Mentiuni:

1. ROSU = ce mi s-a parut mai putin important
2. VERDE = ce mi s-a parut mai important si mai interesant (nu sunt multe chestii totusi:)))
3. Materia pentru examen e pana la Curs 10 slide 18 (daca se si tine de cuvant…)
4. Cursurile 1, 3 si 4 sunt expuse cat de “pe larg” posibil, am facut cursul online si am putut *nota* mai multe (fara cursul 2, acolo am fost dat afara pentru ca nu am dat camera:)))
5. De la cursul 5 in colo am facut o oarecare intersectie din subiectele mai importante abordate la curs (cel putin ce s-a discutat si este si pe slide-uri) si ce se mai gaseste prin alte materiale
6. In multe materiale se gasesc foarte multe chestii pe langa ce s-a vorbit la curs. Cum profu nu cred ca isi aminteste foarte bine ce s-a predat e posibil sa dea si din alea (cum e posibil sa dea din absolut orice ii vine in gand)...

Curs 1

Sistemele de calcul hardware se clasifica in:

- Sisteme strans cuplate (distanta intre procesoare e de maxim cativa metri)
- Sisteme slab cuplate
  - Mai multe noduri (calculatoare, senzori, microcontrollere etc)=orice cu procesor, memorie si modalitate de comunicare
  - Formeaza “Internet of Things”

Principala problema a sistemelor distribuite este ascunderea heterogenitatii nodurilor. O solutie = creare strat de virtualizare (cu Virtual Machine sau Cloud sau cu Interpretare).

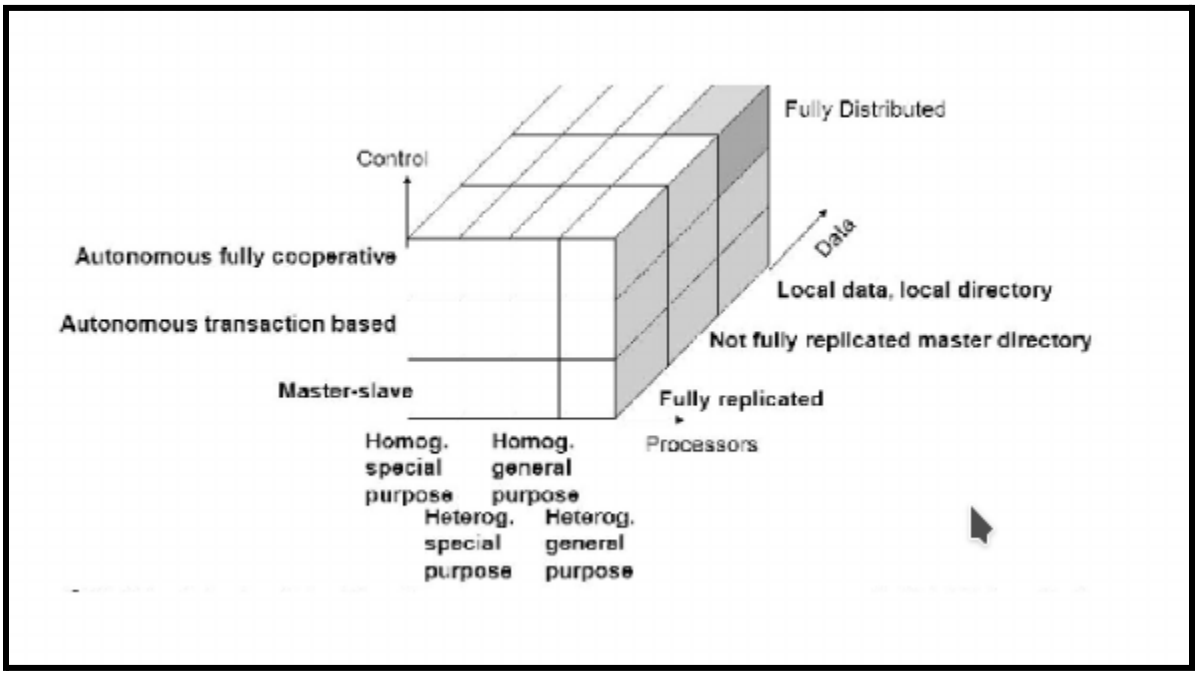
Arhitecturi software strans cuplate = aplicatii clasice care colaboreaza intre ele prin anumite scheme de cooperare (ierarhic, client-server) si capata atributul de distribuit.

Slab cuplate = pot coopera la nivel aplicatie dar arhitectura lor interna e tot slab cuplata si formata din componente care interopereaza in vederea realizarii unui scop comun. Fiecare portiune a arhitecturii se poate scala independent =>

fiecare aplicatie poate fi alcatuita din mai multe componente ce opereaza independent.

Enslow: sistemele distribuite se pot clasifica dupa 3 dimensiuni ale distribuirii:

- Control
- Procesare
- Date



Distribuirea fizica: exista minim 2 noduri, fiecare putand comunica printr-un mediu de comunicatie. Nu are legatura cum sunt distribuite dpdv fizic cu rolurile lor (ex: pe un nod pot rula anumite microservicii, pe alt nod mai multe microservicii, pe alt nod un singur microserviciu dar mai solicitant computational)

Distribuirea datelor:

Prin

- replicare (copii multiple in locatii diferite)
- partitionare (bucati din date la diferite locatii).

Se face pentru:

- A se gestiona mai bine defectele
- A creste performantele

S-au coborat anumite functionalitati la nivelul sistemelor de operare pentru dezvoltarea mai usoara a sistemelor distribuite.

1. SO ofera suport pentru servicii de retea.

2. Se ascunde heterogenitatea nodurilor prin introducerea unui nou strat de servicii “middleware”.
3. SO ofera servicii distribuite.

Altfel privit: aplicatiile trebuie sa comunice printr-un mediu de comunicare care sa ascunda cat mai bine detaliile. Astfel a aparut MOM (Message Oriented Middleware), permite cereri asincrone.

Avantaje SD

- Modularitate: proiectare simpla, ascundere heterogenitate
- Flexibilitate: prin folosirea de diferite echipamente se obtine acelasi comportament
- Scalabilitate: se pot adauga noi noduri in sistem fara a pierde prea multa performanta
- Fiabilitate si integritate: daca avem distributie fizica, un nod lucreaza indiferent de starea celorlalte noduri (daca 1 nod se strica sistemul continua sa functioneze daca software-ul e proiectat sa redirectioneze sarcinile nodului stricat catre alte noduri)
- Performanta:
    - timp de raspuns mai mic daca procesarea se face local (dar trebuie tinut cont si de complexitatea sarcinii si puterea de calcul a nodului) deoarece se evita comunicarea prin retea
    - comunicarea prin retea ar putea fi mai costisitoare decat realizarea task-ului
    - uneori e mai rapid de facut preprocesare si arhivare local si apoi folosita comunicatia prin retea

Dezavantaje SD

- Necunoasterea starii globale:
    - Starea globala e discreta
    - Nodurile au viteze de comunicatie si rate de defectare diferite
    - E nevoie de cunoasterea starii globale pentru echilibrarea incarcarii, redirectare task-uri (la defectari/supraincarcari)
    - Informatiile despre starea nodurilor pot veni prea tarziu/deloc
- Lipsa unui timp global:
    - Nu se stie cine a facut ultima modificare (pentru ca ceilalti sa poata prelua de la el informatia)
    - Ar fi nevoie de un timp comun

        - Daca nu exista o sincronizare chiar si un algoritm bun poate avea probleme
        - Problema s-a mai diminuat deoarece exista servicii care asigura sincronizarea
    - Nedeterminismul:
        - Nu se poate prezice sigur rezultatul unui task cu anumite date de intrare
        - Exista nedeterminism cauzat de diferentele vitezelor de procesare a nodurilor
    - Comunicatii:
        - Ar trebui sa existe un schimb singur de info (in timp util, date nealterate)
        - Datele trebuie rutate pentru a se echilibra incarcarea si neparcurgerea unui drum prea lung
        - Exista posibilitatea de a fi atacat un sistem distribuit exploatandu-se comunicatiile
    - Securitatea
        - Multe sisteme distribuite au la baza HTTP, REST etc si, deci, nu au securitate serioasa
        - Daca nodurile comunica intre ele prin alte mijloace decat cele bazate pe REST, atunci se poate vorbi despre o oarecare securitate
        - Peste TCP/IP se pot folosi design pattern-uri specifice securitatii si criptari suplimentare pentru a spori securitatea

Cloud-urile

Prezinta arhitecturi

- Multi-strat
    - Nr. nedeterminat
    - Functionalitati diferite
    - Separate intre ele
- Multi-nivel
    - Deobicei 3: prezentare, business logic si persistenta

Pana acum am folosit arhitecturi de tip monolit, cel mai ok fiind arhitectura de tip modular (cu dezavantaje ca: cuplare mare intre clase, ciclu de viata redus, cost mare de intretinere etc).

In aplicatiile moderne apare “controlul invers”.

Controlul invers

- De cele mai multe ori, in aplicatii se utilizeaza anumite framework-uri. Acestea au o multime de functionalitati iar alegerea functionalitatilor dorite este de multe ori complicata. Uneori se pot creea framework-uri de tip *wrapper* (care impacheteaza) pentru alte framework-uri mai complicate, facilitand rezolvarea unor task-uri comune, simplificand astfel utilizarea framework-urilor complicate.

- Astfel se rezolva task-urile folosind acea multime de functionalitati dar in moduri generice, cat mai transparente pentru utilizator

- Controlul (puterea) framework-urilor se poate spune ca e mai mare decat cel al clientilor (deoarece acestia apeleaza doar niste metode, greul e realizat de framework-uri).

Injectarea dependentelor

- E o abordare a PAOO.
- Se amesteca cu controlul invers de multe ori.
- A condus la paradigma programarii orientate aspect
- Presupune injectarea unor obiecte (numite dependente) ce asigura anumite functionalitati
- De exemplu: exista un client de mail. Se vrea ca acel client sa aiba inca o functionalitate f1. Se foloseste mecanismul de injectare a dependentelor: i se creeaza clientului o dependenta de un obiect care sa aiba functionalitatea f1. Acel obiect poate fi creat de la 0, poate fi furnizat de un framework, poate fi furnizat de un microserviciu. Clientul va depinde de acel obiect, se va folosi de functionalitatea f1 a obiectului si se va comporta de parca el insusi ar furniza-o.
- Vechi: injectie manuala: fabrica de obiecte, proxy, apel de constructor
- Nou: injectie automata:
    - Se separa modul de injectie de codul de baza
    - Codul de baza=obiecte primare, dependente

    - In exterior=intr-un limbaj dedicat/simplist (ex: XML) se descrie maniera de injecte a dependentelor
- Exista framework-uri (ex: Spring) care injecteaza dependentele in codul de baza folosind fisierele de descriere (din exterior)
- Adnotarea automata functioneaza doar pentru aplicatii simple (inferenta/deductia facuta de framework e limitata, nu functioneaza in cazuri prea complicate)

Dupa OOP a aparut programarea orientata pe componente (unitati independente de implementare), numite CBSE (Component Based Software Engineering).

