Fine grained architecture (Arhitectura cu granularitate fina)

- mascheaza pauzele prin rularea circular a firelor (fire diferite in cicluri successive).
- Daca numarul ciclurilor de asteptare din fiecare fir este mai mic decat numarul firelor in paralel, atunci pauzele sunt eliminate.
- Este mai eficienta decat granularitatea grosiera

Coarse grained architecture (Arhitectura cu granularitate grosiera)

- ruleaza un fir pana apare un ciclu de asteptare, dupa care comuta la firul urmator.
- Admite un ciclu de pauza inainte de comutarea firelor.
- Functioneaza mai bine decat cea fina atunci cand conditia anterioara nu mai este indeplinita.

Replicarea

- Se refera la partajarea informatiilor astfel incat sa se asigure coerenta intre resursele redundante(software, hardware) pentru imbunatatirea fiabilitatii, tolerantei la erori si accesibilitatii.
- este de doua feluri:
  - **Replicarea activa (**procesarea aceleasi cereri la fiecare replica)
  - **Replicarea pasiva (**procesarea fiecărei cereri pe o singură replică și transferarea rezultatului către celelalte replici)

Middleware

- Este un software care ofera servicii dincolo de cele furnizate de sistemul de operare pentru a permite diferitelor componente ale unui sistem sa comunice si sa gestioneze date.
- Accepta si simplifica sistemele distribuite complexe.

- Acesta include servere web , servere de aplicații , mesagerie și instrumente similare care susțin dezvoltarea și livrarea aplicațiilor. Middleware este deosebit de integrant în tehnologia informației moderne bazată pe XML , SOAP , servicii web și arhitectură orientată spre servicii .

SWOT (Strengths, Weaknesses, Opportunities, and Threats)

S(Puncte tari):

- **costuri reduse**(faptul ca poti folosi chestii de la alte firme, fara sa mai iei tu tot hardul, doar folosesti)
- **modularitate si flexibilitate**(in loc sa proiectezi aplicatii monolit, le poti imparti in module/servicii care pot fi extinse independent)
- **fiabilitate si integritate**(sistemul functioneaza fara defecte intr-un interval de timp si spatiu dat)
- **performanta**(poti lucra in paralel si nu esti limitat de hardul de pe o singura masina)

W(Puncte slabe/ Dezavantaje):

- **lipsa cunostintelor despre starea globala**(de obicei serviciile sunt stateless)
- **lipsa unui timp global**(pot aparea intarzieri datorate lipsei de sincronizare a clock-urilor la nivel global)
- **nedeterminismul**(nu stii mereu cui se trimite)
- **comunicatiile**(trebuie ca conexiunea sa fie buna pentru a nu scadea viteza sistemului din cauza transmisiei datelor)
- **securitatea**(trebuie securitate buna ca sa nu se afle date confidentiale trimise prin retea)

REST (Representational State Transfer)

- arh software
- poate utiliza SOAP fiind model de proiectare arhitectural poate utiliza orice protocol ar dori
- utilizeaza URI pentru a expune logica de afaceri. Poti mapa mai multe operatii pe acelasi URI
- preia securitatea protocolului de transport utilizat
- accepta diverse formate(XML, JSON, HTML)

SOAP (Simple Object Acces Protocol)

- nu poate utiliza REST fiindca e protocol(sau set de standarde)
- utilizeaza interfata serviciului(contractul) pentru a expune logica de afaceri
- defineste propriul standard de securitate
- lucreaza doar cu formatul XML

URI

- secventa unica de caractere ce identifica o resursa logica sau fizica utilizata de tehnologii web.
- structura: schema, autoritate(server+port), cale, interogare, fragment

Enterprise Java Beans

- permite constructia modulara a software-ului de intreprindere
- necesita un server de aplicatii sau un EJB container pt rularea aplicatiilor
- este mai complex decat Java Breans
- programatorul se poate preocupa de logica afacerii ca serverul de aplicatii sa gestioneze servicii ca tranzactii sau exceptii de manipulare
- TIPURI de EJB:

1. session beans:
   - stateless session beans= nu menţin o legătură **client ↔ *bean***
   - stateful session beans= menţin legătura cu clientul apelant (fiecare client cu *stateful session bean*-ul lui)
   - singleton session beans= o singură instanţă disponibilă la nivel de server *enterprise*

2. entity beans= încapsulează funcţionalitate, respectiv maparea datelor dintr-o bază de date sub formă de obiecte

