii vor trebui să aștepte pentru a implementa schimbările. În ciuda acestor dezavantaje, în majoritatea cazurilor implementarea unei astfel de entități aduce valoare sistemului.

4.3.5 Implementarea unui API Gateway

Odată listate motivele folosirii unui API Gateway împreună cu compromisurile făcute, un alt subiect de interes reprezintă probleme legate de design ce trebuie luate în calcul.

4.3.5.1 Performanță și scalabilitate

Doar o mică parte din companii funcționează la o scară îndeajuns de mare pentru a avea nevoie de a procesa milioane de cereri pe zi. Totuși, pentru majoritatea aplicațiilor performanța și scalabilitatea API Gateway-ul sunt foarte importante. În acest caz, are sens construirea unui API Gateway pe baza unei platforme asincrone cu I/O neblocant. Există o varietate de tehnologii care pot implementa un astfel de componentă. Bazându-ne pe JVM (Java Virtual Machine) se poate construi o astfel de platformă folosind unul din framework-urile de tip NIO (Non-blocking IO) precum Netty, Vertx, Spring Reactor sau JBoss Undertow.

4.3.6 Folosirea unui model de programare reactiv

API Gateway-ul prelucrează o parte din cereri prin simpla rutare către serviciile de backend adecvate. O altă parte din cereri este prelucrată invocând mai multe servicii și agregând rezultatele. În unele cazuri aceste cereri sunt independente. Pentru a minimiza timpul de răspuns, API Gateway-ul ar trebui să lanseze cererile în mod independent în același timp. Uneori însă există dependențe între cereri. În acest caz este nevoie de o validare a cererii prin apelarea unui serviciu de autentificare înainte de a ruta cererea către un serviciu de backend.

Scrierea codului care apelează API-ul folosind abordarea tradițională de apelare asincronă poate deveni repede foarte complicată. Codul va fi foarte imbricat, dificil de înțeles și predispus la erori. O abordare mult mai bună este scrierea codului pentru API Gateway folosind stilul declarativ ce folosește o abordare declarativă. Exemple de abstractizări declarative includ Future pentru Scala, CompletableFuture pentru Java8 și Promise pentru JavaScript. Mai există de asemenea și Reactive Extensions, care a fost inițial dezvoltată de Microsoft pentru .NET. Netflix a creat RxJava pentru JVM pentru a fi folosit în API Gateway-ul lor. Există de asemenea și RxJS pentru JavaScript, care rulează atât în browser cât și pe Node.js. Folosirea unei abordări reactive va permite scrierea mult mai simplă și eficientă a codului pentru API Gateway.

4.3.7 Invocarea serviciilor

O aplicație bazată pe microservicii este un sistem distribuit și trebuie să folosească un mecanism de comunicare inter-proces. Există două stiluri de comunicare inter-proces. O primă opțiune este folosirea unui mecanism asincron bazat pe mesaje. Câteva implementări folosesc brokere de mesaje precum JMS sau AMQP. Altele precum Zeromq, nu dispun de brokeri, iar serviciile comunică direct. Celălalt stil de comunicare inter-proces se bazează pe mecanisme asincrone cum ar fi HTTP sau Thrift. Un sistem va folosi în mod normal ambele stiluri. Ar putea chiar folosi și mai multe implementări ale fiecărui stil. În consecință, API Gateway-ul va trebui să suporte o varietate de mecanisme de comunicare.

4.3.8 Descoperire serviciilor

API Gateway-ul are nevoie de cunoașcă locația (adresa IP și portul) fiecărui microserviciu cu care comunică. Într-o aplicație tradițională, acestea ar putea fi codate în dur, însă într-o aplicație modernă bazată pe microservicii ce rulează în cloud aceasta nu este o problemă simplă. Serviciile legate de infrastructură, cum ar fi broker-ul de mesaje, va avea în mod normal o locație statică, care poate fi specificată prin intermediul variabilelor de mediu ale sistemului de operare. Serviciile aplicației au locații asignate în mod dinamic. În plus, setul de instanțe al unui serviciu se schimbă dinamic datorită auto-scalării și actualizărilor. În consecință, API Gateway-ul, ca orice alt serviciu client din sistem, are nevoie să folosească un sistem de descoperire a serviciilor : fie Server-Side Discovery (descoperire pe partea de server), fie Client-Side Discovery (descoperire pe partea de client). Acest subiect va fi detaliat în capitolele ce urmează, însă pentru moment putem menționa că în cazul folosirii a descoperirii serviciilor efectuate de client API Gateway-ul trebuie să poată interoga registrul de servicii, care este o bază de date cu toate microserviciile și locațiile lor.

4.3.9 Tratarea penelor parțiale

O altă problemă care trebuie adresată atunci când implementăm un API Gateway este problema penelor parțiale ale serviciilor. Această problemă apare în cadrul tuturor sistemelor

distribuite când se apelează un serviciu lent sau indisponibil. API Gateway-ul nu ar trebui să fie blocat așteptând un răspuns al unui serviciu. Totuși modul în care tratează penele de serviciu depinde de fiecare scenariu în parte și de ce serviciu este indisponibil. De exemplu, dacă serviciul de recomandări este indisponibil în cazul detaliilor de produs, API Gateway-ul ar trebui să returneze restul detaliilor de produs deoarece acestea sunt în continuare relevante pentru utilizator. În acest caz secțiunea de recomandări ar putea fi fie goală, fie populată cu o listă de zece produse codate în dur. Dacă, însă serviciul care oferă informații despre produs este indisponibil atunci API Gateway-ul ar trebui să returneze o eroare clientului.

API Gateway-ul ar putea returna de asemenea date din memoria cache, dacă aceasta este disponibilă. De exemplu, din moment ce prețurile produselor nu se schimbă foarte des, API Gateway-ul ar putea returna date din memoria cache în cazul în care serviciul de prețuri nu este disponibil. Datele ar putea fi stocate de către API Gateway în sine sau ar putea fi stocate într-un sistem extern precum Redis sau Memcached. Returnând fie lista de produse codate în dur fie date din memoria cache, API Gateway-ul asigură o bună experiență a utilizatorului în cazul în care apar pene de serviciu.

Netflix Hystrix este o bibliotecă foarte folositoare pentru scrierea de cod ce apelează servicii la distanță. Hystrix anulează apeluri ce nu sunt rezolvate într-un anumit timp. Implementează deasemenea un șablon de tip *circuit breaker*, care împiedică clientul să aștepte fără sens după un serviciu indisponibil. Dacă rata de erori depășește un anumit nivel specificat, Hystrix lansează întreruperea circuitului și toate cererile vor eșua instant pentru o perioadă de timp. Hystrix permite definirea unor acțiuni în cazul în care o cerere eșuează, cum ar fi citirea dintr-o memorie cache sau returnarea unei valori implicite. Se recomandă folosirea unei biblioteci alternative în cazul folosirii unor tehnologii care nu sunt bazate pe JVM.

4.4 Construirea microserviciilor: Comunicarea inter-process în cadrul microserviciilor

4.4.1 Introducere

Într-o aplicație monolit, componentele se invocă una pe cealaltă prin apeluri la de proceduri și metode la nivel de limbaj. Pe de alta parte, o aplicație bazată pe microservii este un sistem distribuit care rulează pe mai multe masini. Fiecare instanță a unui serviciu este un proces. În consecință, după cum se poate vedea mai jos, serviciile trebuie să interacționeze folosind un mecanism de comunicare inter-proces (IPC - eng. inter-process communication)

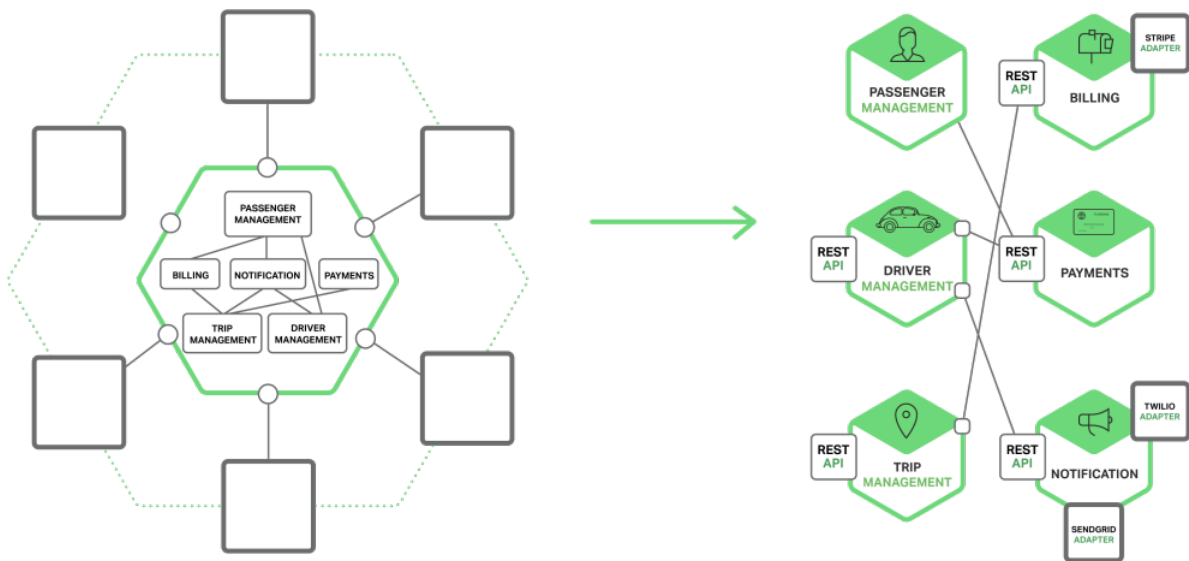


Figura 4.9: Comunicare inter-process

În cele ce urmează vom studia diversele tehnologii IPC, dar înainte vom explora diverse subiecte legate de proiectare.

4.4.2 Stiluri de interacțiune

În alegerea unui mecanism IPC pentru un serviciu, este util să luam în calcul cum serviciul urmează să interacționeze. Există o varietate de stiluri de interacțiune de tip client-server. Ele pot fi clasificate în două dimensiune. Prima dimensiune este dacă sau nu interacțiunea este unu-la-unu sau unu-la-mai-mulți:

- unu-la-unu : fiecare cererea a clientului este procesată de exact o instanță a unui serviciu.
- unu-la-mai-mulți: fiecare cerere este procesată de mai multe instanțe de serviciu.

Al doilea criteriu de luat în calcul este sincronicitatea:

- sincronă: clientul așteaptă răspunsul de la serviciu, timp în care blochează execuția altor operații
- asincronă: clientul nu blochează execuția și răspunsul, dacă există, nu trebuie trimis imediat.