Comportamentul unei componente = un set de interfete. Implementarea specificatiilor componentei poate depinde si de alte componente. Componenta instalata = copie a implementarii componentei. Instalarea componentei are loc in timpul executiei. O componenta poate avea mai multe componente.

Java Beans = model de dezvoltare a componentelor software care permite componentelor din Java (numite Beans) sa comunice intre ele. Un Bean = clasa Java care respecta anumite conventii de constructie:

- Are constructor implicit (fara argumente)
- Trebuie sa fie serializabila (implementeaza interfata Serializable)
- Poate avea un numar de proprietati (+gettere si settere pentru ele)

Arhitectura CORB

E de tip magistrala, deci are probleme cu scalabilitatea (mai mult centralizata decat distribuita).

Exista gestionari de obiecte (Brokeri), ei cauta obiecte in mediul distribuit si asigura comunicarea intre ele.

Java RMI

Clientul cere Naming Service-ului (serverului) o referinta la un obiect

Serverul creeaza (daca nu este deja un obiect disponibil) si il “leaga” cu clientul (*bind* daca il creeaza, *rebind* daca e deja creat)

Clientul se foloseste de Proxy Servant Object pentru a utiliza functionalitatile Servant Object-ului in stilul Remote Procedure Call

Dezavantaj: clientul trebuie sa fie in Java

Serviciu = actiune care satisface o necesitate. Este diferit de un bun. Se pune accent pe competentele celui ce satisface necesitatea.

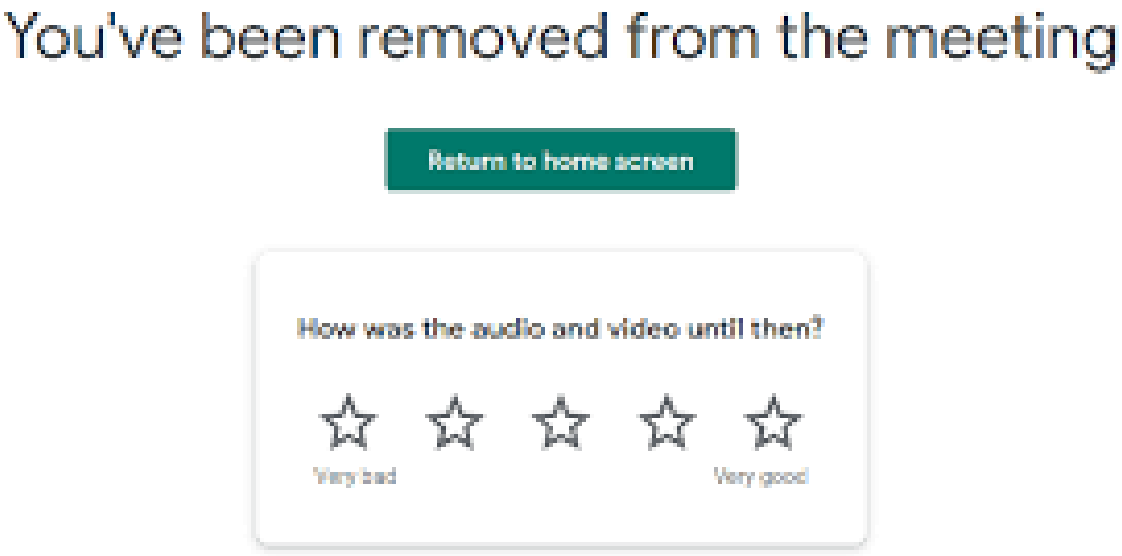
Serviciu WEB = colectie de API-uri care colaboreaza in vederea realizarii unei sarcini. Pot fi accesate printr-un protocol de comunicare (nu neaparat HTTP).

Standarde

2 standarde: SOAP si REST (nu sunt sigure nici una, SOAP e mai sigur totusi)

SOAP este un protocol sau un set de standarde	REST este o architectură software.
SOAP acronimul lui **Simple Object Access Protocol**	REST acronimul lui **Representational State Transfer**
SOAP nu poate utiliza rest REST deoarece este un protocol.	REST poate utiliza SOAP deoarece fiind model de proiectare arhitectural poate utiliza orice protocol ar dori
SOAP utilizeaza interfața serviciului pentru a expune logica de afaceri.	REST utilizeaza URI pentru a expune logica de afaceri. Deoarece el foloseste cereri HTTP același URI poate fi utilizat petru diverse tipuri de operații
SOAP își definește propriul standard de securitate.	REST preia/are securitatea protocolului de transport utilizat
SOAP lucrează numai cu formatul XML.	REST aceptă diverse formate de date, fișier text, HTML, JSON, XML etc.

Curs 2



Stateless session bean

- Nu persista (stocheaza) date
- Are 2 stari:
  - Does not exist
  - Ready
- Ciclu de viata:
  1. Clientul initiaza ciclul de viata prin obtinerea unei referinte catre Bean
  2. Containerul injecteaza dependintele acesteia
  3. Containerul apeleaza metodele adnotate cu @PostConstruct (daca exista)
  4. Bean-ul devine Ready si poate fi folosit
  5. La sfarsit se apeleaza metodele cu @PreDestroy
  6. Garbage collector colecteaza bean-ul
- La o a 2a utilizare a bean-ului se creeaza o a 2a instanta
- Nu supravietuiesc caderilor sistemului

Stateful session bean

- Are 3 stari:
  1. Does not exist

2. Ready
3. Passive

- Ciclu de viata:
    1. Clientul initiaza ciclul de viata prin obtinerea unei referinte catre Bean
    2. Containerul injecteaza dependintele acesteia
    3. Containerul apeleaza metodele adnotate cu @PostConstruct (daca exista)
    4. Bean-ul devine Ready si poate fi folosit
    5. E posibil ca, atat timp cat e Ready, bean-ul sa fie trecut in modul Passive de catre container
    6. La sfarsit clientul apeleaza Remove
    7. Se apeleaza metode adnotate cu @PreDestroy
    8. Garbage collector colecteaza bean-ul

Spring

- Este un framework ce a fost construit pentru a face mai usoara dezvoltarea aplicatiilor pentru intreprindere.
- S-a urmarit flexibilitatea si interoperabilitatea.
- Interopereaza cu GlassFish
- Persistenta nativa: MySQL
- Poate face render la JSP-uri
- Prin Maven si Gradle ajuta dezvoltatorul sa se concentreze la programarea primara
- DAO (Data Access Objects) sunt utilizate pentru a incapsula orice accesare catre sistemul de persistenta, daca se foloseste Hibernate, codul acestuia se afla in DAO.
- MVC:
    - Controller: un servlet care decide ce se afiseaza
    - View: o pagina JSP
- @Autowired - adnotare pentru variabile in care vor fi injectate Bean-uri
- @Component, @Repository, @Service, … - Bean-uri
- Tipuri principale de containere:
    - Fabrici de bean-uri
    - Contextele pentru aplicatii

CoreSpring

- Se ocupa cu gestiunea bean-urilor (creare, configurare)
- E componenta de baza a Spring
- Contine fabrica de bean-uri ce realizeaza injectarea dependentelor

AOP (Aspect Oriented Programming)

- Paradigma de programare
- Are ca scop cresterea modularitatii
- Permite separarea preocuparilor transversale
- Completeaza OOP-ul oferind alt mod de a gandi structura unui program
- La OOP unitatea fundamentala e clasa, la AOP este aspectul
- Are legatura cu paradigma programarii declarative

Necesitati/concepte transversale (crosscutting concerns)

- Concepte care se aplica in si afecteaza intreaga aplicatie
- Ex: jurnalizarea, securitatea

Stiva de programe

- Concretizarea cu tehnologii a modelului arhitectural multi-nivel multi-strat aparut si definit in fazele de analiza si proiectare

Automatizarea dezvoltarii aplicatiilor client se face prin gestionare de proiecte precum Maven si Gradle

JEE este nesigur deoarece are o arhitectura de monolit distribuit, totul fiind amestecat.

Servleti = programe Java care proceseaza cereri (de ex. cele care raspund la cereri HTTP, facand render la un template JSP)

Paginile JSP = documente HTML “template” ce contin cod Java (snippets/scriptlets). Servletii le randeaza, executand codul Java din ele si transformandu-le in pagini HTML

Bean de sesiune = obiect utilizat in cadrul unei conversatii temporare cu un client

Bean activat de mesaje

- Implica:
  - Mesaje de sesiune
  - Un sistem numit Java Message Service ce le gestioneaza (ca pe intreruperi)

Enterprise information system

- Orice sistem informational care imbunatateste productivitatea intreprinderii

JNDI = serviciu care asigura descoperirea altor servicii

EAR

- **E**nterprise **AR**chive
- Un JAR specific aplicatiilor pentru intreprinderi

Bean-uri

- De sesiune (cu/fara stare)
- De entitate (incapsuleaza date)
- De mesaje (intermediaza comunicarea intre bean-uri si alte componente)

Curs 3

Arhitectura tehnologiei = elementele fizice, constitutive si fundamentale utilizate in proiectarea unei tehnologii.

Tipuri de arhitecturi:

- Bazata pe componente
  - O arhitectura a unei aplicatii poate contine mai multe arhitecturi de componente
- De integrare
  - Pentru 2+ aplicatii/sisteme
  - Include si alte tehnologii, resurse, extensii necesare
  - Contin middleware si adaptoare
  - Pot contine arhitecturi precursoare acestui tip
- SOA (Service oriented architecture) - arhitectura moderna
  - Arhitectura poate referi componente software/hardware

- Defineste in mod principal maniera de interactionare a subsistemelor software
- Proiectarea arhitecturii pentru intreprinderi duce la modelul de interoperare intre sisteme distribuite si aplicatii distribuite
- Se iau in calcul atat componentele software cat si cele hardware (deoarece acestea “executa” pe cele software)

Cyberspatiul este constituit din internet si dintr-o multitudine de noduri interconectate. Acestea ofera servicii si microservicii si interopereaza pentru a executa o aplicatie.

Infrastructura tehnologica = mediu fizic in care sunt instalate programe si aplicatii software. Contine rutere, firewall-uri, cabluri etc.

Infrastructura software contine sisteme de operare, API-uri, medii de executie, agenti.

Deci arhitectura intreprindere = arhitectura software + arhitectura hardware.

O solutie IT trebuie sa fie proiectata asa incat sa suporte ciclul de crestere.

Definitii

Specificul proiectului = proprietate care rezulta din maniera de proiectare.

Caracteristica = orice abilitate concreta (ex: granularitatea programului)

Proiectarea fizica a unei aplicatii = partial definita de arhitectura si de infrastructura inconjuratoare

Specificarea functionala = stabilesc noi caracteristici

Principiu de proiectare = practica industriala de proiectare. La SOA: set de principii de proiectare care, aplicate simultan, obtin celulele necesare calculului orientat pe servicii.