3.message-driven beans=permit procesarea mesajelor in mod asincron

Java Beans

- clase ce incapsuleaza obiecte intr.un singur obiect(numit simplu, bean).Scopul lor este de uşura reutilizarea componentelor software
- clase serializabile, au constructor fara argumente si permit accesul la proprietatile private folosind getter si setter
- este mai simplu decat EJB

POJO (Plain Old Java Object)

- are restrictii impuse de limbajul Java
- nu permite control foarte strict al membrilor
- nu se poate implementa interfata Serializabila
- campurile pot fi accesate direct prin numele lor si pot avea nivel de vizibilitate
- este permisa dar nu obligatorie ultilizarea unui constructor fara argumente
- este recomandat spre utilizarea atunci cand nu se doreste nici un fel de restrictii asupra memrilor iar utilizatorul poate avea acces complet la entitatea creata

Arhitectura multilayer multitier

- multinivel, multistrat
- cadru pentru dezvoltare rapida a aplicatiilor
- instalarea implica hard heterogen(sisteme cu mai multe tipuri de procesor/nuclee)

Servlet

- componenta web
- servletii primesc si raspund solicitarilor din partea clientilor WEB, de obicei pesteHTTP
- contine metode de tratare a fiecarui tip de cerere HTTP:GET->doGet(), POST->doPost() etc

Java RMI (Java Remote Method Invocation)

- este un API Java ce efectueaza metodele de la distanta
- desemneaza doar interfata de programare
- se aseamana cu dokerul adica avem un singur registru din care putem apela diferite metode

RDBMS (Relational database management system)

- conceput special pentru baza de date relationale
- O bază de date relațională se referă la o bază de date care stochează date într-un format structurat, utilizând rânduri și coloane .
- Acest lucru facilitează localizarea și accesarea valorilor specifice în baza de date.
- Este „relațional” deoarece valorile din fiecare tabel sunt legate între ele.
- Tabelele pot fi, de asemenea, legate de alte tabele.
- Structura relațională face posibilă rularea interogărilor pe mai multe tabele simultan.

JDBC (Java Database Connectivity)

- Este un API ce permite aplicatiilor Java să se conecteze și să interogheze o gamă largă de baze de date
- Face posibil ca dezvoltatorul de software sa ruleze interogari SQL intr-o aplicatie Java

JPA (Java Persistence API)

- Este o specificatie a interfetei de programare a aplicatiei Jakarta EE ce descrie gestionarea datelor relationale in aplicatiile Java ale intreprinderii
- Defineste un set de concepte care poate fi implementat de orice tool sau framework

JAR (Java Archive)

- Format de fisier de pachete utilizat de obicei pt agregarea mai multor fisiere de clasa Java
- Asociaza metadate si resurse intr-un singur fisier pt distributie

EAR (Enterprise Application aRchive)

- Este un format de fisier utilizat de Java EE pt ambalarea unuia sau a mai multor module intr-o singura arhiva, a.i. implementarea diferitelor module pe un server de aplicatii sa aiba loc simultan si coerent
- Contine fisiere XML numite si descriptori de implementare care descriu modul de implementare a modulelor
- Un fișier EAR este un fișier JAR standard (și, prin urmare, un fișier Zip) cu o extensie .ear, cu una sau mai multe intrări care reprezintă modulele aplicației și un director de metadate numit META-INF care conține unul sau mai mulți descriptori de implementare.

WAR (Web Application Resource)

- este un fișier folosit pentru a distribui o colecție de JAR -files, JavaServer Pages , Java Servlets , Java clase , XML fișiere, biblioteci de etichete, pagini web statice (fișiere HTML și conexe) și alte resurse care împreună constituie o aplicație web .
- Un fișier WAR poate fi semnat digital în același mod ca un fișier JAR pentru a permite altora să determine de unde a venit codul sursă.

Avantajele fișierelor WAR

- Testarea și implementarea ușoară a aplicațiilor web
- Identificarea ușoară a versiunii aplicației implementate
- Toate containerele Java EE acceptă fișiere WAR
- Structura MVC acceptă fișiere WAR.

Event driven arhitecture

- este o paradigmă de arhitectură software care promovează:
  - Comunicare broadcast
  - reactie eficienta
  - evenimente cu granularitate mica
  - ontologie
  - procesare evenimente complexe
- la aceasta arhitectura un microserviciu publica un eveniment cand are ceva clar care se intampla.