Tabelul de mai jos arată diferite tipuri de interacțiuni:

	Unu-la-unu	Unu-la-mai-multi
Sincron	Cerere/Răspuns	-
Asincron	Notificare Cerere/Răspuns asincron	Publish/Subscribe Publish/Răspunsuri asincrone

Există următoarele tipuri de interacțiuni unu-la-unu:

- Cerere/răspuns - Un client lansează o cerere către un serviciu și așteaptă un răspuns. Clientul așteaptă răspunsul într-un timp rezonabil. Într-o aplicație care folosește fire de execuție, firul de execuție care lansează cererea s-ar putea chiar bloca în timp ce așteaptă.
- Notificare - clientul trimite o cerere la un serviciu, însă un răspuns nu este așteptat.
- Cerere/răspuns asincron - un client trimite o cerere la un serviciu, care răspunde asincron. Clientul nu se blochează în timp ce așteaptă și este proiectat plecând de la presupunerea că răspunsul ar putea ajunge după o perioadă mai lungă de timp.

Există următoarele tipuri de interacțiuni unu-la-mai-multi:

- Publish/subscribe - un client publică un mesaj de notificare, care este consumat de zero sau mai multe servicii interesate.
- Publish/async responses - un client postează un mesaj de cerere și apoi așteaptă o perioadă de timp pentru răspunsurile de la serviciile interesate.

Fiecare serviciu folosește o combinație a acestor stiluri de interacțiune. Pentru unele servicii, un singur mecanism IPC este suficient. Alte servicii ar putea avea nevoie să folosească o combinație de mecanisme IPC. Schema de mai jos arată cum serviciile dintr-o aplicație de preluare a cererilor de taxi ar putea interacționa atunci când un client lansează o cerere.

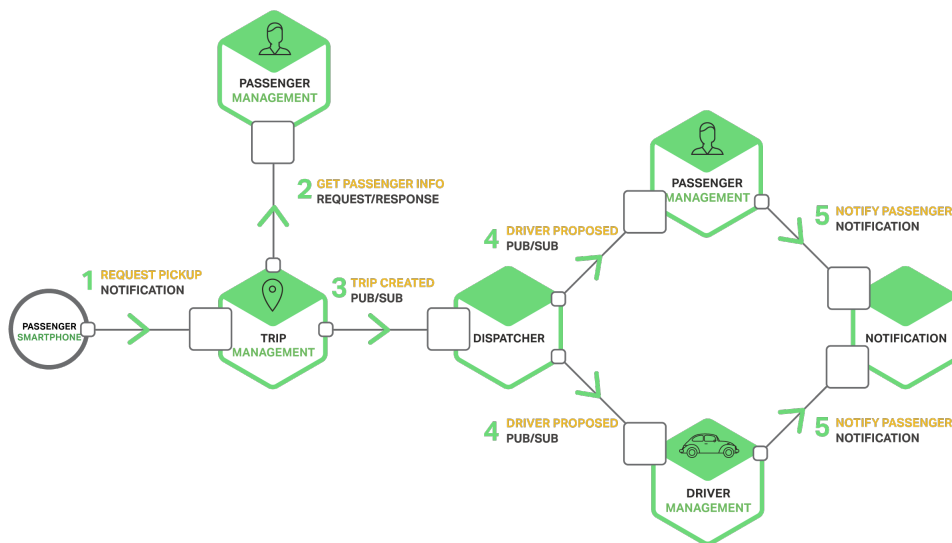


Figura 4.10: Exemplu de interacțiune servicii

Serviciile folosesc o combinație de notificări, cerere/răspuns și publish/subscribe. De exemplu, telefonul mobil al clientului trimite o notificare către serviciul "Trip Management" pentru

a cere o mașină. Acest serviciu verifică dacă contul clientului este activ folosind cerere/răspuns pentru a invoca serviciul "Passenger". Apoi serviciul "Trip Management" crează o excursie și folosește publish/subscribe pentru a notifica celălalte servicii incluzând dispatcher-ul, care va localiza un șofer disponibil.

4.4.3 Definirea API-urilor

API-ul unui serviciu reprezintă contractul între serviciu și clienții săi. Indiferent de mecanismul IPC ales, este importantă definirea unui API-ului folosind un limbaj de tip IDL - Interface Definition Language. Există chiar argumente care susțin abordare "API-first" în cadrul procesului de definire a API-urilor. În acest caz se începe dezvoltarea serviciului prin definirea interfeței împreună cu clientul. Numai în momentul stabilirii interfeței se poate începe implementarea serviciului. Operând în acest mod sunt îmbunătățite șansele ca serviciul să răspundă cerințelor clientului.

Natura API-ului depinde foarte mult de mecanismul IPC folosit. Dacă se folosesc mesaje, atunci API-ul va fi compus din canale de mesaje și tipuri de mesaje. Dacă în schimb se folosește HTTP, API-ul va fi format din URL-uri și formate de cereri și răspuns.

4.4.4 API-uri evolutive

API-ul unui serviciu se va schimba în mod invariabil pe parcursul timpului. Într-o aplicație monolit modificarea API-ului și a clienților acestui este o operație simplă. Într-o aplicație bazată pe microservicii aceasta implică un efort mult mai mare, chiar dacă toți consumatorii API-ului sunt alte servicii din cadrul aceleiași aplicații. În plus procesul de migrare spre noua versiune a serviciului se va face incremental astfel încât și vechea și noua versiune a serviciului vor funcționa în același timp. Este important a lua în considerare o strategie pentru tratarea acestor probleme.

Modul în care este implementată schimbarea depinde de mărimea schimbării. Unele sunt minore și retro-compatibile cu versiunea precedentă, cum ar fi de exemplu în cazul adăugării unor atribute la cererile sau răspunsurile HTTP. Serviciile și clienții ar trebuie să implementeze principiul robusteții *eng. robustness principle*, care spune că un serviciu ar trebui să fie conservativ în ceea ce trimite ca și răspuns, dar flexibil în ceea ce primește ca și cerere. Clienții care folosesc un API mai vechi ar trebuie să poată să-și continue activitatea cu noua versiune a serviciului. Serviciile ar trebui să returneze valori implicite pentru atributele care lipsesc, iar clienții ar trebui să ignore parametrii adiționali primiți în răspuns. Este importantă folosirea unui mecanism IPC și un format al mesajelor care să permită modificarea facilă a API-urilor.

Uneori, însă, modificările aduse API-urile nu sunt atât de simple și nici retro-compatibile. Din moment ce clienții nu pot fi forțați să migreze imediat la noua versiune, serviciul trebuie să ofere suport și pentru versiunile mai vechi o perioadă de timp. În cazul folosirii unui mecanism bazat pe protocolul HTTP, precum REST, o abordare ar fi includerea versiunii de serviciu în URL. Astfel fiecare serviciu ar putea expune mai multe versiuni în același timp. O altă abordare ar fi instalarea mai multor instanțe fiecare dintre ele rulând o anumită versiune a serviciului.

4.4.5 Gestionarea penelor parțiale de serviciu

Într-un sistem distribuit există permanent riscul unui pene de serviciu. Din moment ce clienții și serviciile sunt procese separate, există posibilitatea ca un serviciu să nu răspundă în timp util la cererea clientului. Un serviciu ar putea fi oprit din cauza unei pene, a mentenanței sau supra-încărcat și să răspundă foarte încet la cereri.

Considerând un caz concret în care un client apelează un serviciu indisponibil. O implementare naivă a clientului ar bloca execuția acestui pe termen nedefinit în așteptarea răspunsului. Nu doar că aceasta implică o experiență neplăcută a utilizatorului, dar în multe cazuri ar consuma resurse valoroase pentru un fir de execuție. Într-un final, aplicația va rămâne fără fire de execuție și se va bloca.

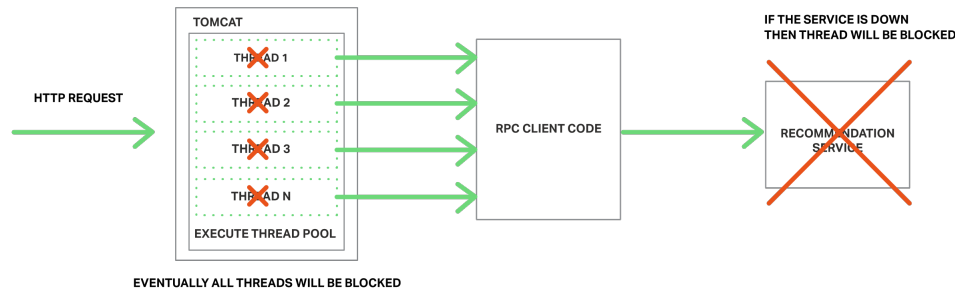


Figura 4.11: Consumarea resurselor unei aplicații

Pentru evitarea acestei probleme, este importantă proiectarea serviciilor în vederea gestionării penelor de serviciu. O abordare bună ar fi folosirea uneia sau a unora din strategiile următoare:

- indisponibilitatea rețelei - evitarea blocării pe termen nedefinit poate fi evitată prin folosirea unor contorii (*eng. timeouts*) în așteptarea unui răspuns. Folosirea contorilor garantează că resursele nu vor fi cuplate în mod nedefinit.
- limitarea numărului de cereri - prin stabilirea unei limite superioare a numărului de cereri pe care un client le are către un serviciu se poate bloca lansarea de noi cereri, deoarece se consideră că acestea nu vor fi rezolvate într-un timp acceptabil.
- circuit breaker pattern - prin monitorizarea numărului de cereri cu succes și eșuate se poate stabili un prag de la care mecanismul poate fi activat, ceea ce sugerează ca serviciul este indisponibil și că noile cereri vor eșua. După o perioadă tampon, clientul poate reîncerca și dacă cererea este cu succes atunci mecanismul este dezactivat.
- mecanisme de rezervă - în cazul în care o cerere eșuează, se poate returna o valoare din memoria cache sau una implicită.

Hystrix este o librărie open-source care implementează toate aceste mecanisme, ce poate fi folosită într-un mediu bazat pe JVM.

4.4.6 Tehnologii IPC

Există o multitudine de tehnologii IPC care pot fi folosite. Serviciile pot folosi mecanisme de comunicare sincrone de tip cerere/răspuns precum REST sau Thrift. Deasemenea, pot folosi mecanisme de comunicare asincronă bazate pe mesaje precum AMQP sau STOMP. Există o varietate de formate de mesaje. Serviciile pot folosi formate ușor de citit precum JSON sau XML sau formate binare, care sunt mai eficiente, precum Avro sau Protocol Buffers.

4.4.7 Comunicarea asincronă bazată pe mesaje

Folosind acest tip de comunicare, procesele interacționează asincron prin schimb de mesaje. Un client lansează o cerere către un serviciu trimițând un mesaj. Dacă serviciul trebuie să răspundă, o va face prin trimiterea unui mesaj separat către client. Din moment ce comunicarea este asincronă, clientul nu este blocat în timp ce așteaptă răspunsul. În schimb, clientul este scris plecând de presupunerea că răspunsul nu va fi recepționat imediat.