Model de proiectare (design pattern) = solutie pentru o problema comuna

Standard de proiectare = conventii de proiectare adaptate de fiecare organizatie pentru obtinerea de solutii potrivite domeniului in care activeaza (standard intern). A nu se confunda cu standardele industriale (XML etc).

Calcul orientat pe servicii

Se refera la calcul distribuit modern. Contine noi principii, tehnologii, instrumente. Contine unitati logice (servicii) proiectate individual si cu grad mare de izolare intre ele. Ele se pot utiliza impreuna pentru realizarea de task-uri.

SOA:

- Imbunatateste flexibilitatea
- Reduce costurile de dezvoltare si intretinere

Serviciu = unitate care implementeaza logica unei solutii. E vazut prin intermediul interfetei lui (numita si **contract**). Serviciile in general sunt apelate la distanta prin intermediul interfetei lor. Nu toate serviciile sunt servicii WEB (sunt folosite si alte protocoale, RPC-uri, socket-uri etc).

Serviciu = container de functionalitati pe care le poate oferi intr-un anumit context.

SOA:

- Abordare in care aplicatiile utilizeaza serviciile oferite de retea
- Acestea sunt la distanta, undeva in internet
- De multe ori se foloseste o abordare bottom-up: se combina functionalitati deja existente in vederea alcatuirii unei aplicatii

Avantaje SOA:

- Reutilizare mare
- Usor de mentinut
  - Decuplare mare => modificari/inlocuiri simple de servicii
- Independente de platforma
- Disponibilitate
  - Daca un nod ce oferea un serviciu se defecteaza se poate folosi un alt nod ce ofera acelasi serviciu (chiar daca cel ce s-a defectat era alegerea optima)
- Fiabilitate
  - Mai usor de depanat servicii decat bucati mari de cod
- Scalabilitate
  - Se pot adauga usor noduri, iar pe noduri diferite servicii

Dezavantaje SOA:

- Incarcare a sistemului
- Costuri initiale mari
  - Necesita investitii mari la trecerea de la abordari clasice
- Gestiunea complexa a serviciilor

Tipuri de servicii:

- Serviciu furnizor
  - Organizatia care mentine serviciul si il face disponibil (public/privat)
- Serviciu consumator
  - Cauta servicii in catalogul de servicii
  - Dezvolta componente clienti in jurul serviciilor disponibile

Un serviciu poate fi din ambele tipuri in acelasi timp.

Orchestrarea serviciilor (ce am inteles eu din curs + vreo 3 site-uri): Exista mai multe straturi:

- Resurse
  - Contine unitati fundamentale ce alcatuiesc servicii (masini virtuale, medii de stocare etc).
- Orchestrare de domeniu
  - Utilizeaza unitatile din stratul de resurse
  - Le combina in vederea de alcatuire de functionalitati
  - Le ascunde detaliile
  - Furnizeaza o interfata pentru stratul de orchestrare a serviciilor
- Orchestrare de servicii
  - Se refera la managementul unificat al stratului de orchestrare de domeniu
  - Cizeleaza procesele capat-la-capat
  - Ascunde detaliile oferite de nivelul de orchestrare de domeniu
  - Ofera o interfata si mai simplificata sistemelor individuale (creste abstractizarea)
  - Ea mediaza procesele capat-la-capat dintre sisteme individuale si domenii
  - Din cate am citit ar exista o entitate centrala ce orchestreaza serviciile

Orchestrarea (explicatie simplificata dupa parcurgerea a 14 laboratoare):

- Exista un serviciu dirijor
- Acesta primeste cereri si le proceseaza utilizand celelalte servicii
- Ex:
  - Dirijorul primeste o cerere
  - Acesta utilizeaza serviciul S1, apeland o metoda din contractul sau cu o parte din informatia din cerere

    - S1 returneaza rezultatul dorit
    - Cu rezultatul acela dirijorul apeleaza metode din contractele serviciilor S2 si S3
    - Cu rezultatele celor 2 metode apelate se creeaza un raspuns final care este trimis clientului

Inventarul serviciilor = colectie a serviciilor care poate reprezenta intreprinderea/o parte din ea

Analiza orientata pe servicii = operatie primara in cadrul dezvoltarii unei SOA. Economistii explica cum merge afacerea (regulile de business) arhitectilor tehnologiei.

Serviciu candidat = serviciu care apare in urma discutiei cu clientii. Acesta deseori nu este in forma sa finala, urmeaza a fi spart in mai multe servicii si rafinat.

Un serviciu are mai multe componente si furnizeaza o **interfata (contract)**. Componentele se bazeaza pe un framework suport (deobicei Java si .NET).

Serviciile REST sunt programe simple, cu incarcare usoara, proiectate pentru simplitate, scalabilitate, reutilizare. Favorizeaza abordari SOA.

Principii de proiectare SOA

- Implementarea de contracte standardizate
    - Serviciile ce indeplinesc functii similare trebuie sa aiba interfete identice
- Cuplare scazuta intre servicii
    - Tinand cont ca incapsularea e mai mare ca cea in cazul COP, AOP si OOP si ca serviciile depind de contractele altor servicii (nu de implementarile propriu-zise ale acestora) cuplarea este considerabil mai scazuta
- Abstractizarea serviciului
    - Se obtine implicit deoarece serviciile furnizeaza doar contracte, nu si detalii de implementare
- Reutilizarea serviciilor
    - Au logica incapsulata, nu depind de nimic extern si pot fi reutilizate
- Autonomia serviciilor
    - Au nivel mare de control al mediului in care se executa

- Service statelessness
    - Starea serviciului este gestionata in afara lui pentru a evita incarcarea si pentru a respecta SOLID

Scopuri SOA

- Interoperabilitate implicita crescuta
    - Serviciile sunt gandite ca sa poata fi reasamblate si reconfigurate in diverse arhitecturi
- Nivel mare de generalitate
    - Ofera doar contracte deci pot fi dezvoltate, modificate si inlocuite fara a afecta sistemul din care fac parte
- Cresterea nr. de posibilitati ale unui vanzator
    - Un vanzator are de ales intre mai multe servicii, putand decide care dintre ele merita (nevazandu-le ca pe *cutii negre*)
- Cresterea afacerii si alinierea la domeniul tehnologic
    - Serviciile se pot dezvolta o data cu afacerea
- Eficienta economica
    - Serviciile = bunuri economice de tip IT
    - Se doreste ca ceea ce produc sa depaseasca costul de implementare si intretinere
- Agilitate organizationala sporita
    - Se adapteaza mai usor serviciile la noi cerinte
    - Mediul de lucru devine mai eficient si se reduce utilizarea aplicatiei
- Reducerea incarcarii datorate IT-ului
    - Diminuare complexitate => se reduc costuri => se poate imbunatati performanta

Caracteristici SOA

- Condusa de necesitatile afacerii
- Nu este dependenta de o tehnologie proprietara
    - Se pot combina solutii proprietare cu solutii gratuite / cumparate
- Centrat pe intreprindere
    - Se potriveste cat mai bine cu necesitatile intreprinderii
    - Gestioneaza cat mai bine aplicatiile autonome ale intreprinderii

- Axat pe compunere
    - Serviciile se pot compune, descompune, recompune

Tipuri SOA

Serviciile sunt puse intr-un inventar. Apoi, din inventar se aleg cateva si se compun cele alese in vederea satisfacerii unei nevoi.

4 tipuri:

- Arhitectura serviciului (individual)
    - Comparabila cu COP
    - Nu mai exista un framework urias ca la COP
    - Apare problema complexitatii infrastructurii (chiar daca aceasta ar fi mai simpla ca a framework-urilor utilizate la COP va trebui sa fie facuta de catre intreprindere)
- Arhitectura compunerii serviciilor (descrie cum sunt ele compuse)
    - Se detaliaza uneori fluxul mesajelor dintre servicii (atat directia mesajelor cat si momentele in care apar)
    - Serviciul responsabil de compozitie = controller de compunere
    - Serviciile compuse = membri ai compozitiei
- Arhitectura bazata pe existenta unui catalog de servicii inrudite si independente
- Arhitectura intreprinderii bazate pe servicii

Agentii serviciului

Unele dependente intre servicii pot avea agenti. Agentii proceseaza mesajele de la si catre serviciu. Pot fi furnizati de mediul in care se executa serviciile sau pot sa fie dezvoltati special.

POJO vs Java Bean

POJO	Java Bean
Nu are alte restricții decât cele impuse de limbajul Java	Este un caz particular de POJO care are unele restricții.
Nu permite un control foarte strict al membrilor	Permite controlul total asupra membrilor
Nu poate implementa interfața Serializable	Trebuie să implementeze interfața Serializable
Câmpurile pot fi accesate direct prin numele lor	Câmpurile pot fi accesate numai prin *getteri* și *setteri*.
Câmpurile pot avea orice nivel de vizibilitate	Câmpurile pot fi numai *private*
Este permisă dar **nu obligatorie** utilizarea unui constructor fără argumente (no-arg).	Este **obligatorie** utilizarea unui constructor fără argumente (no-arg)
Este recomandat a fi utilizat când nu se dorește nici un fel de restricții asupra membrilor iar utilizatorul poate avea acces complet la entitatea creată	Este utilizat atunci cand se dorește furnizarea unui acces restricționat a utilizatorului la unele părți din entitatea creată (interfață contract etc)

EJB vs Java Bean

EJB	JAVABEANS
A Java API that allows modular construction of enterprise software	Classes that encapsulates many objects into a single object
EJB requires an application server or EJB container to run EJB applications	JavaBeans should be serializable, have a zero argument constructor and allow access to properties using getter and setter methods
EJB is complex than JavaBeans	JavaBeans is simpler than EJB
Programmer can concern about the business logic as the application server manages services such as transactions and exception handling	JavaBeans allow another application to use properties and methods of the Bean

Curs 4

Principii de utilizare a serviciilor

1. **Principiul vizibilitatii serviciului**
   - Serviciile au metadate prin care sunt descoperite si intelese.
   - Metadate = un fel de fise care ofera informatii de importanta critica.
2. **Contractul standard al unui serviciu**
   - Serviciile din acelasi inventar de servicii au contracte proiectate dupa acelasi standard
3. **Tipuri standardizate de continut ale contractelor**
   - Contractele urmeaza modele prestabilite de continuturi
   - Reprezinta intelegeri intre furnizorul de servicii si utilizator
4. **Principiul contextului de executie al unui serviciu**
   - In functie de contextul in care se executa, un serviciu se poate comporta intr-un mod diferit
   - Serviciile interopereaza direct cu elemente din organizatie => Incapsulare mai mica decat la MSA => Serviciile sunt afectate intr-un mod mai direct de schimbarile de mediu inconjurator
5. **Separarea de functionalitati la nivelul serviciilor**
   - Serviciile detin si controleaza numai functionalitatile lor
6. **Abstractizarea serviciilor**
   - Contractele contin doar informatii esentiale
   - Tot ce se poate sti despre servicii se gaseste in contracte
7. **Abstractizarea contractelor**
   - Contractele contin doar functionalitatile publice ale serviciilor
   - Ele abstractizeaza serviciile deoarece implementarile concrete ale serviciilor pot contine si alte functionalitati (deobicei ca metode private), ele netrebuind sa fie cunoscute de catre consumatori
8. **Compunerea serviciilor**
   - Serviciile pot fi compuse din alte servicii
   - Serviciile pot compune alte servicii in timp ce sunt compuse din servicii
9. **Autonomia serviciilor**
   - Au nivel mare de control asupra mediilor in care se executa
   - **Autonomia relativa**: au o independenta relativa fata de contextele de executie
     - Isi controleaza resursele interne
     - Interactioneaza cu entitati externe pe baza unor contracte