AOP(Aspect Oriented Programming)

- este o paradigmă de programare care are ca scop creșterea modularității, permițând separarea preocupărilor transversale
- Completeaza OOP-ul si ofera alt mod de a gandi structura unui program. Unitatea fundamentala la OOP e clasa iar la AOP e aspectul. E declarativ.
- De exemplu o functie ar putea suna cam asa: "da log la toate apelurile functiilor ale caror nume incep cu set"
- E pentru tanzactii, securitate, logging etc

Portret

- Componenta bazata pe web ce va procesa cereri si va genera continut dinamic
- Utilizatorul final ar vedea portretul ca fiind o zona de continut specializata intr-o pagini Web ce ocupa
- In functie de natura continutului site-ului web care furnizeaza portretul, poti vedea o zona ce primeste diferite tipuri de informatii cum ar fi informatii de calatorie, stiri de afaceri sau chiar vremea locala
- Ofera utilizatorilor capacitatea de a personaliza continutul, aspectul si pozitia portret

JMS (Java Message Service)

- Este standard de mesagerie care permite componentelor bazate pe JAVA EE sa creeze, sa trimita sa primeasca si sa citeasca mesaje
- Permite cominicatiilor distribuite care sunt slab cuplate. Fiabile si asincrone

ORM(Object Relational Mapping)

- tehnica de programare pt conversia datelor intre tipuri de sisteme incompatibile utilizand OOP
- creaza o baza de date de obicei virtuala care poate fi utilizata din limbajul de programare

DAO(Data Acess Object)

- este un pattern care ofera o interfata abstracta pt un anumit tip de baza de date sau alt mecanism de persistenta
- prin maparea apelurilor de aplicatie la stratul de persistenta, DAO ofera anumite operatiuni, fara a expune detalii despre BD

Software program

- Definit ca un set de instructiuni sau un set de modele sau proceduri, ceea ce permite un anumit tip de operatie pe computer

RPC (Remote Procedure Call)

- Apelez o procedura de pe alt calculator, prin retea, ca si cum ar fi local

ESB(Enterprise Service Bus)

- Implementeaza un sistem de comunicatie intre aplicatiile software care interactioneaza reciproc intr.o arhitectura orientata spre servicii

SOA(Service Oriented Arhitecture)

- reprezinta modul de a face componentele software reutilizabile prin intermediul interfetelor de servicii

- interfetele utilizeaza standardele comune de comunicatie in asa fel incat sa poata fi incorporate rapid in noi aplicatii fara a fi nevoie de efectuarea de fiecare data a integrarii profunde

- Tipurile de arhitectura in SOA:
  - component
  - application
  - integration
  - enterprise

- Avem:
  - program software
  - arhitectura tehnologica
  - infrastructura tehnologica

- Fiecare tip de arhitectura are propriile specificatii:

Component arhitecture->application arhitecture(in care grupezi mai multe componente)->
->integration arhitecture(in care lipesti mai multe aplicatii) -> enterprise tehnology arhitecture(in care pui de toate + DB si sist legacy)

Cele 4 tipuri comune de SOA:

- **la nivel de serviciu**
- **la nivel de compozitie**
- **la nivel de inventar**
- **la nivel de intreprindere**

Grid computing

- Practica utilizarii mai multor computere, adesea distribuite geografic, dar conectate prin retele pt a lucra impreuna pentru a indeplini sarcini comune
- Este rulat pe o „grila de date” un set de computere care interactioneaza direct intre ele pt a coordona lucrarile

Cloud computing

- disponibilitate la cerere a resurselor computerelor, in special a stocarii datelor(cloud storage) si putere computationala fara management direct si activ din partea userului.
- Se bazeaza pe partajarea resurselor pt a atinge coerenta si economiile de scara
- model de plata functie de utilizare care permite accesul pe baza de retea, la cerere, convenabil, disponibil, la o grupare de resurse de calcul configurabile(retele, servere, stocare, aplicatii)

IAAS(Infrastructure As A Service)

- producatorul manageriaza de la OS in jos, si tu de la OS in sus(mult mai convenabil)

SAAS(Software As A Service)

- Se ocupa provider-ul de tot

FAAS(Function As A Service)

- producatorul manageriaza de la runtime in jos, tu doar aplicatiile si data
- nu e nevoie de un server care sa lucreze mereu in spate
- are si mecanisme de catching si overall
- Exemple:
  - API Gateway
  - Funciton Watchdog
  - Prometheus
  - Swarm
  - Kubernetes
  - docker