Un mesaj este alcătuit din antete - metadate precum expeditorul - și corpul mesajului. Mesajele sunt trimise pe canale. Un număr variabil de producători pot trimite mesaje către canal. În mod similar, orice număr de consumatori pot primi mesajele de la canal. Există două tipuri de canale: punct-la-punct sau publish-subscribe. Un canal punct-la-punct livrează mesajul fix unui consumator care citește de pe canal. Serviciile ce folosesc canale punct-la-punct folosesc interacțiunea unu-la-unu descrisă mai devreme. Un canal de tip publish-subscribe livrează mesajul către toți consumatorii. Serviciile ce folosesc canale publish-subscribe interacționează în modul unu-la-mai-mulți descris mai sus.

Există multe sisteme de mesaje, dar este de preferat să fie ales unul care să suporte mai multe limbaje de programare. Unele sisteme de mesagerie oferă suport pentru protocoale standard precum AMQP și STOMP. Altele folosesc protocoale proprietare, dar care însă sunt documentate. Există un număr mare de sisteme de mesagerie open-source precum RabbitMQ, Apache Kafka, Apache ActiveMQ și NSQ. La nivel înalt toate oferă suport pentru mesaje și canale și încearcă să fie performante și scalabile, însă fiecare prezintă diferențe la nivelul modelului de broker.

Avantajele folosirii mesajelor:

- decuplarea clientului de serviciu - un client lansează o cerere prin simpl trimitere a unui mesaj pe canalul adecvat. Clientul nu cunoaște ce instanțe de serviciu există și nici nu trebuie să folosească vreun mecanism de identificare pentru a afla locația serviciului.
- zona de tampon - folosind protocoale sincrone de tip cerere/răspuns, precum HTTP, atât clientul cât și serverul trebuie să fie disponibil în momentul schimbului. Pe de altă parte, un broker de mesaje poate să adune mesajele scrise într-un canal până ce acestea sunt procesate de un consumator. Asta înseamnă că un magazin online poate să accepte comenzi de la clienți chiar dacă sistemul de preluare al acestora este încet sau indisponibil.
- interacțiuni flexibile client-server - mesageria implementează toate stilurile de interacțiune descrise mai devreme.
- comunicare inter-process explicită - mecanismele bazate pe RPC dau impresia unui apel local al metodei, când aceasta este pe un server distant. Totuși, datorită posibilității unei pene, lucrurile sunt un pic diferite. Mesageria explicitează aceste diferențe astfel încât programatorii să nu aibă falsul sentiment de securitate în momentul apelării unor funcții.

Dezavantajele folosirii mesajelor ar fi :

- complexitate operațională crescută - sistemul de mesagerie este încă o componentă a sistemului care trebuie instalată, configurată și menținută. Este foarte important ca broker-ul de mesaje să aibă un grad înalt de disponibilitate, altfel robustețea sistemului va fi impactată.
- complexitatea implementării interacțiunilor cerere/răspuns - interacțiunile cerere/răspuns necesită efort în plus pentru a fi implementate. Fiecare cerere trebuie să conțină un identificator al canalului de răspuns și un identificator de corelație. Serviciul va scrie

în răspuns identificatorul de corelație pe canalul de răspuns. Clientul va folosi acest identificator pentru a corela răspunsul cu cererea. În aceste cazuri este mai simplu să se folosească un mecanism IPC care oferă în mod implicit funcționalitățile menționate.

Recent, limbajele pentru definirea unei interfețe pentru API-urile de tip REST au devenit din nou populare. Există câte opțiuni precum RAML sau Swagger. Câteva IDL - Interface Description Language - precum Swagger permit definirea formatului mesajelor de cerere și răspuns. Altele precum RAML necesită folosirea unei alte componente precum JSON Schema. Pe lângă descrierea API-urilor, limbaje IDL oferă unelte pentru generarea funcționalităților pe partea de client sau server plecând de la definirea interfeței.

Apache Thrift este o alternativă interesantă la REST. Este o colecție de biblioteci pentru scrierea de clienți și servere de tip RPC agnostică față de limbaje. Se poate folosi compilatorul pentru a genera părți ale clientului și serverului într-o multitudine de limbaje precum: C++, Java, Python, PHP, Ruby, Erlang și Node.js.

O interfață Thrift este compusă din unul sau mai multe servicii. Definiția unui serviciu este similară cu o interfață în Java. Este o colecție de metode cu parametri bine definiți. Metodele pot fie returna sau nu să returneze o valoare. Cele care returnează o valoare implementează un stil de interacțiune de tip cerere/răspuns. Clientul așteaptă un răspuns și ar putea arunca o excepție. Metodele unidirectionale corespund notificărilor; serverul nu trimite un răspuns.

Thrift oferă suport pentru o varietate de formate de mesaje precum: JSON, binar și binar compact. Formatul binar este mai eficient decât JSON deoarece este mai rapid de decodat. După cum sugerează și numele, formatul binar-compact este și mai eficient. Formatul JSON, este desigur mult mai facil de citit. Thrift oferă posibilitatea folosirii directe a protocoalelor de transport TCP sau HTTP. Folosirea directă a TCP-ului este mult mai eficientă decât HTTP, însă HTTP este mai ușor de citit de către firewall-uri, browsere și oameni.

4.5 Identificarea serviciilor

Pentru apelarea unui REST (Representational State Transfer) API este nevoie de lansarea unei cereri, care să includă locația de rețea a serverului (adresa IP și portul). În cazul unei aplicații monolit care rulează pe un server fizic, aceste informații sunt relativ statice. Ele ar putea fi citite de exemplu dintr-un fisier de configurare care este actualizate periodic. În schimb, într-o aplicație modernă ce rulează în cloud și este bazată pe microservicii acest lucru este mult mai dificil de rezolvat după cum arată schema de mai jos:

Instanțelor serviciilor le sunt alocate în mod dinamic locații de rețea. În plus, instanțele se vor modifica în mod dinamic datorită auto-scalării, penelor și actualizărilor. În consecință codul client are nevoie de un mecanism mai elaborat de identificare a serviciilor.

Există două categorii de șabloane: identificare pe partea de client (*eng. client-side discovery*) și identificare pe partea de server (*eng. server-side discovery*).

4.5.1 Identificarea serviciilor de către client

Atunci când descoperirea este realizată de către client, acesta este responsabil pentru determinare locațiilor de rețea ale instanțelor de serviciu și pentru a face balansare pe baza acestora. Clientul lansează o cerere către registrul de servicii (*eng. service registry*), care este o bază de date a instanțelor de serviciu. Clientul apoi folosește un algoritm de balansare pentru a selecta unul dintre ele și pentru a lăsa cererea propriu-zisă.

Locația de rețea a unei instanțe de serviciu este înregistrată în registrul de serviciu atunci când este lansată. Este înlăturată atunci când instanța este oprită. Starea instanțelor serviciilor este actualizată periodic folosind un mecanism de tip puls (*eng. heartbeat*).

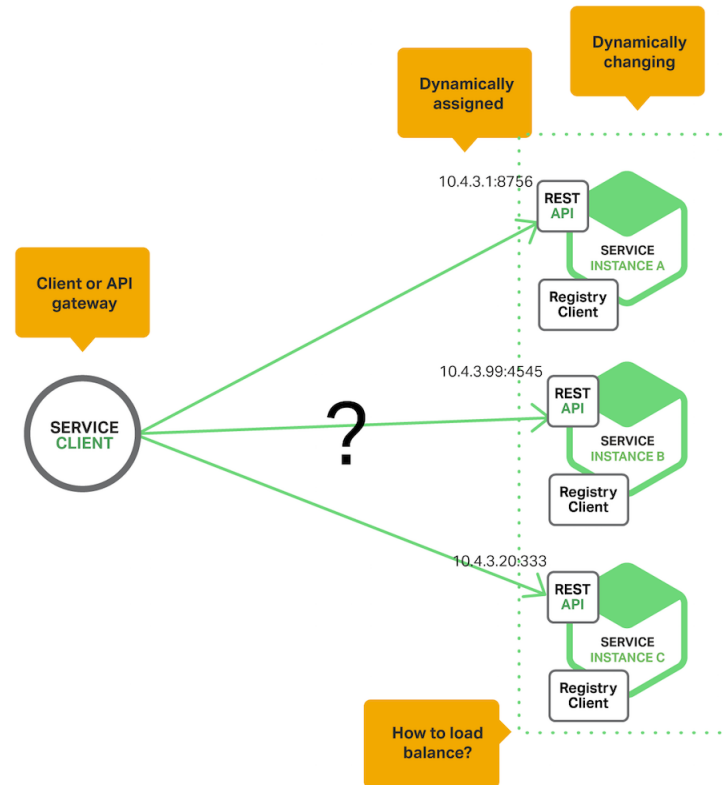


Figura 4.12: Identificarea serviciilor în cloud

Principiul de funcționare al identificării serviciilor de către client funcționează conform schemei de mai jos:

Acest mod de funcționare prezintă o varietate de beneficii și dezavantaje. Se bazează pe un principiu simplu și exceptând registrul serviciilor nu există alte componente mobile. În plus, din moment ce serviciul cunoaște toate instanțele de serviciu disponibile, poate să ia decizii de balansare inteligente, specifice aplicației. Un dezavantaj major este cuplarea clientului cu registrul serviciilor; în acest caz trebuie implementată o logică de descoperire a serviciilor pentru fiecare limbaj de programare și set de biblioteci folosite de către serviciile-client.

4.5.2 Identificarea serviciilor de către server

Cealaltă abordare posibilă este identificarea serviciilor de către server, care funcționează conform schemei de mai jos:

Clientul lansează o cerere către serviciu prin intermediul unui load balancer, care la rândul său interoghează registrul de servicii și rotează fiecare cerere către o instanță a serviciului. Ca și în cazul identificării de către client, instanțele de serviciu sunt gestionate de registrul de servicii.

AWS Elastic Load Balancer (ELB), produs al companiei Amazon este un exemplu de asemenea mecanism. Este folosit în principiu pentru a balansa traficul extern venit din Internet. Totuși poate fi folosit pentru a distribui traficul care este intern unui cloud privat. Un client lansează cereri via ELB folosind numele DNS. Acesta distribui traficul între instanțele înregistrate. Nu există un registru de servicii separat, în schimb toate aceste instanțe sunt înregistrate direct în ELB.