10. **Reutilizarea serviciilor**
    - Contin logica independenta
    - Reprezinta resurse reutilizabile
11. **Fara urmarirea starii**
    - Pot sa isi gestioneze singure starea sau pot transfera aceasta intrebuintare altor entitati
    - Se poate ca transferul intrebuintarii acesteia sa scada utilizarea resurselor
12. **Cuplare scazuta**
    - Decuplarea de mediul inconjurator se realizeaza prin faptul ca legatura cu un serviciu se face doar pe baza contractului acestuia
    - Ar trebui sa fie cat mai putin cuplate
13. **Limite precise**
    - Implementarile unui serviciu nu trebuie sa depaseasca limitele acestuia nici prin apeluri, nici prin dependenta de resurse pe care nu le controleaza explicit

Proiectare orientata pe dependenta

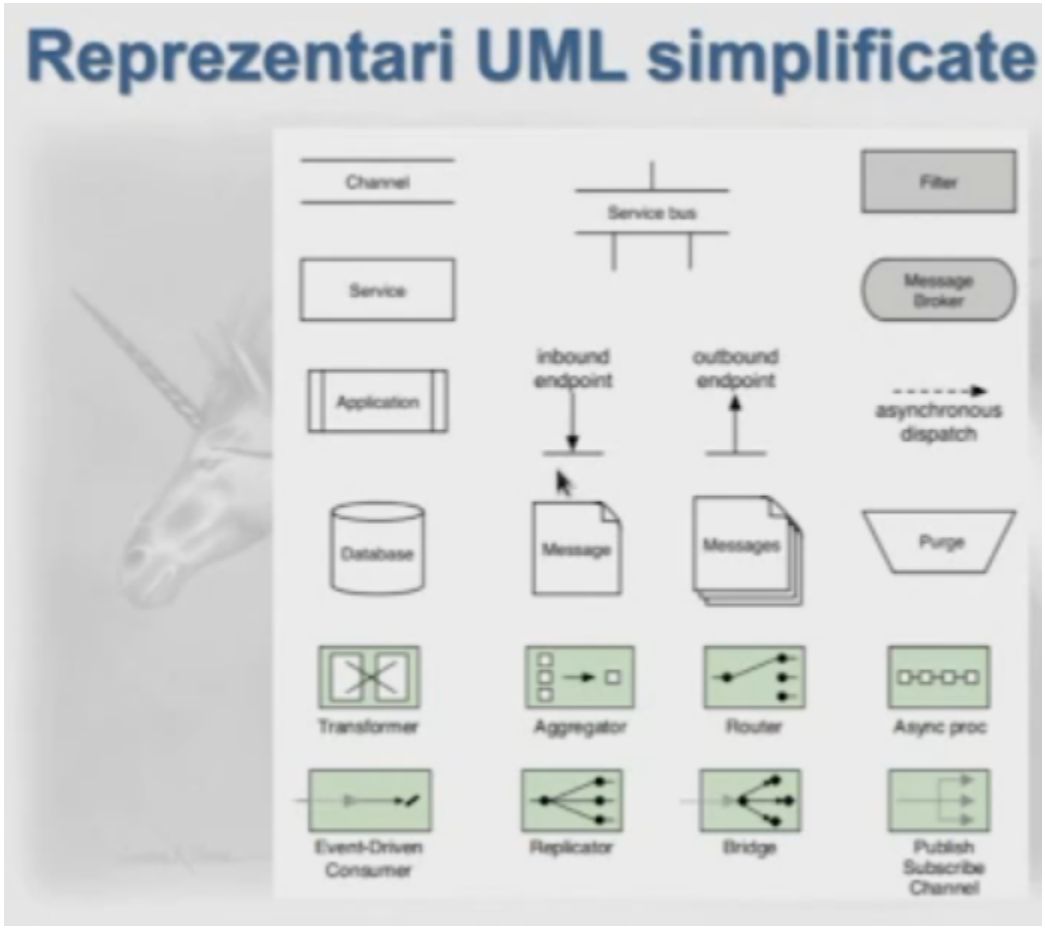
- Se realizeaza cu standardul TOGAF (pentru arhitecturi de intreprinderi)

Dependente:

- Externe si roluri
- La nivelul stratului de afaceri
- La nivelul aplicatiilor
    - Nu se refera la componentele ce se injecteaza in obiecte
    - Clarifica interactiunile dintre aplicatii
    - Model functional: aplica coeziunea pe domenii ca sa grupeze functiile de afaceri
    - Colectia de aplicatii (*portfolio*): prezinta implementari fizice ale aplicatiilor
- Servicii si produse
    - Servicii = grupari de operatii expuse si grupate, accesate prin interfete
    - Produse = grupari de operatii expuse si grupate in interiorul proceselor, accesate prin interfete
- La nivel de date
    - Internal data dependence: prezinta diagrame de pachet, clase, deployment etc
    - External data dependence:

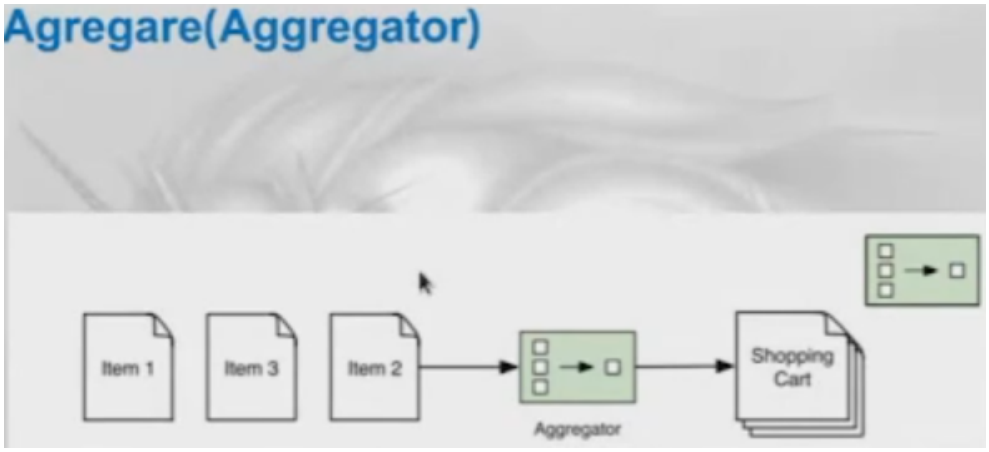
        - Entitatile de gestiune a datelor interne sunt slab cuplate cu cele de interfatare de date
        - Se realizeaza prin componentele ce ofera servicii
        - Interfetele spre exterior sunt bidirectionale
- La nivel tehnologic
    - Technology portfolio: urmareste elementele logice ale aplicatiei care sunt implementate la nivel tehnologic (~ ca diagrama de deployment)
    - Distributia tehnologiilor: cum tehnologiile sunt instalate in locatii diferite

Reprezentari UML simplificate



Modele de proiectare

Agregator/Combinator

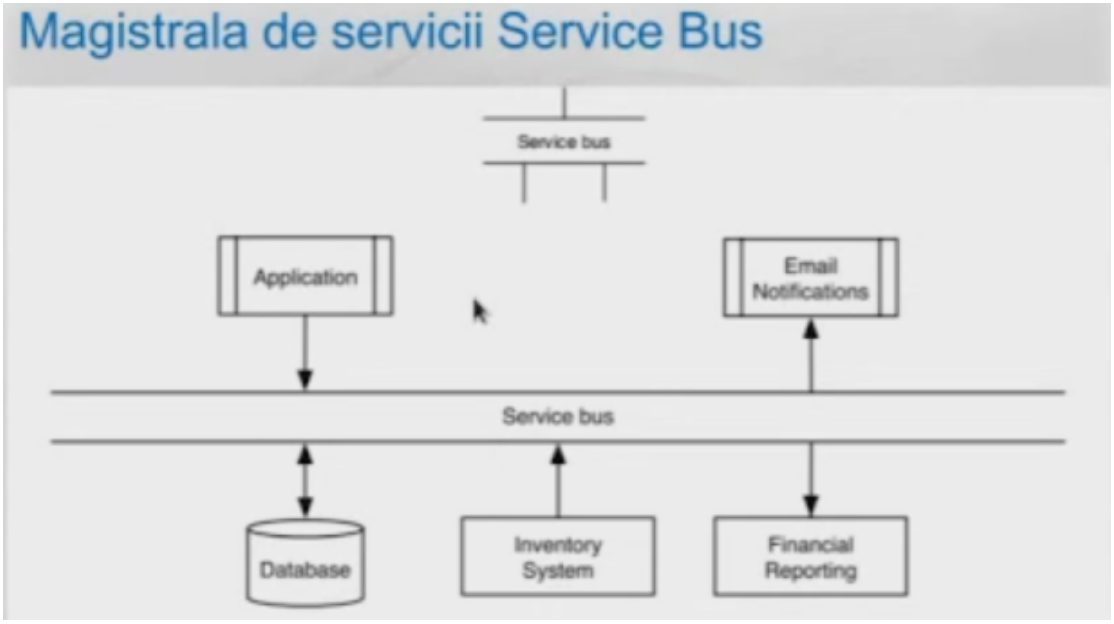


Agregare

Problema	Mesajele fara stare nu vor ajunge intr-o ordine predeterminata
Solutie	Se va defini un combinator care primeste un flux de date si grupeaza mesajele similare intr-o singura entitate care va fi trimisa unui punct de conectare .
Aplicare	Gruparea mesajelor vehiculate printr-o magistrala pentru servicii ...
Rezultate	Flexibilitatea in implementare pentru ca furnizorii individuali de servicii pot procesa asincron datele fara a exista o preocupare asupra starii sau secventei.