- **XaaS(Orice ca Serviciu)**

  - Storage
  - DB
  - Communication
  - Network
  - Monitoring
  - Testing
  - HPC
  - Human
  - Process
  - Information
  - Identity
  - Application
  - Integration
  - Governance
  - Security
  - Backup

Data center tiers

- Sistem utilizat pt a descrie tipuri specifice de infrastructuri de centre de date intr.un mod consecvent
- Nivelul 1 este infrastructura cea mai simpla, in timp ce nivelul 4 este cel mai complex si are cele mai multe componente redundante
- Fiecare nivel include componentele solicitate de toate nivelele anterioare

Load balancer

- Procesul de distribuire a unui set de task-uri la mai multe resurse(PC-uri) a.i. sa se faca procesarea lor mai eficienta
- Poate optimiza timpul de raspuns pentru fiecare task, evitand supraincarcarea inegala a nodurilor de calcul, in timp ce alte noduri de calcul sunt lasate inactive

Vertical Scaling

- cererile vin printr-un front-end, sunt preluate de un Load Balancer si apoi serverul hosteaza o copie completa a aplicatiei
- dai mai multe resurse unui singur calculator pt cresterea performantei

Horizontal Scaling

- scalare prin adaugarea mai multor servere(creste perform. adaugand mai multe calculatoare)

DBMS (Database Managment System)

- totalitatea programelor utilizate pt crearea, interogarea si intretinerea unei baze dedate
- Operatii CRUD: Create(POST/PUT), Retrive(Get), Update(PUT/PATCH), Delete

Monolit (o singura unitate)

- vin request-urile printr-un frontend layer
- se trec printr-un load-balancing layer
- si ai servere care hosteaza copii complete ale aplicatiei
- fiecare copie contine toate feature-urile si functiile pe care le poate oferi aplicatia, fie ca le folosesti fie ca nu
- o aplicatie de tip monolit descrie o aplicatie software pe un singur nivel in care interfata utilizatorului si codul datelor de acces sunt combinate intr-un singur program pe o sg platforma
- independenta de alte aplicatii

Avantaje:

- Sunt simplu de dezvoltat (toate instrumentele pentru dezvoltare suportă acest tip de aplicații)
- Sunt simplu de lansat (toate componentele fiind împachetate la un loc)
- Este o scalare ușoară a întregii aplicații

Dezavantaje:

- Sunt complexe și ca urmare este dificilă îmbunătățirea în timpul de viață.
- Adaptarea la noile tehnologii este destul de dificilă
- Sunt greu de integrat într-un proces continuu de dezvoltare CI/CD (Continuous Integration / Continuous Delivery)
- Pornirea aplicației durează destul de mult, din cauza faptului că toate componentele trebuie încărcate înainte de lansarea în execuție.
- Comportamentul eronat al unei componente va conduce la oprirea parțială sau totală a execuției

SOLID la microservicii

S:Single

- un microserviciu ar trebui sa implementeze doar o functie de business

- poti sparge microserviciul coarse-grained in microservicii fine si sa le bagi un orchestrator

O:Open-Closed

- un microserviciu nu ar trebui sa fie niciodata modificat ca sa ofere functionalitati ocazionale sau de exceptie

- in loc sa ai un serviciu ce include cazurile exceptionale, faci un serviciu pentru ele si apelezi serviciul initial

L:Liskov

- o versiune noua a unui microserviciu ar trebui intotdeauna sa poata inlocui o versiune veche fara sa strice nimic

I:Interface Segregation

- un microserviciu n-ar trebui sa expuna metode care nu-s direct asociate

- daca folosesti doar 30% din microserviciu, atunci ala nu e microserviciu

D:Dependency Inversion

- un microserviciu n-ar trebui sa apeleze direct alt microserviciu

- ar trebui sa foloseasca un un Service Discovery(gen Docker) sau sa lase aplicatia sa determine la runtime ce microserviciu sa invoce

Cozi de mesaje

- seamana cu fifo dar fara complexitate asociata cu deschiderea si inchiderea acestora
- permit o modalitate de transmitere a unui bloc de date de la un proces la altul
- daca le folosim, nu va trebui sa parcurgem mesajele care ne intereseaza

Semafoare

- unul dintre cele mai cunoscute mecanisme de sincronizare
- introduse de Dijkstra
- o variabila specifica care poate avea numai valori pozitive si asupra careia pot fi permise numai doua operatii: WAIT, SIGNAL

CLUSTER

- resursele unei rețele de calculatoare distribuite sunt combinate pt a deservi un singur utilizator sau task
- au fiecare nod setat sa efectueze aceeasi sarcina controlata si programata de software