Unele medii care folosesc Kubernetes sau Marathon folosesc un proxy la nivelul fiecărei mașini din cluster. Proxy-ul joacă rolul load-balancer-ului responsabil pentru descoperire de

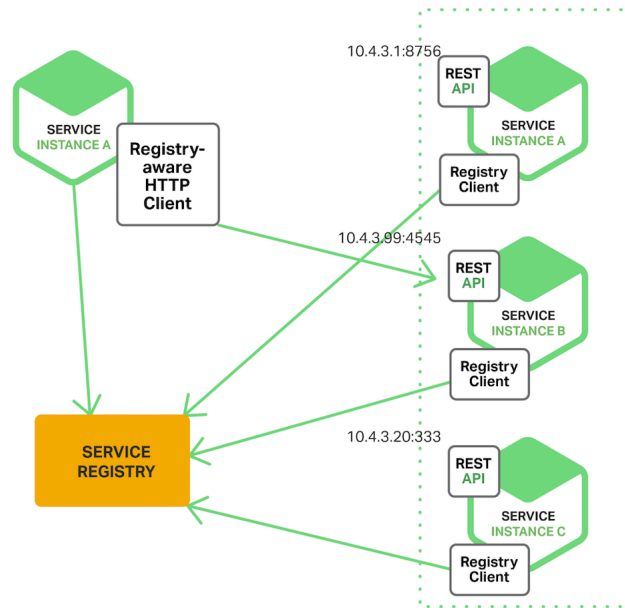


Figura 4.13: Identificarea serviciilor de către client

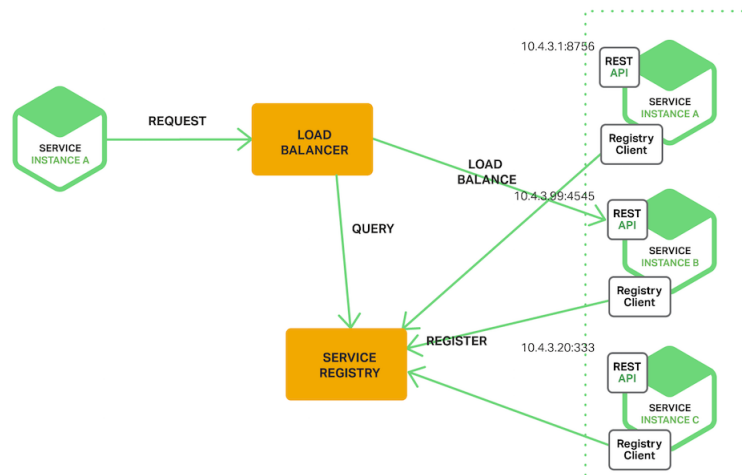


Figura 4.14: Identificarea serviciilor de către server

către server. Pentru a lansa o cerere către un serviciu, un client rutează cererea via proxy folosind adresa IP a hostului și portul serviciului. Apoi proxy-ul transmite mai departe cererea către o instanță disponibilă din cluster.

Acest mod de funcționare prezintă deasemenea avantaje și dezavantaje. Un mare avantaj ar fi faptul că detaliile legate de identificare sunt decuplate de client. Aceștia lansează pur și simplu cereri către load balancer, ceea ce elimină nevoia implementării unei logici de identificare pentru fiecare limbaj de programare sau set de biblioteci folosit de către serviciile client. Există totuși și dezavantaje. Dacă funcționalitatea nu este livrată împreună cu mediul de instalare, este încă o componentă ce trebuie configurată și menținută.

4.5.3 Registrul serviciilor

Registrul serviciilor este o parte cheie din procesul de identificare a serviciilor. Este o bază de date ce conține locațiile de rețea ale instanțelor serviciilor. Acesta trebuie să fie o componentă

cu un grad înalte de disponibilitate și mereu actualizat. Clienți pot stoca în memoria cache locațiile de rețea obținute de la acesta, însă informația va deveni la un moment dat învechită și clienții vor fi nevoiți să reapeleze registrul. În consecință, un registru de servicii reprezintă un cluster de server care folosesc un protocol de replicare pentru a menține consistența.

Un exemplu de astfel de componentă este Eureka. Acesta oferă un API de tip REST pentru înregistrarea instanțelor de serviciu, cât și pentru interogări. O instanță de serviciu se înregistrează printr-o cerere de tip POST. La fiecare 30 de secunde înregistrarea trebuie actualizată printr-o cerere de tip PUT. Înregistrarea este ștearsă fie folosind o cerere HTTP DELETE fie pentru expirarea unui timer. Un client poate obține lista de instanțe înregistrate folosind o cerere de tip HTTP GET.

Alte exemplu de astfel de componente:

- **etcd** - reprezintă o componentă de stocare de tip cheie-valoare cu un grad înalte de disponibilitate folosit pentru stocarea configurațiilor și pentru identificarea serviciilor. Două proiecte notabile care folosesc această componentă sunt Kubernetes și Cloud Foundry.
- **consul** - este o unealtă pentru identificarea și configurarea serviciilor. Oferă un API care permite clienților să se înregistreze și să descopere servicii. Acesta poate efectua verificări pentru determinarea disponibilității serviciului.
- **Apache Zookeeper** - este un serviciu de coordonare a aplicațiilor distribuite foarte răspândit. Inițial a fost un subproiect Hadoop, dar acum a devenit un proiect separat.

Deasemenea, este de notat faptul ca sisteme pentru Kubernetes, Marathon și AWS nu au un registru al serviciilor explicit. În schimb, acesta este o componentă integrată a infrastructurii.

4.5.4 Metode de înregistrare a serviciilor

Așa cum am menționat anterior, instanțele de serviciu trebuie să fie înregistrare și șterse din registrul serviciilor. Există câteva modalități pentru a gestiona aceste două operațiuni. Una este ca serviciile să se înregistreze ele însele, numit și auto-înregistrare. Cealaltă opțiune este ca o componentă a sistemului să gestioneze această, numit și înregistrare terță.

4.5.4.1 Auto-înregistrarea

Folosind auto-înregistrarea, o instanță a serviciului este reponsabilă pentru înregistrare și anularea acesteia din cadrul registrului. Deasemenea, dacă este necesar, instanța serviciului trimite pulsuri pentru a preveni expirarea înregistrării.

Un exemplu bun al acestei abordări este client Eureka. Acesta se ocupă de toate aspectele legate de înregistrarea și înlăturarea unei instanțe în registrul serviciilor. Proiectul Spring Cloud, care implementează o serie de șabloane incluzând identificarea serviciilor, facilitează abonarea unui serviciu la Eureka printr-o simplă anotare `@EnableEurekaClient`.

Un prim avantaj al acestei metode ar fi simplitatea, datorită faptului că nu necesită nici o altă componentă de sistem. Însă, dezavantajul major este cuplarea între instanțe și registrul serviciilor. Aceasta crează necesitatea codării acestei funcționalități la nivelul fiecărui limbaj de programare.

Abordarea alternativă, care decuplează serviciile de registrul serviciilor, este înregistrarea terță.

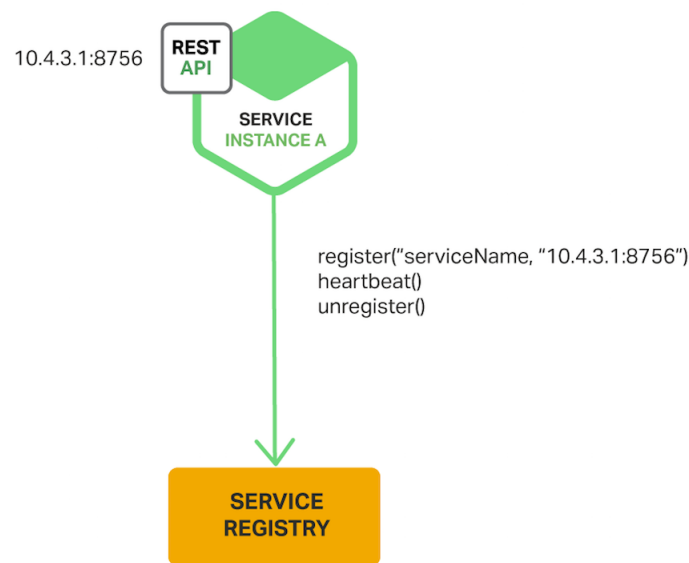


Figura 4.15: Auto-înregistrare

4.5.4.2 Înregistrarea terță

Folosind înregistrarea terță, instanțele serviciului nu mai sunt responsabile pentru înregistrarea în cadrul registrului serviciilor. În schimb, o altă componentă de sistem cunoscută sub numele de registrar al serviciilor (*eng. service registrar*) se ocupă de aceasta. Acesta monitorizează schimbările la nivelul instanțelor fie prin interogări periodice al mediului de instalare fie prin abonarea la evenimente. Atunci când detectează o nouă instanță acesta o înregistrează la nivelul registrului. Deasemenea tot el se va ocupa și de eliminare înregistrării în momentul în care instanța nu mai este disponibilă.

Un exemplu de o astfel de componentă este proiectul open-source *Registrar*. Acesta este capabil să gestioneze instanțele de serviciu care sunt instalate ca și containere Docker. Acesta dispune suport pentru o serie de registrii de servicii precum, **etcd** și **Consul**.

Principalul beneficiu al acestui mod de funcționare este faptul că decuplează serviciile de registrul serviciilor. Nu mai este necesară implementarea logici la nivelul fiecărui limbaj de programare, în schimb procesul este gestionat într-o manieră centralizată de către un serviciu dedicat. Dezavantajul este că, dacă nu este furnizat de mediul de instalare acesta devine încă o componentă ce trebuie configurată și menținută.

Capitolul 5

Implementare practică

Punerea în practică a acestor concepte teoretice a fost realizată cu ajutorul unei aplicații web pentru vizionarea de conținut video pe baza de cont și abonament similar cu platformele deja existente pe piață. Aceasta permite unui utilizator crearea unui cont, alegerea tipului de abonament și în cazul alegerii unui abonament plătit alegerea metodei de plată pentru abonamentul respectiv. În funcție de tipul de cont creat acesta va putea vedea anumite clipuri video.

Aplicația implementează câteva cazuri de utilizare ce asigură o funcționalitate minimală pentru a constitui, ceea ce în literatura de specialitate se numește un produs minim viabil - *eng. minimal viable product*. În cadrul metodologiei Agile un astfel de produs ar fi livrat de o echipa la finalul unui *sprint*.

Pentru a respecta principiile enunțate în partea teoretică aplicația a fost împărțită în mai multe microservicii, care vor constitui noduri ale aplicației, după cum urmează:

- service registry - componenta care este responsabilă pentru menținerea listei de instanțe ale fiecărui serviciu.
- gateway - punctul principal de comunicare între utilizator și sistem
- serviciul de gestiune al utilizatorilor
- serviciul de gestiune al rolurilor utilizatorilor
- serviciul de gestiune al conținutului video
- serviciul de gestiune al comentariilor
- instanțe ale bazelor de date aferente fiecărui serviciu

Aplicația putea fi constituită dintr-un singur bloc, ceea ce ar fi conferit unitate acesteia și poate o arhitectura mai simplă, dar i-ar fi redus din flexibilitate. În esență, modul în care dezvoltarea s-a realizat a fost plecând de la implementarea funcționalității dorite în cadrul gateway-ului. Odată satisfăcută cerința apelul local pentru realizarea acesteia a fost înlocuit cu un apel distant către un serviciu REST. Apoi funcționalitatea a fost retestată pentru asigurarea consistenței la nivelul logic al acesteia.