- Similar cu API:
  - Asemanare: ofera un contract ale caror functionalitati sunt oferite prin apelarea altor microservicii si compunearea raspunsurilor lor intr-o entitate de raspuns mai mare si mai complexa
  - Diferenta: nu e un punct de intrare in sistem (nu reprezinta nivelul de prezentare)
- Deobicei “se situeaza” intre straturile arhitecturii

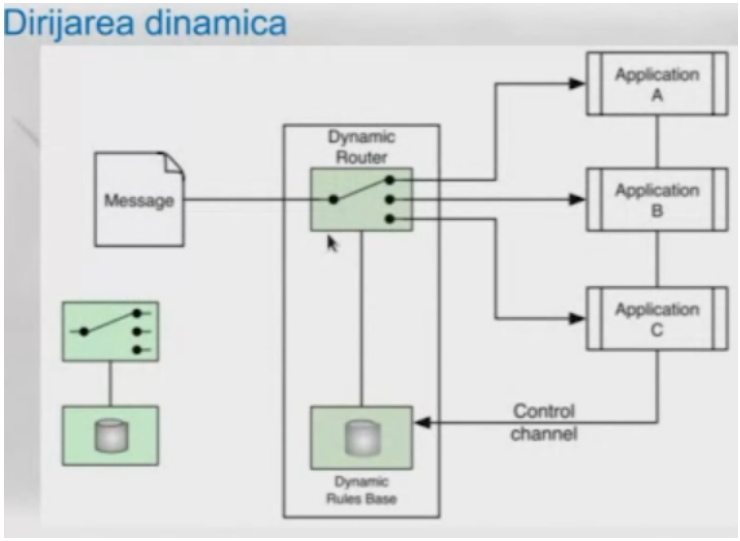
Magistrala de servicii (Service Bus)



Magistrala de servicii

Problema	Aplicatiile trebuie sa comunice intre ele utilizand de multe ori protocoale si tehnologii diferite.
Solutie	Utilizarea unui canal de comunicare indiferent la tipul datelor sau ale protocoalelor utilizate.
Aplicare	Integrarea sistemelor eterogene, interoperabilitate intre sistemele clasice si cele moderne, abstractizarea protocolului.
Rezultate	Message-Oriented Middleware (MOM): cozi de tip publicare/abonare - publish/subscribe queuing si magistrale de servicii pentru intreprindere.

Dirijare dinamica

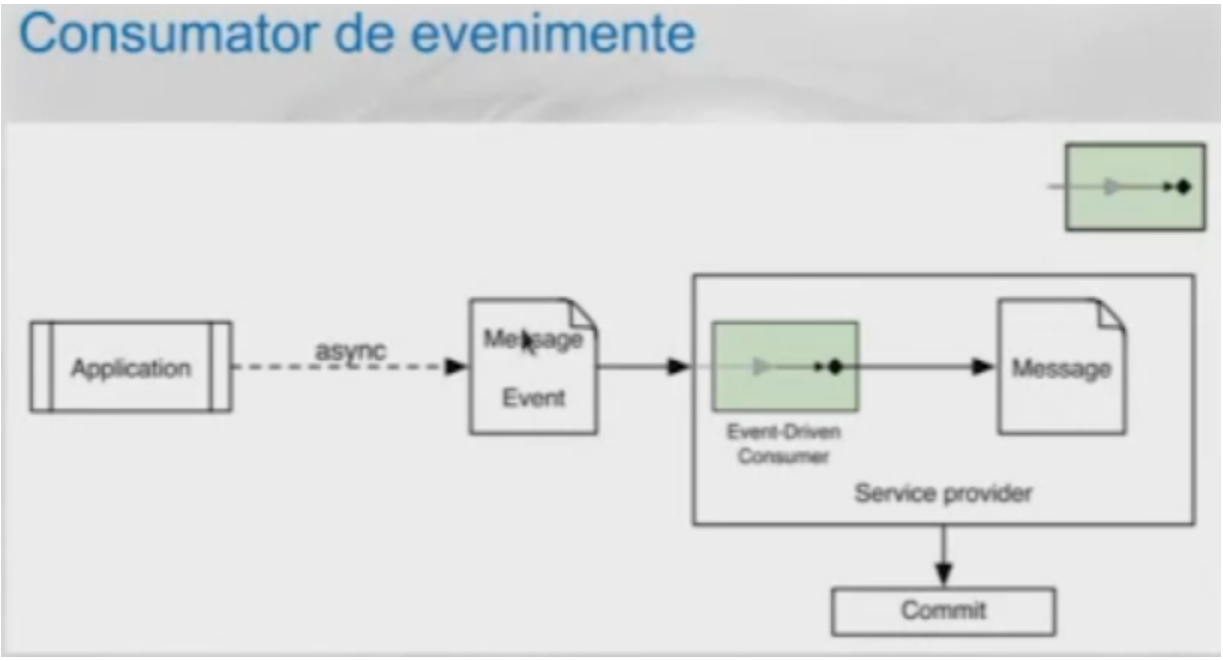


Dirijarea dinamică

Problema	Dirijarea mesajelor printr-un sistem distribuit utilizand reguli de filtrare este ineficientă
Solutie	Se defineste un dirijor de mesaje (router) care include atat regulile de filtrare cat si cunostiinte depre caile necesare pentru ca un mesaj sa ajunga la destinatie.
Aplicare	Livrarea mesajelor bazata pe o serie de informatii specifice din aplicatie
Rezultate	O performanta mai buna in procesarea si livrarea mesajelor care insa vine cu un cost suplimentar indus de complexitatea crescuta a sistemului de dirijare dar si de incarcare a lui.

- Dirijeaza traficul prin sisteme distribuite
- Cunoaste:
    - Destinatiile mesajelor
    - Rute optime catre acestea (de cost minim, fara noduri defecte)
- Se aseamana cu Dirijor (router):
    - Diferenta: acesta isi poate modifica regulile la runtime observand incarcarea si defectiunile microserviciilor

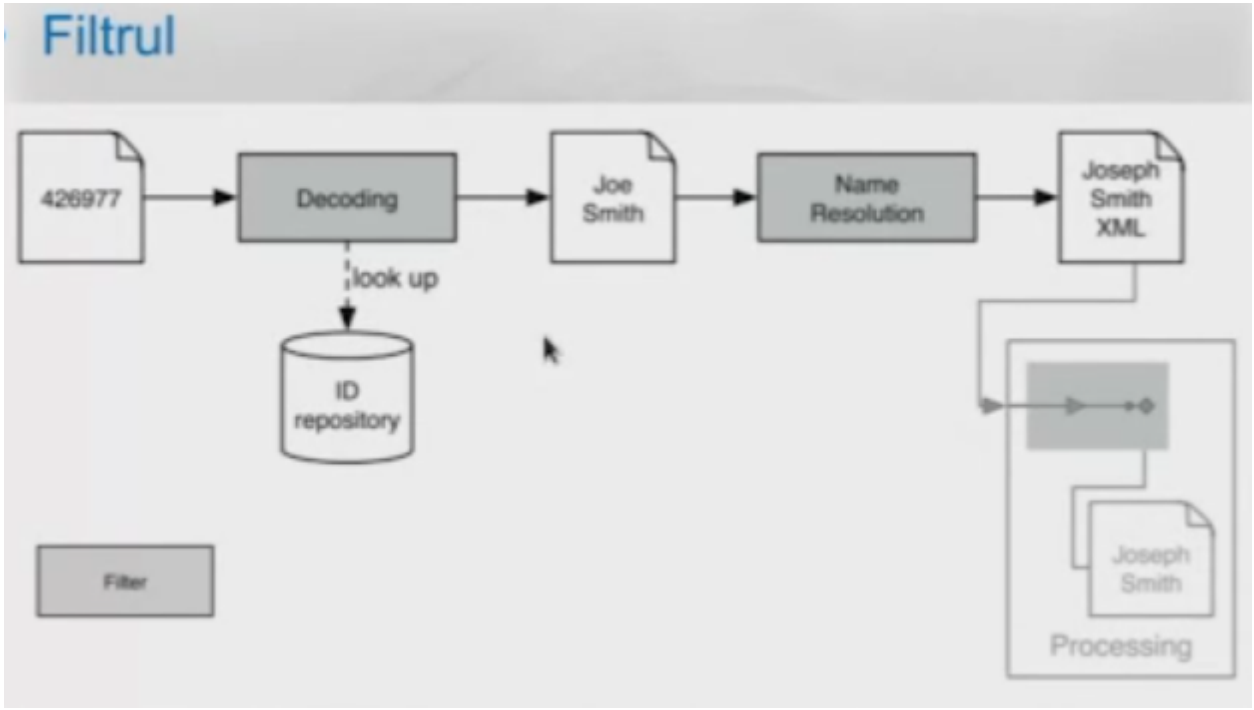
Consumator de evenimente



Consumator de evenimente

Problema	Sistemele de mesaje bazate pe receptori blocanti sau asteptare utilizeaza ineficient resursele de calcul cand canalul este gol.
Solutie	Implementatarea unui mecanism de tratare a evenimentelor bazat pe magistrala sau specific aplicatiei care este apelat numai cand primeste un mesaj.
Aplicare	Sistemele distribuite cu un set variabil de consumatori si furnizori de servicii care induc incarcari variabile la nivelul procesorului in functie de mesaj.
Rezultate	Procesarea mesajelor este scalata linear pe un singur fir cu numarul de mesaje trimise.

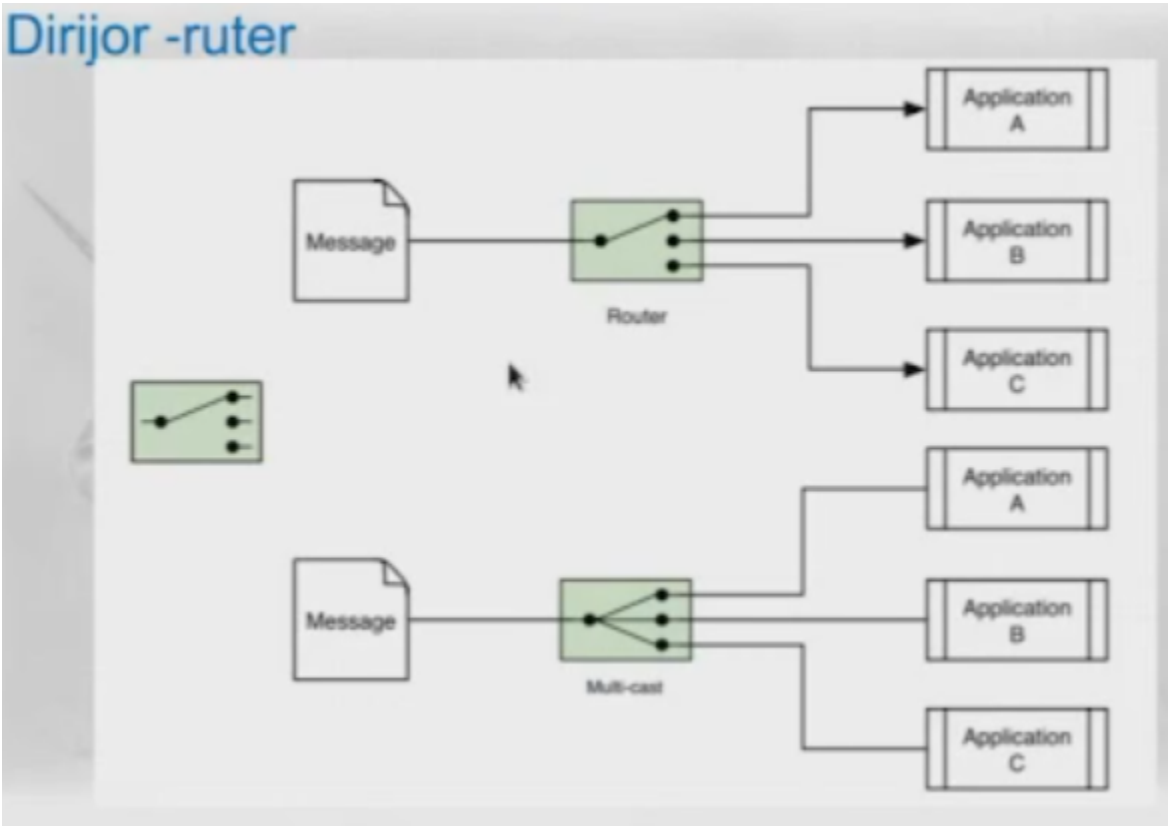
Filtrul



Filtrul

Problema	Necesitatea implementării într-o manieră independentă de platformă, fara cuplare stransă şi dependente suplimentare, a unui sistem flexibil de procesare a mesajelor schimbate intre sisteme.
Solutie	Implementarea unor canale cu o interfață simplă pentru intrare/procesare/iesire modelate dupa o funcție sau conducta (pipe) care permit compunerea unor filtre in stil lant de margarete
Aplicare	Utilizarea functiilor discrete asupra mesajelor.Filtrare imparte sarcini de calcul mai mari in unitati discrete mai usor de gestionat care pot fi recombinate pentru a fi utilizate de mai multi furnizori de servicii.
Rezultate	Filtrele elimina datele si dependentele prin definirea unui contract uniform care incurajeaza reutilizarea prin compunere.