INLANTUIRE (CHAINING)

- sunt invocate mai multe apeluri de metode din OOP, fiecare metoda returnand un obiect si permitand ca apelurile sa fie inlantuite intr-o singura declaratie fara a necesita variabile pt a stoca rezultatele intermediare (fiecare dansator se misca sincronizat cu ceilalti dansatori dar nimeni nu le coordoneaza miscarile)

ORCHESTAREA

- coordonarea mai multor servicii printr-un mediator centralizat precum un consumator de servicii
- EX: o orchestra : dirijorul->consumator de servicii

Muzicienii-> sunt coordonati de dirijor

EDGE COMPUTING

- Paradigma de calcul distribuita care aduce calculul si stocarea datelor mai aproape de locatia in care este necesar pt a imbunanati timpul de raspuns si a economisi latimea de banda

SERVERLESS COMPUTING

- model de executie in cloud in care un provider ruleaza serverul si manageriaza dinamic alocarea resurselor masinii. Pretul e bazat pe cate resurse consuma aplicatia, fata de un pret stabilit dinainte.

If your PaaS can efficiently start instances in 20ms that run for half a second, then call it serverless"

- **Avantaje:**

  - incarcari mici
  - decuplare maxima
  - scalabilitate
  - eficiente economic

- **Limitari:**

  - timp
  - memorie
  - tehnologie proprietara
  - solutii publice
  - securitate

PROTOCOALE DE COMUNICATIE

AMQP(Advance Message Queuing Protocol):

- Protocol de nivel de aplicatie standard deschis pt a mijloci mesajele orientate
- CARAC: orientarea mesajelor, asteptarea, rutarea, fiabilitatea si securitatea

XMPP(Extensible Messaging and Presence Protocol)

- Protocol de comunicatie deschis/proiectat pt mesageria instant, informatii prezente si informatii si intretinerea listei de contacte
- Bazat pe XML

STOMP(Streaming Text Oriented Messaging Protocol)

- Cunoscut ca TTMP
- Protocol simplu bazat pe text, proiectat pt lucrul message-oriented middleware

Map Reduce:

- Model de programare si o implementare asociata pt procesarea si generarea seturilor mari de date cu un algoritm paralel si distribuit pe un cluster.
- Model specializat pe strategia ”imparte-aplica-combina” pt datele analizate

CI_CD(ContinuousIntegration_ContinuousDeliveryDeployment)

- Set de practici folosite de dezvoltatori care sa asigure scalabilitatea proiectului, sa poti adauga cu usurinta proiecte noi si sa mentii compatibilitatea

SERVICIU

- Forma de incapsulare folosita pt scalabilitate mare si vine peste stratul de OOP si alte paradigme de programare

AGGREGATOR

- Putem grupa datele de la diferite servicii si sa trimitem un raspuns final consumatorului

BPM(Business Process Modeling)

- Activitatea de reprezentare a proceselor unei intreprinderi a.i.procesul curent sa poata fi analizat

BIG DATA

- Colectie de date ce au un volum mare, totusi crescand exponential cu timpul
- Sunt date cu mari dimensiuni si complexitati, atat de mari incat niciunul dintre instrumentele traditionale de gestionare a datelor nu le poate stoca sau prelucra eficient

ROI(Return On Investment)

- Masura de performanta utilizata pt evaluarea eficienta sau profitabilitatea unei investitii sau pt a compara eficienta unui numar de investitii diferite
- Incearca sa măsoare direct valoarea rentabilitatii unei anumite investitii in raport cu costul investitiei

Software Factory

- Colectie structurala de active software conexe care ajuta la producerea de aplicatii software pt computer sau componente sotfware in conformitate cu cerintele specifice, definite extern ale utilizatorului final printr-un proces de asamblare

SWARM

- Grup de masini fizice sau virtuale care ruleaza aplicatia DOKER si care au fost configurate pt a se uni intr-un cluster
- Instrument de orchestrare a containerelor =>permite utilizatorului sa gestioneze mai multe containere desfasurate pe mai multe masini gazda

Facilitati oferite de D.S.

- Management
- Scalare
- Retea intre gazde
- Descoperirea serviciilor
- Echilibrarea incarcarii
- Nodul din roi

Microserviciile aplicatiilor:

Avantaje:
- pot folosi cele mai noi tehnologii.
- compozabilitatea este ridicată.
- cele independente pot fi scalate separat (nu e nevoie de scalarea întregului sistem)
- Defectarea unei componente nu va duce la căderea sistemului.
- Pentru dezvoltare se utilizează echipe mici care lucrează în paralel la dezvoltarea microserviciilor-> timpul de dezvoltare se micșorează.
-Procesul de integrare/dezvoltare continuă este nativ.