Arhitectura aplicației exemplificată în schema de mai jos demonstrează gradul mare de decuplare al componentelor acesteia. Nu doar că acestea sunt decuplate, dar comunicarea se face prin interfețe bine definite ce folosesc protocoale cu specificații clare, respectiv HTTP.

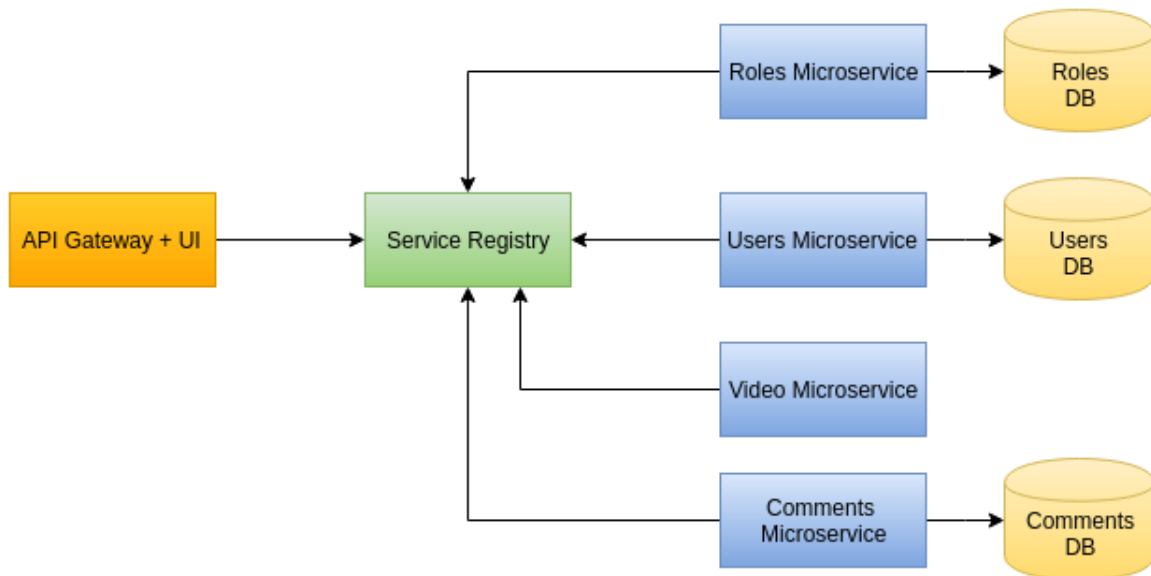


Figura 5.1: Arhitectura aplicației

Mai sus sunt reprezentate blocurile funcționale ale aplicației. Microserviciile nu se cunosc între ele, altfel spus niciodată un microserviciu nu va apela un alt microserviciu. Ele expun funcționalități prin intermediul unui API de tip REST care urmează a fi consumat. Punctul central al acestei arhitecturi îl reprezintă "Service registry", deși interacțiune cu exteriorul se realizează prin intermediul "API Gateway". Acesta este componentă foarte importantă deoarece toate celălalte servicii îl vor apela pentru a se putea înregistra. În acest caz și API Gateway-ul va stabili o conexiune cu acesta în primul rând pentru a se înregistra și apoi pentru a obține informații legate de celalate servicii și a efectua apeluri.

Nodurile din partea dreapta a schemei sunt construite într-o manieră similară. Acestea nu mențin nici un fel de informație legată de sesiunea utilizatorului care este conectat. Pe de altă parte API Gateway-ul este cel care utilizat funcționalitățile expuse de servicii implementează cazuri de utilizare, precum crearea unui nou cont sau autentificare și menține informații legate de sesiune.

5.1 Descrierea componentelor

Fiecare din componentele din componentele sistemului are anumite caracteristici, în funcție de rolul pe care îl îndeplinește în cadrul cazurilor de utilizare. Astfel, privind schema de arhitectura a aplicației putem identifica faptul că modul în care microserviciile au fost construite este diferit de modul în care a fost construit "Service Registry" și "API Gateway". O caracteristică comună a microserviciilor din partea dreaptă ar fi faptul că aceste nu dispun de o interfață grafică, ele fiind proiectate ca simple servicii REST ce expun funcționalități pentru efectuarea de operațiuni de bază legate de entitățile pentru care sunt construite. În literatura de specialitate acestea se numesc și repository - depozit.

5.1.1 Microserviciile de tip depozit

Microserviciile de tip depozit sunt cele care implementează logica de business asupra entităților sistemului. O specificitate a acestora este faptul că acestea se conectează la un mediu de stocare persistent. Acesta poate fi o bază de date sau o locație pe un disc. În cazul de față trei dintre servicii se contează câte o bază de date, iar cel de-al patrulea are acces la un disc. Ceea ce poate fi remarcat în cadrul microserviciilor, spre deosebire de o abordare monolit este faptul că microserviciile se contează la baze de date diferite. Aceasta are două implicații. Prima este că administrarea bazelor se poate face de către echipe diferite. A doua presupune o libertate în alegerea sistemului de gestiune a bazei de date - SGBD, sistem care poate fi ales pentru a răspunde mai bine cerințelor serviciului respectiv. Deși în schemă fiecare microserviciu este reprezentat ca un singur bloc, în realitate pot fi mai multe instanțe ale acestuia. În cazul acesta pentru a putea asigura coerența la nivelul datelor, toate instanțele se vor conecta la aceeași bază de date.

O altă particularitate a acestor componente este faptul că ele nu mențin nici un fel de informație legată de stare. Plecând de la informația primită ca și dată de intrare ele vor putea furniza mereu același rezultat, fiecare caz de utilizare implementat de acestea fiind complet independent față de celălalt. În vederea proiectării sistemelor auto-scalabile acesta este un avantaj, deoarece permite crearea și distrugerea instanțelor unui serviciu, fără riscul de a pierde vreunul fel de informație de stare. Putem spune că operațiunile efectuate de aceste componente sunt atomice.

Din punct de vedere tehnologic, acestea sunt realizate folosind limbajul de programare Java. Motivul alegerii acestuia este nivelul de competență ridicat și popularitatea acestuia în rândul dezvoltatorilor. Această popularitate se traduce la rândul ei printr-o abundență de biblioteci software ce pot accelera procesul de dezvoltare.

Pe parcursul acestei lucrări s-a făcut adesea referire în mod interschimbabil la serviciu, microserviciu și instanțe ale unui serviciu. Ne putem referi la serviciu, sau microserviciu - acesta fiind o restrângere din punct de vedere al funcționalității serviciului, ca la acel bloc al sistemului capabil să îndeplinească anumite reguli de business. Îl vom putea considera precum un șablon sau dacă facem o paralelă cu programarea orientată pe obiecte acesta ar fi comparat cu o clasă. O instanță a serviciului este o materializare a serviciului, efectiv serverul care îndeplinește funcțiile descrise.

Un serviciu poate fi compus din mai multe instanțe de serviciu - din mai multe servere. În abordarea clasică, monobloc, echipa responsabilă cu gestiunea infrastructurii ar oferi echipei de dezvoltare servere pe care aplicațiile să poate fi instalate. Modalitatea aceasta de lucru este greoaie și predispusă la erori. Furnizare și configurare unor astfel de servere poate dura și luni și chiar și în acest caz pot apărea discrepanțe între configurare serverului folosit de dezvoltator și cel furnizat.

O abordare modernă este crearea de aplicații "self-contained" în care serverul este inclus în aplicație. Aceasta elimină îndoilele legate de consistența configurărilor între cele două medii. În acest caz serverul devine parte a artefactelor ce vor fi livrate.

Astfel întregul sistem a fost gândit ca fiecare aplicație să conțină propriul server. Una din soluțiile cele mai populare pentru acest tip de server este SpringBoot. Acesta este un proiect al firmei Pivotal, cea care susține și librăria devenită standard în industrie, Spring. SpringBoot are la bază un server Apache Tomcat. Acesta face dezvoltarea foarte facilă și rapidă deoarece Apache Tomcat este un simplu Servlet Container, opțiune mult mai puțin pretențioasă în materie de resurse decât un server aplicativ.

Pentru folosirea SpringBoot este necesară adăugarea acestuia în fisierul de configurare a sistemului de gestiune a librăriilor proiectului Apache Maven.

Se poate observa din extrasul de mai sus modul de împachetare al artefactului este `.jar` și


```

<groupId>ro.andreimihaescu</groupId>
<artifactId>video-app-gateway</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>video-app-gateway</name>
<description>Gateway for video app project</description>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.0.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

```

Figura 5.2: pom.xml - SpringBoot

nu .war așa cum ar fi de așteptat în cazul unei aplicații de tip web. Aceasta datorită faptului că pentru a rula aplicația vom putea pur și simplu să rulăm aplicația precum orice altă aplicație banală java folosind comanda `java -jar`. În momentul în care această comandă va fi lansată serverul Tomcat va fi lansat cu aplicația rulând.

```

andreih@homebase ~/IdeaProjects/video-app-gateway/target $ java -jar eurekaclient-remotecall-0.0.1-SNAPSHOT.jar
2016-08-30 20:55:07.963 INFO 19289 --- [main] o.s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext
2016-08-30 20:55:08.160 INFO 19289 --- [main] f.a.AutowiredAnnotationBeanPostProcessor : JSR-330 'javax.inject.Inject' annotation found and supported by all BeanPostProcessors
2016-08-30 20:55:08.184 INFO 19289 --- [main] trationDelegatesBeanPostProcessorChecker : Bean 'configurationPropertiesRebinderAutoConfiguration$EnhancerBySpringCGLIB$$ea78f60e' is not eligible for getting processed by all BeanPostProcessors (for example: not eligible for autowiring)
2016-08-30 20:55:08.308 INFO 19289 --- [main] f.a.EurekaClientRemotecallApplication : No active profile set, falling back to default profile
2016-08-30 20:55:08.319 INFO 19289 --- [main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext
2016-08-30 20:55:08.906 WARN 19289 --- [main] o.s.c.a.ConfigurationClassPostProcessor : Cannot enhance @Configuration bean definition 'refresh' with a non-static @Bean method with a BeanDefinitionRegistryPostProcessor return type: Consider declaring such methods as 'static'
2016-08-30 20:55:08.966 INFO 19289 --- [main] o.s.c.a.t.TomcatEmbeddedServletContainer : BeanFactory id=043a73f5-b48f-34e5-9e8f-ca5b5ce761b6
2016-08-30 20:55:08.973 INFO 19289 --- [main] f.a.AutowiredAnnotationBeanPostProcessor : JSR-330 'javax.inject.Inject' annotation found and supported by all BeanPostProcessors
2016-08-30 20:55:09.133 INFO 19289 --- [main] trationDelegatesBeanPostProcessorChecker : Bean 'org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration$EnhancerBySpringCGLIB$$ea78f60e' is not eligible for getting processed by all BeanPostProcessors
2016-08-30 20:55:09.454 INFO 19289 --- [main] o.s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8081 (http)
2016-08-30 20:55:09.466 INFO 19289 --- [main] o.apache.catalina.core.StandardService : Starting service Tomcat
2016-08-30 20:55:09.469 INFO 19289 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.4
2016-08-30 20:55:09.549 INFO 19289 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext