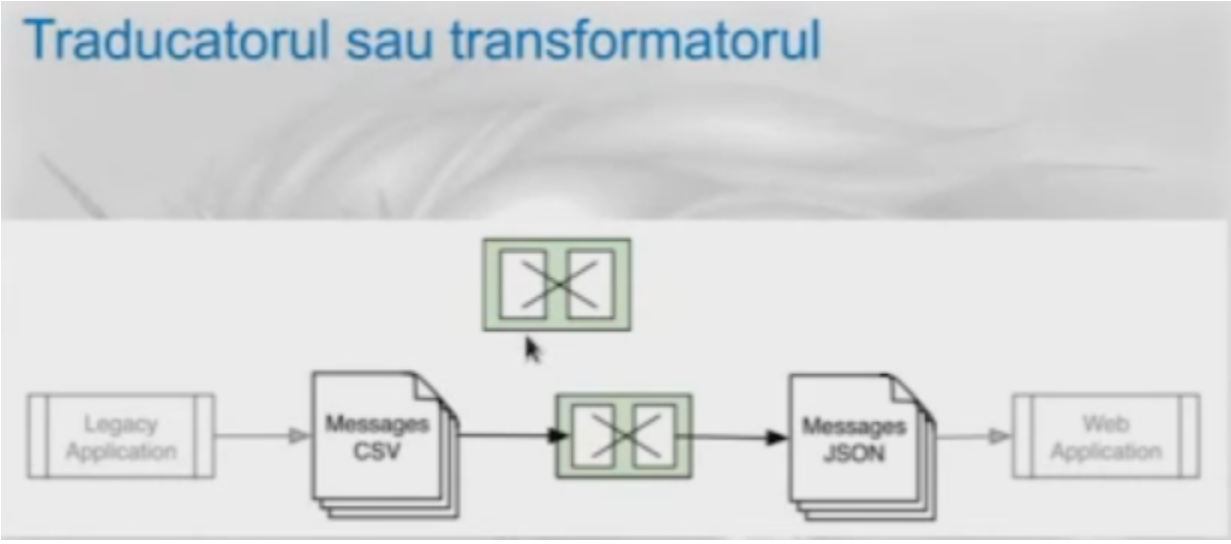
Dirijor



Dirijor

Problema	O aplicatie trebuie sa se conecteze cu una sau mai multe puncte de interconectare ale altor aplicatii dar fara a avea o cuplare stransa cu nici una dintre ele.
Solutie	Utilizarea unei conducte care permite regului de livrare configurabile care sunt bazate pe continutul mesajului,tipul acestuia sau filtre de date.
Aplicare	Livrarea de continut in magistralele de servicii, dirijarea mesajelor, intermediari de mesaje (proxy), aplicatii de integrare in intreprindere ..
Rezultate	Abtractizarea unui ruter este utilizata, sub diverse forme, in toate sistemele moderne SOA.

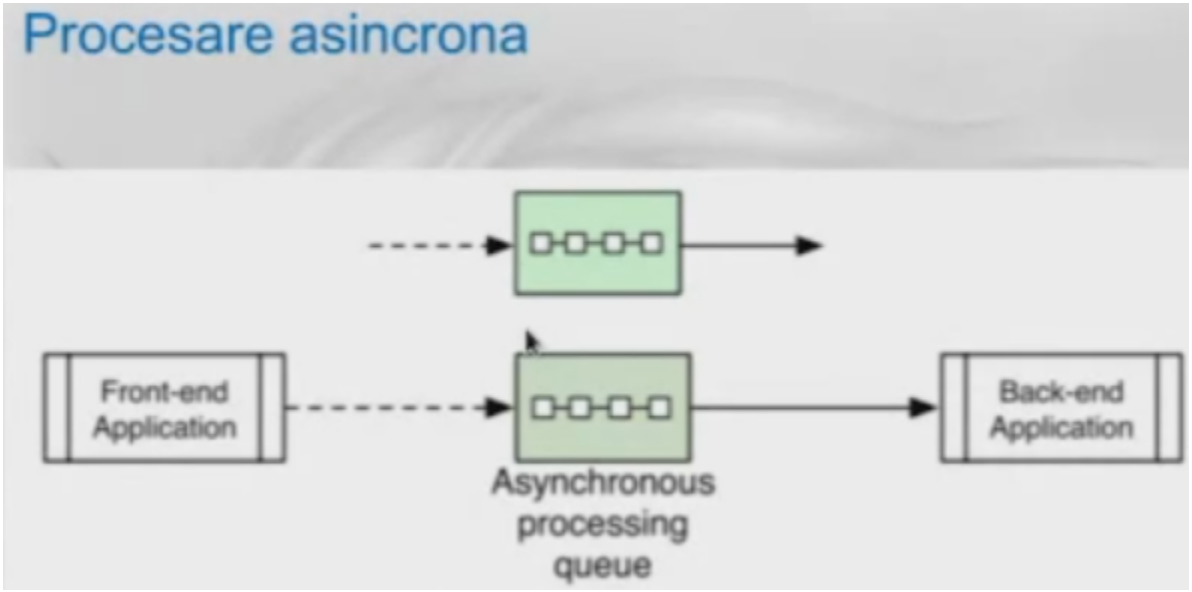
Traducatorul/Transformatorul



Traducatorul

Problema	Integrarea sistemelor eterogene care de obicei utilizeaza reprezentari diferite a datelor in mesaje si de multe ori nu suporta alte standarde sau numai un numari mic din acestea.
Solutie	Furnizeaza un mecanism independent de sistem care permite modificarea continutului mesajelor inainte de a le trimite catre un punct de intrare al unei alte aplicatii.
Aplicare	Traducerea mesajelor la nivelul punctului de intrare
Rezultate	Translatoarele sunt una din tehnicile cele mai eficiente de transformare a mesajelor

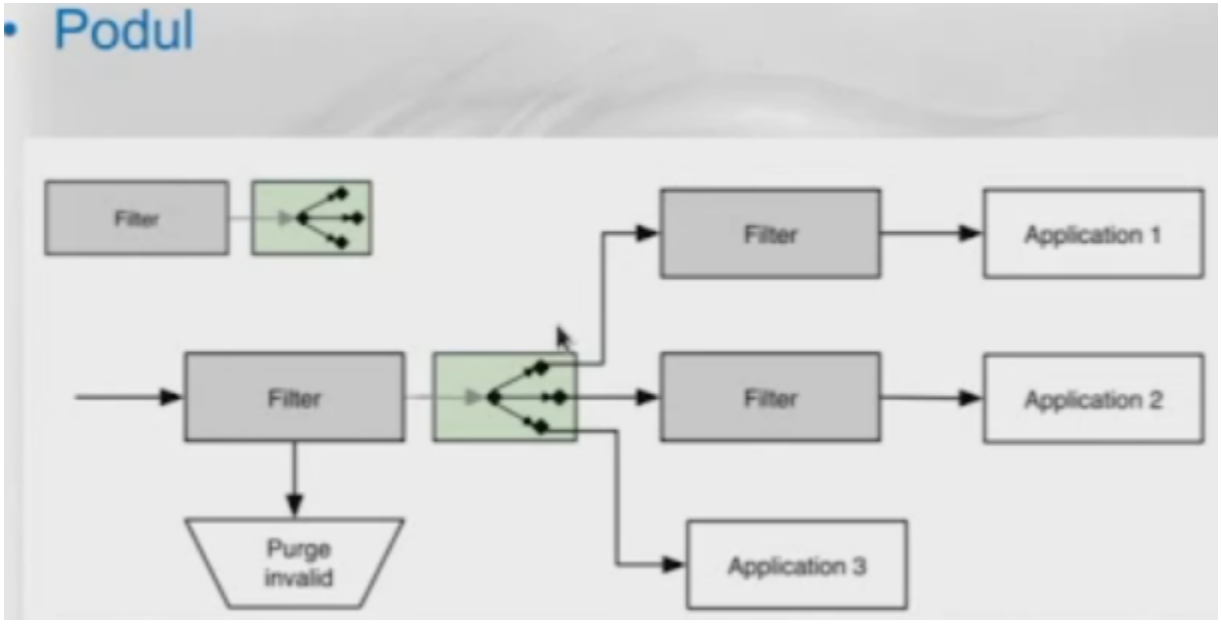
Procesare asincrona



Procesare asincrona

Problema	Procesarea sincrona poate conduce la performante scazute a unui server precum si la scaderi ale stabiliattii acestuia.
Solutie	Consumatorii vor schimba mesaje intre servicii prin intermediul unei cozi de procesare care decupleaza zonele de intrare de cele de procesarei.
Aplicare	In orice aplicatie care necesita o scalabiliate independenta pentru zonele de intrare fata de cele de procesare
Rezultate	Cozile de procesare sunt bine intelese si scaleaza corect atat orizontal cat si vertical in functie de necesitatile aplicatiei.