Dezavantaje:
- Mentenanța codului de bază independent este foarte dificilă.
- Monitorizarea întregului sistem este o adevărată provocare, din cauza decentralizării.
- Are un cost (overhead) de performanță adițional din cauza latenței rețelei (network latency)

Principii generale pentru proiectarea cu microservicii:

Modeled around business capabilities - proiectarea software are o componentă de abstractizare, dezvoltatorii fiind obișnuiți să primească sarcini și să le implementeze, dar trebuie luat în considerare cum o să fie înțeleasă soluția, atât acum cât și în viitor.
Cuplare scăzută (Loosely couple) - Nici un microserviciu nu există pe cont propriu,fiecare sistem având nevoie să interacționeze cu altele, dar e nevoie ca această interacțiune să fie slab cuplată. De exemplu, dacă se proiectează un microserviciu care returnează numărul de oferte disponibile pentru un anumit client, este nevoie de o relație către clientul respectiv (customer ID), iar acesta ar fi nivelul maxim de cuplare acceptat.
Responsabilitate unică (single responsibility) - Fiecare microserviciu are ca responsabilitate o singură parte din funcționalitatea aplicației, iar acea responsabilitate este încapsulată în interiorul lui.
Ascunderea implementării (Hiding implementation) - Microserviciile au în general un contract (o interfață) clar și ușor de înțeles, care ascunde detaliile de implementare.Detaliile interne nu ar trebui expuse, nici implementarea tehnică, nici regulile de business care o conduc.
Izolare (Isolation) - Un microserviciu trebuie izolat fizic și/sau logic de infrastructura care utilizează sistemul de care depinde (baza de date, server, etc). Astfel, se poate garanta că nimic extern nu poate afecta funcționalitatea aplicației, iar aplicația nu poate afecta ceva extern.
Instalare independentă (Independently deployable) - Un microserviciu trebuie să poată fi instalat (deployed) în mod independent. În caz contrar, există un nivel de cuplare în interiorul arhitecturii care trebuie rezolvat. Abilitatea de a livra în mod constant este un avantaj al arhitecturii microserviciilor; orice constrângere ar trebui înlăturată, la fel de mult cum dezvoltatorii rezolvă erori în aplicațiile lor.
Creat pentru gestiunea posibilelor erori (Build for failure) - Dacă ceva poate merge prost, va merge prost(Murphy) - Nu contează câte teste sunt realizate, câte alerte pot fi declanșate; dacă microserviciul „pică", dezvoltatorii trebuie să ia în calcul acea posibilă eroare, să o trateze pe cât de elegant posibil și să definească cum se poate corecta(recovery). Când se proiectează un microserviciu, se au în vedere următoarele arii:
Upstream = înțelegerea felului în care dezvoltatorii o să trimită sau nu notificări de eroare clienților, ținând totodată cont de evitarea cuplării
Downstream = cum vor gestiona dezvoltatorii defectarea unui microserviciu sau a unui sistem (precum o bază de date) de care depind
Logging = afișarea tuturor erorilor într-un fișier de log, ținând cont de cât de des se realizează salvarea acestor informații, de cantitatea de date și cum pot fi

acestea accesate. De asemenea, trebuie luate în considerare și cazuri speciale, cum ar fi informații sensibile și implicații de performanță.
Monitoring = Monitorizarea trebuie să fie proiectată cu mare atenție. Este foarte dificil de gestionat o eroare fără informațiile potrivite în sistemele de monitorizare. Dezvoltatorii trebuie să determine ce elemente ale aplicației au informații semnificative.
Alerting = presupune înțelegerea căror semnale pot indica faptul că ceva nu este în regulă, legătura semnalelor cu sistemul de monitorizare și logging-ul.
Recovery = proiectarea modului în care se revine (în urma unor erori) într-o stare normală. Revenirea automată (automatic recovery) este ideală, dar având învedere că aceasta poate eșua, nu trebuie evitată revenirea manuală (manual recovery).
Fallbacks = Un mecanism bun de tratare a erorilor permite ca aplicația să funcționeze în continuare după apariția unei erori în sistem, în timp de dezvoltatorii lucrează să rezolve problema respectivă.
Scalabilitate (Scalability) - Microserviciile trebuie să fie scalabile independent. Dacă este nevoie să se mărească numărul de cereri care poate fi gestionat, sau câte înregistrări poti fi stocate, acestea trebuie făcute în izolare. Se evită scalarea aplicației prin scalarea mai multor componente, impusă de o cuplare mare.
Automatizarea (Automation) - Microserviciile trebuie proiectate ținând cont de lanțul specific CI/CD, de la construire și testare până la instalare și monitorizare. Modelul pentru dezvoltare/integrare continuă CI/CD trebuie proiectat de la începutul arhitecturii.