```

Figura 5.3: Lansare instanță microserviciu

Se poate observa din imaginea de mai sus că folosim versiunea 1.4.0.RELEASE a SpringBoot, care la rândul său folosește Apache Tomcat versiunea 8.5.4. Pentru lansarea în modul acesta a oricărei aplicații ce folosește SpringBoot este necesar un minim de configurare. Acesta se rezumă la folosirea unei anotări asupra unei clase `@SpringBootApplication` și aplearea metodei static `run` din clasa `SpringApplication` în punctul de intrare al oricărui program Java public static void main().

```

@SpringBootApplication
public class EurekaClientRemotecallApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaClientRemotecallApplication.class, args);
    }
}

```

Figura 5.4: Lansare instanță microserviciu

Unul din avantajele folosirii SpringBoot este faptul că acesta dispune de pachete de librării grupate pentru a îndeplini o anumită funcționalitate. De exemplu conectarea la baza de date și

folosirea acestora într-un mod facil programatorilor Java se face, în mod tradițional folosind o serie de librării, care la rândul lor pot depinde de alte librării care pot intra în conflict. În plus, dacă aceste nu sunt în conflict una cu cealaltă pot intra în conflict cu librăriile furnizate de server. Așadar SpringBoot simplifică mult această sarcină prin furnizarea unor pachete testate, gata a fi folosite.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>9.4-1206-jdbc42</version>
</dependency>
```

Figura 5.5: Lansare instanță microserviciu

Mai sus sunt adăugate două "dependențe", prima care oferă toate funcționalitatea pentru accesarea entităților din baza de date și cea de-a doua care oferă motorul pentru conectarea la baza de date. Acestea împreună cu o configurație minimală în fișierul de proprietăți a aplicației vor realiza conexiunea.

```
spring.datasource.url= jdbc:postgresql://${database.connection}/application
spring.datasource.username=application
spring.datasource.password=application
```

Figura 5.6: Lansare instanță microserviciu

Pentru a facilita portarea aplicației pe mai diferite medii - dezvoltare, testare, producție - adresa bazei de date este luată ca și variabilă de mediu.

Plecând de la aceste librării se poate crea un microserviciu de tip depozit după cum urmează - exemplificarea va fi realizată folosind serviciul de gestiune a comentariilor. Se pleacă de crearea unei clase care reprezintă entitate ce va fi stocată în baza de date. Folosind anotări specifice aceasta este marcată ca entitate (`@Entity`), după care este specificat numele tabelului care va fi creat în baza de date prin `@Table`. Mai departe se pot crea interogări specifice folosind `@NamedQueries`. În final clasa va arăta ca în imaginea de mai jos:

```
@Entity
@Table(name = "comments")
@NamedQueries({
    @NamedQuery(name = "findCommentByMovieId", query = "from Comment c where c.movieId = :movieId")
})
public class Comment {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String userName;
    private Long movieId;
    private String text;

    public Comment() {
    }

    public Comment(String userName, Long movieId, String text) {
        this.userName = userName;
        this.movieId = movieId;
        this.text = text;
    }

    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }

    public String getUserName() { return userName; }
```

Figura 5.7: Clasă de tip entitate

Pe lângă anotările prezentate mai sus se poate observa `@Id` și `@GeneratedValue` folosite deasupra câmpului `id`. Acestea sunt folosite pentru marca acest câmp ca fiind cheie a tabelului. Celălalte câmpuri nu mai conțin niciun fel de anotare deoarece este folosit principiul "convention over configuration" care aplicat în acest caz va crea în mod automat coloane în tabelă.

Folosirea acestor anotări împreună cu o linie adițională în fișierul de configurare vor permite crearea și actualizarea tabelului în baza de date. Configurarea specifică tipul de DDL - Data Description Language - specific Spring care va fi folosit în cadrul conexiunii la baza de date.

```
spring.jpa.hibernate.ddl-auto=create-drop
```

Figura 5.8: Configurarea DDL

Continuând în sus pe scara arhitecturală, folosind această clasă model vom crea o altă clasă ce expune funcționalitățile de crearea, modificare și ștergere. În mod tradițional asigurarea acestor funcționalități se face prin crearea mai multor clase responsabile pentru gestiunea tranzacționabilității interogărilor către baza de date, însă folosind Spring totul se poate face în mod declarativ prin polimorfism extinzând clasa `JpaRepository<T, ID>`. Aceasta folosește "generics" pentru a putea fi aplicată oricărei entități.

```
CommentRepo
package ro.andreimihaescu.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import ro.andreimihaescu.model.Comment;

public interface CommentRepo extends JpaRepository<Comment, Long> , CommentRepoCustom{
}
```

Figura 5.9: Configurarea DDL

Interfața mai extinde o altă interfață care nu conține decât prototipul unei funcții cu același nume ca și interogarea declarată în clasa model.

```
CommentRepoCustom findCommentByMovieId()
package ro.andreimihaescu.repository;

import ...

public interface CommentRepoCustom {

    List<Comment> findCommentByMovieId(Long movieId);
}
```

Figura 5.10: Configurarea DDL

Astfel, doar cu aceste două interfețe foarte simple avem acces la o multitudine de operațiuni care vor persista entități în baza de date, ce vor putea fi acesate dintr-un nivel superior al aplicației - nivelul de serviciu. Acesta componentă va fi injectată folosind anotarea `Autowired` ceea ce implică faptul că ne așteptăm ca aceasta să existe în contextul administrat de Spring.

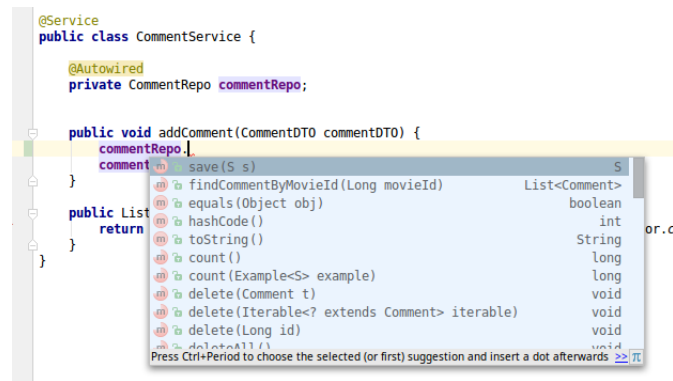


Figura 5.11: Operațiuni legate de baza de date

La nivelul serviciu vor implementate reguli de business ce vor altera datele obținute din baza de date înainte de a le trimite mai departe la la controller.

Ultima componentă a microserviciu este controller-ul, cel care va expune informațiile agregate și prelucrate de serviciu prin intermediul unui API de tip REST.

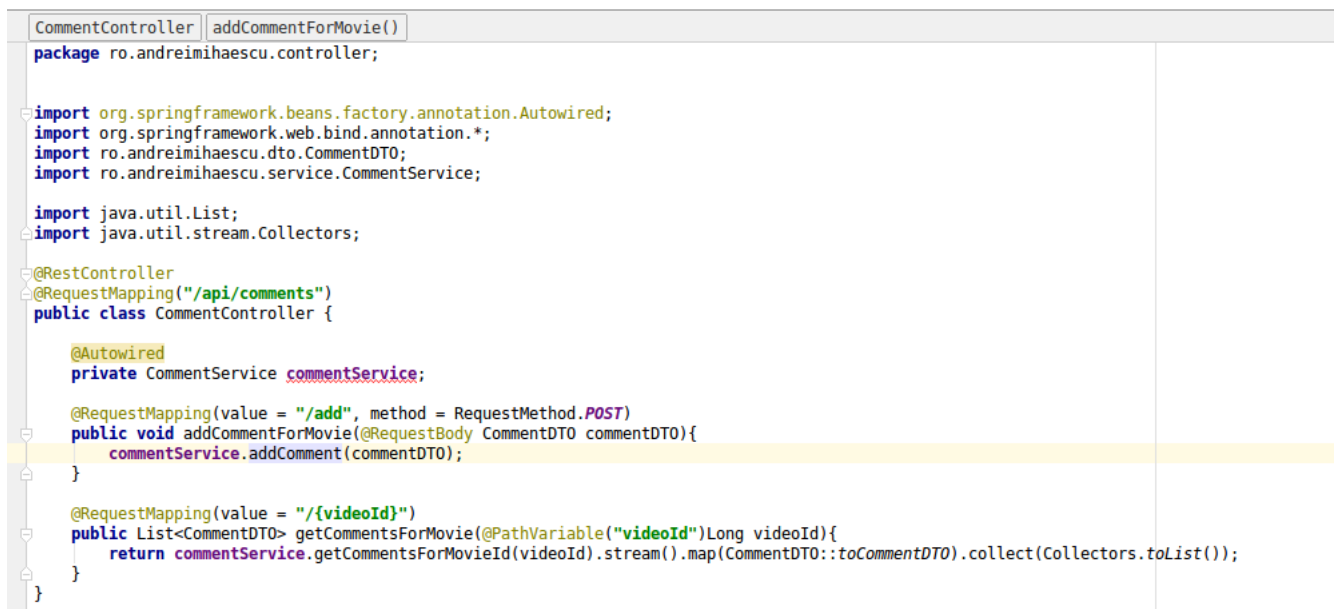


Figura 5.12: Expunerea serviciilor prin intermediul unui REST API

5.1.2 Service Registry

Un punct cheie al microserviciilor este funcționalitatea de înregistrarea și de obținere a informațiilor legate de instanțe de serviciu active. Acestea se realizează prin intermediul protocolului de *Service Discovery*. Spre deosebire de protocoalele de rețea acesta nu cuprinde o specificație formală clară. El este implementat de către compania Netflix în cadrul unei componente open-source denumită *Eureka* care a fost integrată cu Spring.

Această componentă cuprinde două părți : server și client. Partea de server va asculta pentru cereri de înregistrare și va răspunde cererilor de interogare pentru instanțele unui anumit serviciu. Partea de client presupune doar acțiunea de înregistrare și de reactualizare a acesteia.

Datorită bunei integrări dintre Spring și proiectul Eureka partea de server se realizează foarte simplu prin folosirea unei simple anotări `@EnableEurekaServer`

```

EurekaServerApplication
package ro.andreimihaescu;

import ...