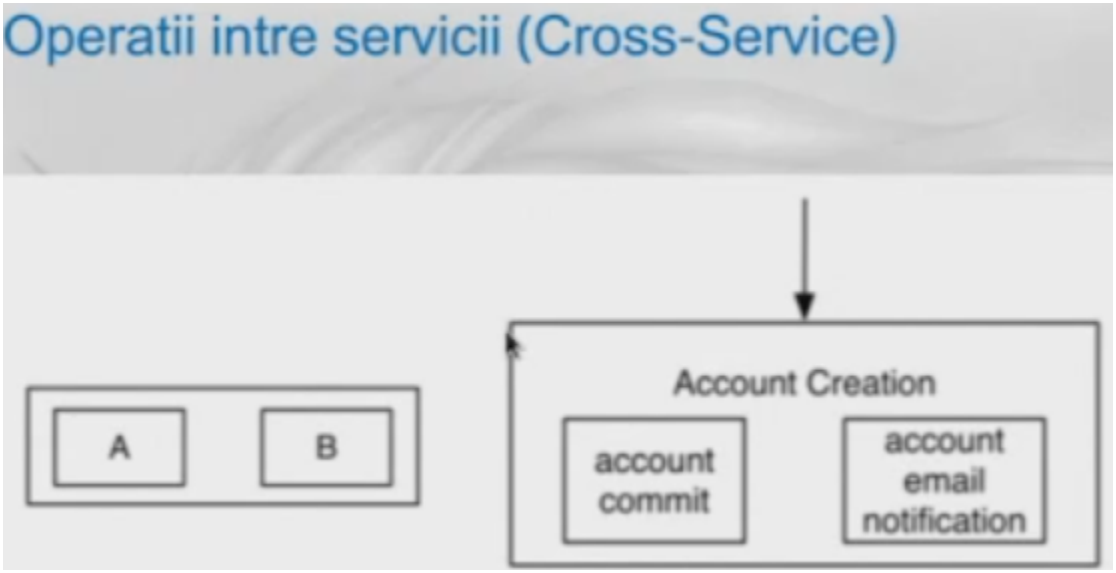
Podul



Podul

Problema	Punctele de intrare ale aplicatiilor pot sa existe in diverse parti ale retelei intreprinderii, sa foloseasxa protocoale diferite sau sa aiba nevoie de procesare bazata pe atribute specifice ale mesajului.
Solutie	Se defineste un pod intre aplicatii care furnizeaza un mecanism pentru dirijarea, filtrarea si transformarea mesajelor.
Aplicare	Pentru intermediari SOA intre punctele de contact ale aplicatiilor din cloud si cele din middleware sau back-end
Rezultate	Sunt bune pentru extinderea aplicatiei deoarece permit concentrarea dezvoltarii numai pe procesarile intermediare intre sisteme

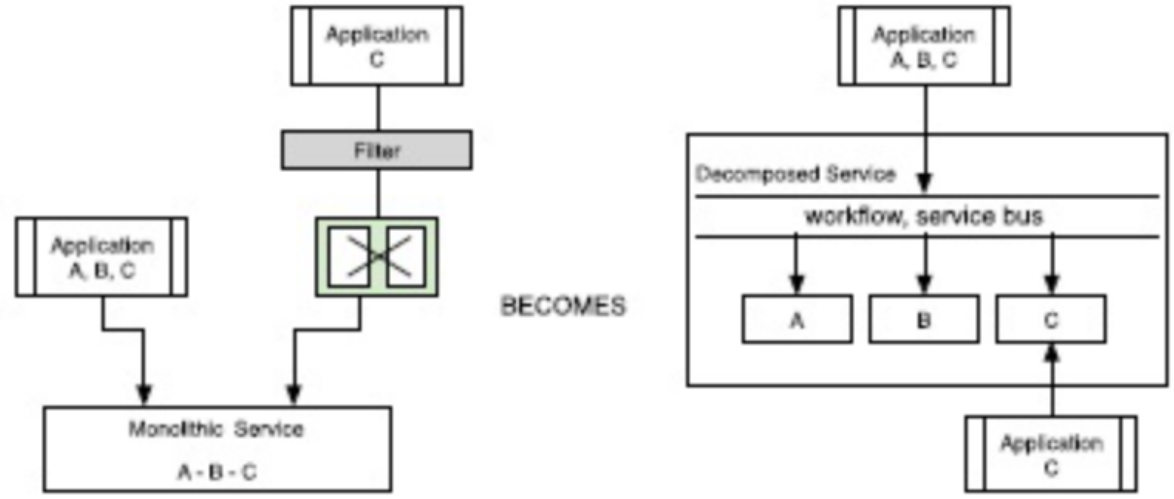
Operatii intre servicii (Cross-service)



Operatii intre servicii

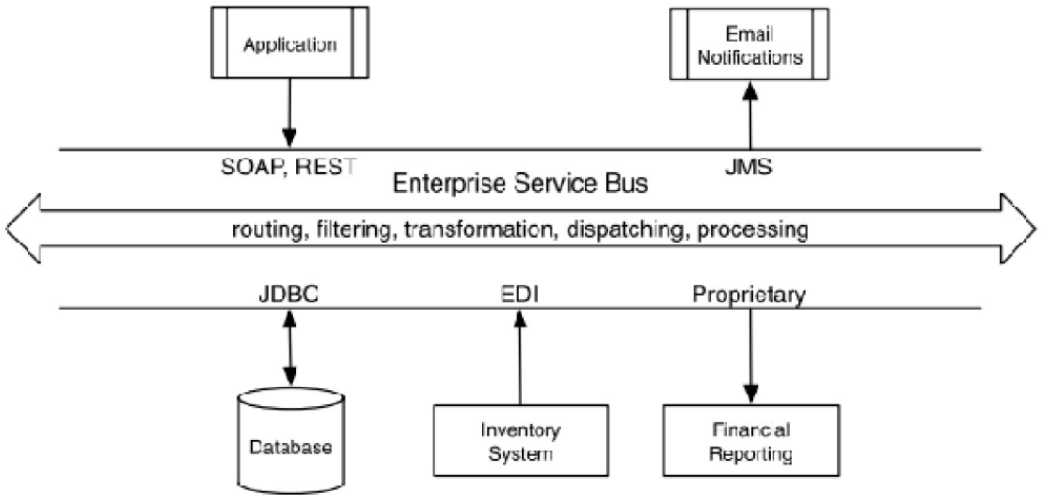
Problema	Avem doua sau mai multe servicii care se pot executa chiar pe mai multe sisteme care trebuie sa am garantia ca se vor termina corect.
Solutie	Serviciile granulare pot fi incapsulate in alte servicii care furnizeaza mecanisme de verificare a integritatii si asigura terminarea corecta a acestora sau in cel mai rau caz o esuare controlata in caz ca avem o cadere.
Aplicare	Sisteme tranzactionale.
Rezultate	Poate fi necesara utilizarea unui procesor de tranzacti care sa furnizeze mecanismele necesare acestui proces de incapsulare si sa interopereze cu restul infrastructurii de tip SOA.

Descompunerea



- Utila cand incarcarea e prea mare si cand se schimba logica de business
- Se pastreaza contractul, functionalitatile prezentate de contract sunt delegate componentelor rezultate din descompunere
- Cloud-ul duce serviciile foarte mari pe masini mai puternice si costurile cresc considerabil
- Dupa descompunere se obtin servicii mai simple
- Se pot reutiliza acele servicii mai usor decat serviciul urias initial
- Costuri de intretinere mai mici

Magistrala pentru serviciile de intreprindere (ESB - Enterprise Service Bus)



- Transmite din orice punct in orice punct
- Permite transformari si filtrari
- De la Sun

Servicii tolerante la erori

- In trecut: replicare => Daca crapa unul cererile se paseaza altuia
- Acum: se incapsuleaza servicii in alte servicii care ofera urmarire
    - Servicii fara stare devin cu stare
    - Jurnalizare => In caz ca serviciul incapsulat crapa se poate reporni + i se poate restabili starea din mers

Adaptor

- Ex: Incapsulare API
- La sisteme clasice in general
- Simuleaza existenta unui contract de care doreste sa depinda clientul, asigurand functionalitatile prezentate in contract utilizand serviciile incapsulate

Anuntarea incetarii definitive a functionarii unui serviciu

…

Curs 5

Grid vs Cloud

- Grid computing presupune stocarea de cantitati masive de date pe masinile din grid
- Cloud computing presupune accesarea datelor prin intermediul unor servicii cu ajutorul internetului

Plasarea norului dpdv al tehnologiei

- Este plasat la intersectia dintre:
    - Virtualizarii hardware
    - Aparitiei chips-urilor multi-core
    - Calculului de retea

    - Dezvoltarii calculului autonomic si a automatizarii centrului de date

IaaS (Infrastructure as a Service)

- Serviciu de Cloud computing care ofera o infrastrctura (resurse esentiale de calcul, de stocare si de retea la cerere) pe baza de plata

Translare binara

- Aplicatiile sunt executate direct de procesor
- Translare: cand parti din cod sunt executate de kernel
    - Programatorul uneori nu are privilegiile necesare unor actiuni
    - Acesta realizeaza apeluri sistem, lasand kernel-ul sa execute actiunile
- Se incearca sa se realizeze cat mai putine translari

Virtualizare

- Abstractizarea sistemului de operare, a aplicatiilor, a zonelor de pastrare a datelor sau a retelei de comunicatii fata de suportul hardware si software care permit acest proces
- Daca are suport hardware se elimina translatarea binara
- Tipuri:
    - Completa
    - Paravirtualizare
    - Hibrida

Virtualizare completa

- Masina virtuala simuleaza complet un hardware
- Permite unui sistem de operare clasic sa fie executat intr-o maniera izolata
- Foloseste translarea binara
- Mai inceata
- Ex: sisteme Microsoft si Parallels

Hypervisor de tip 2 (gazduit)

- Manager de masini virtuale
- Instalat ca aplicatie software pe un sistem de operare (SO) existent

Paravirtualizare

- Hypervisor-ul este instalat direct pe SO

- Sistemele de operare gazduite nu sunt complet izolate
- Foloseste *hypercalls* in loc de translare binara
- Mai rapida
- Nu se mai pot folosi SO native (nu se mai face pe hardware, ci exista drivere pentru comunicare cu hypervisor-ul)
- Ex: VMWare, Xen

Virtualizare hibrida

- O imbinare intre virtualizarea completa si paravirtualizare

Virtualizare la nivel de SO

- Software-ul masinii virtuale se instaleaza in sistemul de operare al gazdei
- Nu se instaleaza direct pe hardware

Cloud computing

- Model de plata in functie de utilizare
- Permite accesul pe baza de retea la resurse de calcul configurabile (baze de date, servere etc)
- Administrare minimala

Caracteristici nor

- Auto-service
    - Norul ofera cate resurse sunt necesare executiei corecte a aplicatiei
    - Aceasta poate fi mutata pe o alta masina in cazul in care consuma prea mult
    - Pretul se modifica in functie de masina pe care se executa
- Acces de oriunde la retea
- Grupare a resurselor independente de locatie
- Plata dupa cat consumi

Componente nor

- Clienti
- Centre de date
- Servere virtualizate
    - Cand un server este replicat (*scalat*) pe orizontala (se creeaza mai multe instante ale serverului)
    - Instantele pot fi pe masini sau chiar cloud-uri diferite

    - Se afla in spatele unui Load Balancer care inainteaza cererile spre servere astfel incat incarcarile sa ramana de valori ~ egale
- Servere distribuite

Tipuri de nor

- Public
    - Doar inchiriat
    - Nu e administrat de cei care il inchiriaza
- Privat
    - Detinut si administrat de client
- Hibrid
    - Nor format din 2+ nori de tipurile 1 si 2