Domain-Driven Design (DDD)

Proiectarea bazată pe analiza domeniului reprezintă o manieră pentru dezvoltarea aplicațiilor complexe prin conectarea continuă a implementării la un model (care evoluează continuu) a conceptelor business de bază.

Premisele DDD:
•accentul principal al proiectului cade pe domeniul de bază (core domain) și pe logica domeniului (domain logic)
•Proiectele mai complexe trebuie bazate pe un model
•inițierea unei colaborări creative între experții tehnici și experții din domeniu pentru a ajunge din ce în ce mai aproape de modelul conceptual al problemei
Problema: când complexitatea scapă de sub control, software-ul nu mai poate fi înțeles suficient de bine pentru a putea fi schimbat sau extins cu ușurință. Dacă complexitatea domeniului nu este tratată în proiectare, nu contează că tehnologia infrastructurii suport este bine concepută.

Principiile de proiectare:
•**Context mărginit (Bounded context):** Când se abordează un sistem complex, deobicei se abstractizează într-un model care descrie aspectele diferite ale sistemului și cum poate fi folosit pentru a rezolva probleme. Când există mai multe modele, iar codul de bază al diferitelor modele este combinat, software-ul devine plin de erori (buggy), nesigur și greu de înțeles. În DDD, se definește contextul în care se aplică un model, se stabilesc explicit granițele în ceea ce

privește organizarea echipei și utilizarea în anumite părți ale aplicației,păstrând modelul **consecvent** cu aceste limite.

•**Limbaj generic specific (Ubiquitous language):** În DDD trebuie alcătuit un limbajcomun și riguros între dezvoltatori și utilizatori. Acest limbaj trebuie să fie bazat pe modelulde domeniu, ajutând în a avea o conversație generală între toți experții din domeniu, acestlucru fiind esențial la abordarea testării.

•**Capturarea/maparea contextului (Context mapping):** Într-o aplicație de dimensiunimari, proiectată pentru mai multe contexte mărginite (bounded contexts), se poate pierdevederea de ansamblu. Inevitabil, contextele mărginite vor fi nevoite să comunice date întreele. O mapare de context este o vedere de ansamblu (global view) asupra sistemului ca unîntreg, care ilustrează maniera în care contextele mărginite ar trebui să comunice între ele.

Folosirea DDD în microservicii:

•**Bounded Context** - Nu trebuie creat un microserviciu care include mai mult de uncontext mărginit

•**Ubiquitous Language** - Dezvoltatorii trebuie să se asigure că maniera de comunicareutilizată este suficient de general valabil, astfel încât operațiile și interfețele care sunt expusesă fie exprimate utilizând limbajul domeniului context

•**Context Model** - Modelul utilizat de microserviciu trebuie definit într-un contextmărginit și să folosească un limbaj generic (ubiquitous language), chiar și pentru entități carenu sunt expuse în nici o interfață pe care o oferă microserviciul

•**Context Mapping** - Trebuie examinat contextul mărginit al întregului sistem pentru a înțelege dependențele și cuplarea microserviciilor.

Flux de date reactiv: - o colecţie de date emisă încontinuu, pe măsură ce datele sunt pregătite

Principiile programării reactive

•**receptivitatea** (*responsiveness*) - aplicaţiile moderne ar trebui să răspundă cererilor întimp util, dar nu numai utilizatorilor care le utilizează, ci şi rezolvarea problemelor şirecuperarea după apariţia erorilor trebuie să se conformeze constrângerilor de timp;

•**rigiditatea** (*resilience*) - realizabilă prin replicare, care la rândul ei depinde descalabilitatea sistemului

•**elasticitatea** (*elasticity*) - sistemele reactive trebuie să fie elastice, astfel încât să se poatăadapta sub diverse grade de încărcare (exemplu: număr mare de cereri), scalând resurseledisponibile în funcţie de nevoie

•**orientare spre mesaje** (*message-driven*) - sistemele reactive folosesc mesaje asincrone pentru a transmite informaţia prin diverse componente, având cuplare foarte slabă cepermite interconectarea acestor sisteme în izolare

•**supra-saturarea fluxurilor** (*back-pressure*) - se produce atunci când un sistem reactivpublică mesaje într-un ritm mai alert decât pot fi gestionate de entităţile înscrise pentru aprimi mesajele.