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

    public static void main(String[] args) { SpringApplication.run(EurekaServerApplication.class, args); }
}

```

Figura 5.13: Cod pentru server Eureka

Folosind doar această anotare și lansând aplicația se crează toată funcționalitatea necesară. În plus aceasta este expusă prin intermediul unui portal web.

The screenshot shows the Spring Eureka web interface running on localhost:8761. The interface has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into several sections:

- System Status:** A table showing environment details (test, default) and system metrics (Current time: 2016-08-30T22:56:00 +0300, Uptime: 04:05, Lease expiration enabled: true, Renew threshold: 10, Renew last min: 16).
- DS Replicas:** A section showing the local data center as 'localhost'.
- Instances currently registered with Eureka:** A table listing four registered services:

Application	AMIs	Availability Zones	Status
ROLE-SERVICE	n/a (1)	(1)	UP (1) - 192.168.100.9:role-service:8083
USER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.100.9:user-service:8082
VIDEO-APP-GATEWAY	n/a (1)	(1)	UP (1) - 192.168.100.9:video-app-gateway:8081
VIDEO-SERVICE	n/a (1)	(1)	UP (1) - 192.168.100.9:video-service:8084
- General Info:** A section at the bottom for additional information.

Figura 5.14: Intefată web pentru server Eureka

5.1.3 API Gateway

Dintre toate componentele API Gateway-ul este cea mai complexă în materie de logică de business inclusă. Mai exact, aceasta include părți de logică aparținând tuturor microserviciilor și implementează cazurile de utilizare. Totuși ca și complexitate tehnică este mai simplă decât celelalte componente deoarece tot ceea ce realizează aceasta este să apeleze serviciile și să agregheze rezultatele obținute de la acestea. Teoretic la acest nivel nu ar mai fi nevoie de nici un fel de procesare, deoarece aceasta a fost deja implementată la nivelul microserviciilor.

Ca și structură a codului este foarte similar organizată cu precedentele componente, folosind o arhitectură stratificată. Diferența majoră este ca la nivelul de depozitare apelurile către baza de date sunt înlocuite cu apeluri către servicii.



```

UserRepository
package ro.andreimihaescu.repository;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.web.client.RestTemplate;
import ro.andreimihaescu.dto.UserRequest;

@Repository
public class UserRepository {

    @Autowired
    private RestTemplate restTemplate;

    public Long createUser(UserRequest userRequest) {
        return restTemplate.postForEntity("http://user-service/users/create", userRequest, Long.class).getBody();
    }

    public Boolean authenticateUser(UserRequest userRequest){
        return restTemplate.postForEntity("http://user-service/users/authenticate", userRequest, Boolean.class).getBody();
    }

}

```

Figura 5.15: API Gateway - clasa depozit

Se poate observa introducerea unei noi clase `RestTemplate` anotată cu `@Autowired`. Aceasta este cea care va fi folosită pentru a lansa apelurile către serviciile înregistrate la "Service Registry". Încă o dată, datorită bunei integrări a proiectului Eureka cu Spring, prin intermediul acestei clase nu doar că vom putea lansa apeluri către servicii, dar vom putea chiar distribui în mod echitabil cererile între instanțele de serviciu.

Pe lângă implementarea apelurilor către serviciile distante, API Gateway-ul mai implementează și o interfață grafică menită să permită utilizatorilor o comunicare facilă cu sistemul. Aceasta a fost realizată folosind tehnologii front-end precum AngularJS și Bootstrap, librării de javascript devenite standard în industrie.

Interfața grafică este concepută ca Single Page Application - SPA. Aceasta presupune faptul că întreaga parte de front-end este încărcată de la prima accesare. Tot scheletul necesar redării paginilor este trimis pe client și apoi datele sunt obținute prin apeluri REST către api-ul expus de server.

Se poate observa că în urma primului apel serverul răspunde cu pagina setată ca index care conține o serie de dependențe către alte fișiere. Ulterior acestea sunt încărcate astfel încât aplicația va rula în browser-ul utilizatorului.

Name	× Headers	Preview	Response	Cookies	Timing
localhost			<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="UTF-8"> 5 <title>Title</title> 6 <link rel="stylesheet" href="webjars/angular-material/1.1.0/angular-material.css" /> 7 <link rel="stylesheet" href="css/style.css" /> 8 </head> 9 <body ng-app="videoApp"> 10 <div ng-controller="indexCtrl"> 11 <md-content class="md-padding"> 12 <md-nav-bar md-selected-nav-item="currentNavItem" aria-label="navigation-links"> 13 <md-nav-item ng-show="hideMain()" name="login" md-nav-href="#/login">Login</md-nav-item> 14 <md-nav-item ng-hide="hideMain()" name="logout" md-nav-click="logout()">Log out</md-nav-item> 15 <md-nav-item ng-show="hideMain()" name="signup" md-nav-href="#/signup">Signup</md-nav-item> 16 <md-nav-item ng-hide="hideMain()" name="main" md-nav-href="#/main">Main</md-nav-item> 17 <md-nav-item ng-hide="hideMain()" name="view" md-nav-href="#/view">View</md-nav-item> 18 </md-nav-bar> 19 </md-content> 20 </div> 21 <div style="width: 80%; margin: auto;" id="content" ui-view></div> 22 <script src="webjars/angularjs/1.5.6/angular.js"></script> 23 <script src="webjars/angularjs/1.5.6/angular-animate.js"></script> 24 <script src="webjars/angularjs/1.5.6/angular-aria.js"></script> 25 <script src="webjars/angularjs/1.5.6/angular-messages.js"></script> 26 <script src="webjars/angular-material/1.1.0/angular-material.js"></script> 27 <script src="webjars/angularjs/1.5.6/angular-resource.js"></script> 28 <script src="webjars/angularjs/1.5.6/angular-route.js"></script> 29 <script src="webjars/angularjs/1.5.6/angular-sanitize.js"></script> 30 <script src="webjars/angular-ui-router/0.2.18/angular-ui-router.js"></script> 31 32 <script src="js/app.js"></script> 33 <script src="js/services/authService.js"></script> 34 <script src="js/services/sharedScope.js"></script> 35 <script src="js/controllers/loginController.js"></script> 36 <script src="js/controllers/signupController.js"></script> 37 <script src="js/controllers/indexController.js"></script> 38 <script src="js/controllers/mainController.js"></script> 39 <script src="js/controllers/viewController.js"></script> 40 41 </body> 42 </html> </pre>		
angular-material.css					
style.css					
angular.js					
angular-animate.js					
angular-aria.js					
angular-messages.js					
angular-material.js					
angular-resource.js					
angular-route.js					
angular-sanitize.js					
angular-ui-router.js					
authService.js					
app.js					
sharedScope.js					
loginController.js					
signupController.js					
indexController.js					
mainController.js					
viewController.js					
login.html					

Figura 5.16: Livrarea aplicației front-end

În ceea ce privește partea de back-end, API Gateway-ul este construit similar cu celălalte microservicii ca și organizare a pachetelor. Una dintre diferențele notabile este faptul că aceasta nu mai dispune de clase de tip depozit apelurile către serviciile distante făcându-se direct de la nivelul de servicii prin injectare obiectului de tip `RestTemplate`.

```

@Service
public class CommentsService {

    @Autowired
    private RestTemplate restTemplate;
}

```

Figura 5.17: Injectare RestTemplate

5.1.4 Baza de date

O ultimă componentă extrem de importantă în cadrul oricărei aplicații este baza de date. În cadrul microserviciilor ce folosesc acest mod de stocare a datelor ele dispun de instanțe individuale. Fiecare componentă se conectează la o instanță de PostgreSQL ce rulează ca în cadrul unui container Docker. Pentru rularea acestor s-a folosit un utilizator Docker și anumite `docker-compose`. Aceasta permite rularea mai multor containere în același timp prin crearea unui fișier de configurare.