Componentele unui pipeline:

-Sursa: reprezintă generatorul de evenimente, sursa de date din pipeline,care produce date ce urmează a fi procesate

-Procesor: entitate care preia evenimente de la sursă şi le procesează sub o anumită formă

-Sink: reprezintă destinaţia evenimentelor procesate; această entitate interceptează mesajele de la Procesor.

Service Consumer (Consumator) :

- cand un program invoca si interactioneaza cu un serviciu se numeste serviceconsumer

- termenul se refera la rolul temporar luat la runtime de un program atunci cand eangajat cu un serviciu pentru schimb de date

Service Provider:

- serviciul invocat de consumer

Principiile service oriented (SOA):

1) Standardized Service Contract

Serviciile din acelasi inventory sunt in conformitate cu aceleasi standarde de design2)Service Loose Coupling

Contractele serviciilor impun cerinte de cuplare scazute pentru consumator si sunt larandul lor decuplate de mediul inconjurator

3) Service Abstraction

Contractele serviciilor contin doar informatii esentiale si informatiile despre serviciisunt limitate doar la ce e publicat in contract

4) Service Reusability

Serviciile contin si exprima logica agnostica(nu conteaza contextul in care suntchemate) si pot fi considerate resurse enterprise reutilizabile

5) Service Autonomy

Serviciile exercita un nivel ridicat de control asupra mediului de executie underlyingla runtime

6) Service Statelessness

Serviciile minimizeaza consumul de resurse prin amanarea managementului informatiilor despre stare cand e necesar

7) Service Discoverability:

Serviciile sunt suplimentate cu date meta de comunicatie prin care pot fi descoperitesi interpretate eficient(cum e docker, ai un registru si le iei de-acolo)

8) Service Composability

Serviciile sunt participanti efectivi la compozitie, indiferent de marimea si complexitatea compozitiei

Caracteristici de baza SOA:

a) Business-driven:

- TA e aliniata cu arhitectura curenta de business. Contextul e apoi mentinut constantasa incat TA sa evolueze in tandem cu businessul de-a lungul timpului.

- TA se tot indeparteaza de Business pana in punctul in care trebuie gandita de la 0 side aia trebuie sa le tii aproape

b) Furnizor neutru:

- Modelul arhitectural nu e bazat strict pe o platforma furnizor, poti combina mai multe tehnologii, sa le inlocuiesti pe unele ca sa maximizezi realizarea necesitatilor afaceriiin mod constant

- ai mai multa libertate la implementare

c)Enterprise-centric:

- Scopul arhitecturii reprezinta un segment important al intreprinderii, permitand reutilizarea si compozitia serviciilor, permitand solutii orientate serviciu in favoarea celor tradiționale

- nu iti imparti serviciile pentru anumite lucruri, le pui la dispozitie pentru toti din firma

d) Composition-centric:

Arhitectura suporta mecanici de agregare repetata a serviciilor, permitand acomodarea schimbarilor constante prin asamblare agila a compozitiilor de servicii

Agregator:

- spune cum putem grupa datele de la diferite servicii si sa trimitem un raspuns finalconsumerului

- se poate face in 2 moduri: un microserviciu composite care face apeluri lamicroserviciile necesare, consolideaza

datele si le transforma inainte de a le trimite inapoi(adica orchestrarea de laSOA, eventual un adapter)

- API Gateway poate face request-uri la mai multe microservicii si sa uneasca dateleinainte de a le trimite la consumer

- daca ai logica de business mergi pe prima varianta

ETCD:

-este un depozit distribuit de tip cheie-valoare, care asigură o consistenţă sporită şi furnizează o modalitate sigură de a stoca date ce trebuie accesate de un sistem distribuit sau de un cluster de maşini de calcul.
-gestionează automat alegerile de lider din timpul partiţionărilor reţelei şi poate tolera defecte ale maşinilor de calcul, chiar şi la nivelul nodului lider.

Arhitectura JEE

Platforma JEE (Java Enterprise Edition) este proiectatăpentru a-i ajuta pe dezvoltatori să creeze aplicaţii specifice întreprinderilor sau corporațiilor,multinivel - multistrat, scalabile, fiabile şi sigure.