```

version: '2'
services:
  db-users:
    image: postgres
    environment:
      POSTGRES_PASSWORD: application
      POSTGRES_USER: application
    ports:
      - 5432:5432
  db-roles:
    image: postgres
    environment:
      POSTGRES_PASSWORD: application
      POSTGRES_USER: application
    ports:
      - 5433:5432
  db-comments:
    image: postgres
    environment:
      POSTGRES_PASSWORD: application
      POSTGRES_USER: application
    ports:
      - 5434:5432

```

Figura 5.18: Fișier configurare docker-compose

5.2 Funcționarea unui caz de utilizare

Pentru a putea înțelege mai bine funcționarea aplicației și avantajele folosirii acestui tip de arhitectură, cel puțin teoretic este necesară exemplificarea unui caz de utilizare ce folosește mai multe microservicii.

Prin modul în care a fost concepută aplicația crearea unui utilizator implică apelarea atât a serviciului responsabil de gestiunea utilizatorilor cât și cel responsabil de gestiunea rolurilor.

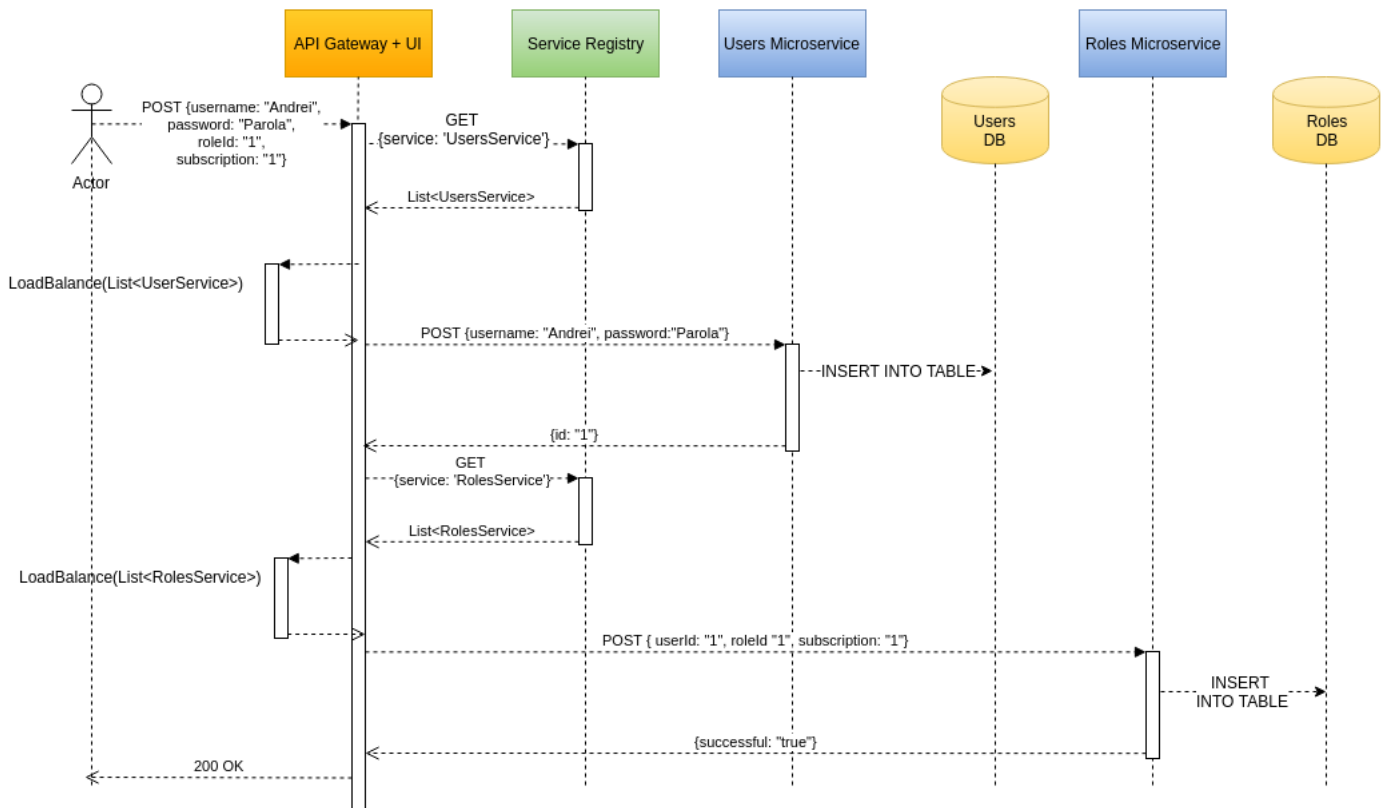


Figura 5.19: Caz de utilizare: crearea unui utilizator

Cazul de utilizare presupune ca aplicația a pornit cu succes și că cele două servicii s-au conectat la "Service Registry".

Utilizatorul prin intermediul interfeței grafice completează câmpurile necesare pentru creare

unui cont, după care la apăsarea pe buton o cerere de tip POST este lansată către gateway. Gateway-ul interceptează cererea și caută să obțină, în primă fază instanțe ale serviciului de gestiune a utilizatorilor de la service registry. Odată obținută lista de instanțe, apelează algoritmul de load-balancing și lansează mai departe cererea către microserviciu. În momentul în care scrierea în baza de date s-a efectuat cu succes, microserviciul răspunde cu identificatorul utilizatorului. Plecând de la această informație, gateway-ul cere mai departe service registry-ului instanțele de serviciu pentru gestiunea rolurilor. Se aplică din nou algoritmul de load-balancing după care, la obținerea confirmării scrierii în baza de date gateway-ul răspunde utilizatorului cu un mesaj HTTP 200 OK, iar acesta este redirectat către pagina principală.

Figura 5.20: Pagină creare utilizator

După cum se poate vedea în acest caz de utilizare, implementând corect microserviciile obținem o decuplare foarte bună între componentele unui sistem, însă introducem foarte multe dependențe către servicii externe. Dependența de servicii externe expune aplicația la anumite riscuri, cum ar fi faptul că serviciile pot să nu fie disponibile. În acest caz aplicația trebuie protejată. În cadrul microserviciilor această protecție se poate realiza prin implementarea șablonului "Circuit breaker". Aceasta anulează toate apelurile către un serviciu în momentul în care cererile eșuate depășesc un anumit prag.

Demonstrarea implementării acestui șablon oferă oportunitate prezentării unui alt caz de utilizare : obținerea comentariilor pentru un video clip.

5.3 Șablonul circuit breaker

Odata creat contul, utilizatorul poate naviga și vizualiza video clipurile la care are acces. Pe lângă aceasta, el poate adăuga și comentarii care vor fi partajate de toți utilizatorii, după cum se poate vede în imaginea de mai jos:

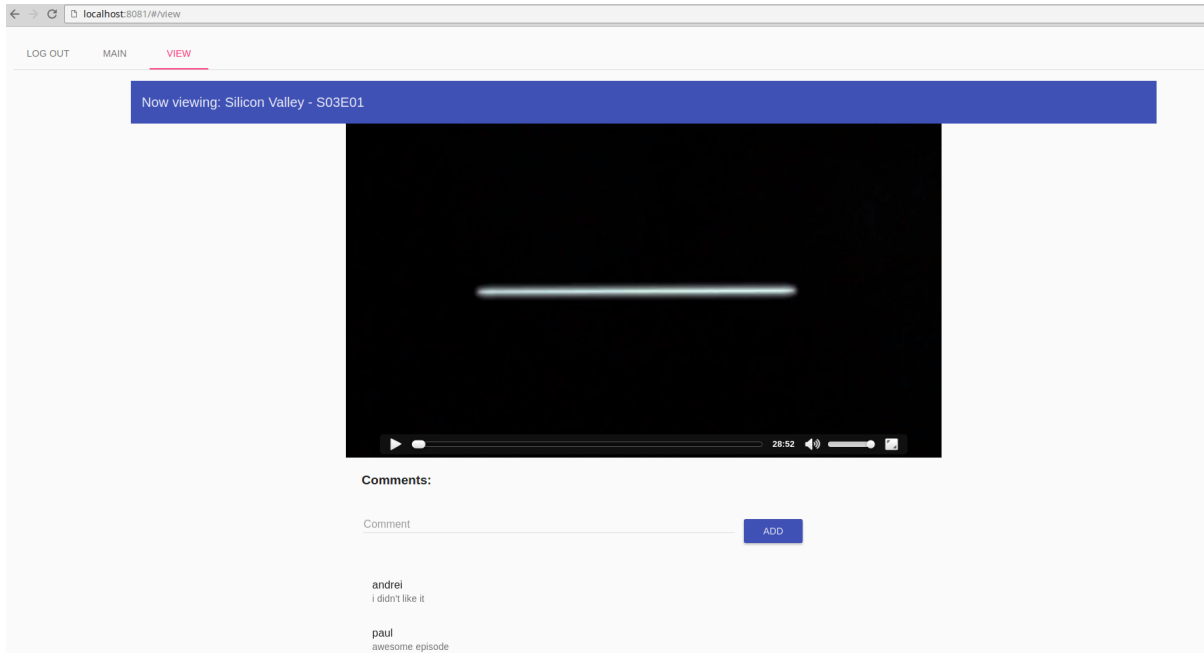


Figura 5.21: Pagină comentarii - serviciu funcțional

Dacă din anumite motive serviciul de gestiune a comentariilor devine indisponibil, atunci este nevoie ca după un anumit număr de cereri, declanșare de noi apeluri către serviciul respectiv să fie scurt-circuitate astfel încât timpul de așteptare să fie redus la minim. În cazul de față, dacă serviciul devine indisponibil singurul comentariu care va fi afișat va fi unul de informare din partea sistemului.

La nivel de implementare acesta se implementează facil folosind un alt proiect open-source al companiei Netflix și anume Hystrix. Prin intermediul unei anotări, `@HystrixCommand`, plasate deasupra metodei care va apela serviciul, aceasta este marcată ca fiind supusă aplicării șablonului. Această anotare mai primește ca și argumente `fallbackMethod` prin intermediul căreia putem specifica metoda ce va fi apelată în caz de pană a serviciului.

```

@HystrixCommand(fallbackMethod = "defaultAddComment")
public Boolean addComment(CommentDTO commentDTO) {
    return restTemplate.postForEntity("http://comments-service/api/comments/add", commentDTO, Boolean.class).getBody();
}

@HystrixCommand(fallbackMethod = "defaultComments")
public List<CommentDTO> getCommentsForVideoId(Long videoId) {
    List<HashMap> list = restTemplate.getForObject("http://comments-service/api/comments/" + videoId, List.class);
    List<CommentDTO> returnList = new ArrayList<>();
    for (HashMap hashMap : list) {
        returnList.add(new CommentDTO(
            (Integer) hashMap.get("id"),
            (String) hashMap.get("userName"),
            (Integer) hashMap.get("movieId"),
            (String) hashMap.get("text")
        ));
    }
    return returnList;
}

public Boolean defaultAddComment(CommentDTO commentDTO){
    return true;
}

public List<CommentDTO> defaultComments(Long videoId){
    return Collections.singletonList(new CommentDTO(1, "System", 0, "Failed to retrieve comments. Adding comments is also unavailable."));
}

```

Figura 5.22: Configurare Hystrix

În urma folosirii acestor anotări se poate observa comportamentul de mai jos:

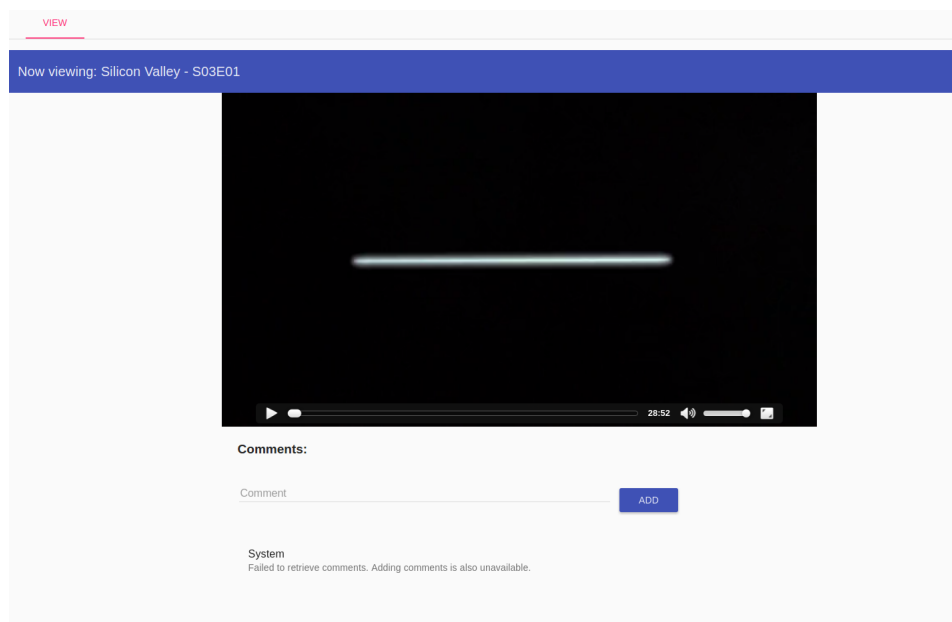


Figura 5.23: Pagină comentarii - serviciu în pană

Capitolul 6

Bibliografie

- Martin Fowler - <http://martinfowler.com/articles/microservices.html>
- Microservice.io - <http://microservices.io/>
- Microsoft Developer Network - <https://msdn.microsoft.com>
- NGINX - <https://www.nginx.com/>
- Building Microservices, Sam Newman, O'Reilly Press