SaaS (Software as a Service)

- Model de distributie software
- Cloud gazduieste aplicatii si le pune la dispozitia utilizatorilor finali pe internet
- Ex: Google Docs, Google Slides, MS Powerpoint Online
- Caracteristici:
    - Model multi-tenancy
    - Provizionare automata
    - O singura logare
    - Facturare bazata pe abonament
    - Disponibilitate ridicata
    - Infrastructura elastica
    - Securitatea datelor
    - Securitatea aplicatiei

aaS (as a Service)

- Acronim
- In contextul cloud computing
- ?aaS = ceva oferit ca serviciu unui client

PaaS (Platform as a Service)

- Mediu complet de dezvoltare si implementare in Cloud

Controlul userului:

Minim			Maxim
SaaS	PaaS	IaaS	Cloud Privat

STaaS (STorage as a Service)

- Model de afaceri de stocare a datelor
- Se plateste doar spatiul de stocare de care este nevoie, atunci cand este nevoie de el
- Ex: GoogleDrive, OneDrive

DaaS (Development as a Service)

- Instrument de dezvoltare bazat pe web comun
- Mediul de executie, impreuna cu instrumentele de dezvoltare integrate, sunt oferite pentru dezvoltarea aplicatiilor.
- Ex: SimpleDS, Cloud9

DBaaS (DataBase as a Service)

- Ex: Amazon Aurora, Google Cloud Datastore

Information as a Service

- O companie imparte sau vinde informatii relevante unei alte companii sau persoane fizice pentru a-si desfasura activitatea
- Ex: Burse, Google Maps

CaaS (Communication as a Service)

- Ex: Skype, Discord, Google Meet

IDaaS (IDentity as a Service)

- Permite organizatiilor sa utilizeze autentificare unica

MaaS (Monitoring as a Service)

- Faciliteaza implementarea functionalitatilor de monitorizare pentru diverse alte servicii si aplicatii din Cloud
- Ex: Amazon CloudWatch, Azure Monitor

Integration as a Service

- Se straduieste sa conecteze date la fata locului cu date situate in aplicatii bazate pe Cloud. Aceasta paradigma faciliteaza schimbul de date si programe in timp real intre sistemele la nivel de intreprindere si partenerii comerciali.

TaaS (Testing as a Service)

BaaS (Backup as a Service)

- Metoda de stocare a datelor in afara site-ului in care fisierele, folderele sau intregul continut al unui hard disk sunt efectuate in mod regulat de catre un furnizor de servicii pe o telecomanda depozit de date securizat bazat pe Cloud printr-o conexiune de retea

FaaS (Function as a Service)

- Ofera optiunea de a executa cod precum Javascript, Linux sau HTML ca raspuns la evenimente fara a fi nevoie sa construiasca vreo infrastructura complexa (adica infrastructura asociata cu construirea si lansarea de aplicatii de microservicii)

XaaS (Anything as a Service)

- Ex: Amazon Web Services (AWS)

Platforma hiperconvergenta

- Infrastructura definita de software care virtualizeaza toate elementele sistemelor conventionale.
- Hardware x86 cu VMWare
- Scalabilitate mare
- Complexitate mai mica
- Include
    - calcul virtualizat (un hypervisor)
    - stocare definita de software
    - retea virtualizata (retea definita de software)
- Pro
    - Executie integral software, nu se depinde de HW

    - Virtualizare stocare si calcul
    - Gestiune automatizata
- Contra
    - Scumpe
    - Centralizate
    - Mai putin sigure decat nor
    - Necesita specialisti

Nor - detalii tehnice

- Infrastructura cu 4 straturi
- Replicare:
    - Orizontala: se creeaza mai multe instante ale entitatii software (pe masini diferite eventual) => costuri mai mici
    - Verticala: se duce aplicatia pe o masina mai puternica => costuri mai mari
- Include supervizor
- Hiperconvergenta? (adineauri se spunea ca norul ar fi mai sigur decat platforma hiperconvergenta, nu stiu cum ar putea fi norul insusi o platforma hiperconvergenta…)

Nivel prezentare nor

- Pagini web => informatii despre servicii
- Infrastructura elastica =>
    - QoS (Quality of Service)
    - Hipervisorul replica pe orizontala

Nivel logic nor

- Virtual networking => acces de la distanta
- Message oriented middleware
    - Reduce complexitatea de deasupra lui
    - Sterge mesajul cand destinatia confirma primirea

Nivel date

- Redundanta => probleme la consistenta copiilor
- Izolare mai stricta => copii consistente

Elasticitate in nor

- Implemantata in business-ul norului
- Are 5 dimensiuni
- Definita ca gradul in care un sistem este capabil sa se adapteze la modificarile volumului de munca prin aprovizionarea si deconectarea resurselor intr-o maniera autonoma, astfel incat, in fiecare moment, resursele disponibile sa se potriveasca cu cererea actuala la fel de strans pe cat posibil.

Arhitectura slab cuplata

- Scalare independenta
- Verticalitate/orizontalitate
- Suporta orice nivel de granularitate

Curs 6

Imutabilitatea microserviciilor

- Atacatorii exploateaza adesea accesul la shell oferit imaginilor pentru a injecta cod rau intentionat
- Acest lucru poate fi evitat prin crearea de containere imutabile

Avantaje arhitecturi orientate pe evenimente

- Nu este nevoie de o mapare relationala a obiectelor
- Cu fiecare schimbare externa capturata ca eveniment starea curenta poate fi repetata
- Asigurarea persistentei se face fara modificari

Probleme JEE

- Asamblarile hibride
- Framework-urile clasice
- Containerele JEE
- JBoss & Spring | PicoContainer

Cauze:

- Influentata de gandirea clasica in sisteme distribuite
- Modelul de thread-uri nu aduce niciun beneficiu

- Framework-ul nu suporta nativ abordarile bazate pe data streams
- Nu suporta rezilienta
- Implementata in containere masive
- Fara suport pentru imutabilitate
- Limitari in alegerea persistentei
- ESB-ul este limitativ

Microserviciile = stil arhitectural care structureaza o aplicatie ca o colectie de servicii care sunt foarte mentenabile si testabile, slab cuplate, implementabile independent.

Argumente microservicii

- Pro
  - Echipa relativ mica
  - Diverse limbaje
  - Integrare usoara si automatizarea instalarii
  - Usor de inteles si modificat
  - Tehnologii moderne
  - Urmeaza gandirea afacerii
  - Un container are instalare si pornire rapida
  - Modificari minimale
  - O mai buna toleranta la erori
  - Sunt usor de scalat si de integrat
  - Nu exista dependenta de o stiva tehnologica
- Contra
  - Testarea poate deveni complicata
  - Lipsa de comunicare
  - Arhitectura aduce complexitate aditionala
  - Probleme ca sistem distribuit
  - Integrarea si gestiunea mai complicata
  - Problemele de complexitate ale unui produs monolit si problemele de complexitate ale sistemelor distribuite
  - Mecanismele de comunicare intre servicii
  - Extindere servicii vs. tranzactional

Monolit

- Aplicatie software cu un singur nivel in care interfata utilizatorului si codul de acces la date sunt combinate intr-un singur program de pe o singura platforma.
- Autonoma si independenta de alte aplicatii de calcul.
- Poate avea granularitatea intalnita la MSA (**M**icro**S**ervice **A**rchitecture) dar este executata tot pe o singura masina
- Tipuri:
    - Cu proces unic = monolit cu un singur modul
        - Ex: la arhitecturi client-server
    - Modular = 2-3 niveluri
        - Ex: la aplicatii vechi cu paradigma programarii orientate pe module
    - Distribuit
        - Indeplineste la limita principiile SOA
        - Are toate dezavantajele sistemelor distribuite si ale monolitilor
        - Ex: JEE+REST (doar prezentarea e distribuita)
- Probleme:
    - Limitare in dimensiune si complexitate
    - Prea mare si complexa pentru noi modificari
    - Poate porni greu
    - La fiecare update trebuie redistribuita toata aplicatia
    - Cuplare mare
- Avantaje:
    - Foarte usor de testat si depanat
    - Teste end-to-end mult mai rapide
    - Mai simplu de implementat (dispare/se reduce logica de comunicare a componentelor prin retea)
- Imbunatatiri:
    - Descompunere aplicatie:
        - Se extrage modulul mai slab proiectat (care are o incarcare mare si care face ca intreaga aplicatie sa fie mutata pe un nod foarte puternic)
        - Se reproiecteaza modulul asigurandu-se comunicarea cu celelalte

        - Interfata sa se pastreaza asemanatoare cu cea veche pentru backwards compatibility (chiar daca ar putea avea o interfata mai buna datorita reproiectarii)
    - Se extrag modulele care ar avea nevoie de scalare
    - Se reproiecteaza aplicatia dupa MSA (modelul de proiectare “sugrumator” (*strangler*) de la MSA) => 1 DB pe MS
    - Se grupeaza MS in servicii si servicii in servere

Sisteme de tip “Cutie neagra”

- Un sistem despre care se stiu doar intrarile si iesirile, fara logica din interior
- Implementarea este opaca

Proiectare MSA

1. Definire domeniu
    - Intrebari despre utilizare:
        i. Unde?
        ii. Cand?
        iii. De catre cine?
        iv. Incarcare maxima asteptata?
2. Pas 2 (nu vad sa aiba nume…)
    - Se preiau informatii de la utilizatori prin diagrame de use case
    - Intrebari:
        i. Timpul maxim de raspuns pentru unele zone?
        ii. Disponibilitatea unui microserviciu?
        iii. Multe altele…
3. Pas 3
    - Diagrama prea blana…
4. Pas 4
    - Alegere SGBD
    - Unde vor fi pastrate datele?
    - Asigurare coerenta
    - Aplicare teorema CAP

Ontologia Zachman

- Structura fundamentala pentru arhitectura intreprinderii
- Ofera un mod formal si structurat de vizualizare si definire a unei intreprinderi

Teorema CAP = intr-un sistem distribuit, un DB are cel mult 2 dintre calitatile:

- Consistenta
    - Se citeste mereu cea mai recenta valoare/eroare
- Disponibiliatate
    - Raspuns rapid
    - Nu neaparat cu ultimele versiuni ale datelor…
- Partitionare (toleranta la erori)
    - Continua sa functioneze chiar la pierderea unor mesaje

Baze de date

- Relationale:
    - Multe tabele
    - Utile cand datele au legaturi intre ele
    - Permit join-uri complexe
- NoSQL
    - Multe date
    - Schema nu e fixa
    - Fara join-uri complexe
    - Utilizate in cadrul procesarilor Big Data

Ascunderea informatiilor = protejarea datelor si logicii de business ale unui serviciu de exterior

Tipuri de cuplare

- In implementare:
    - Ex: incapsulare mai multe servicii intr-un API
- Temporala:
    - Mesajele trebuie sa curga intr-o ordine, de la un MS la altul
    - Deci toate MS trebuie sa fie disponibile (inclusiv reteaua)
    - Solutii:
        - Broker de mesaje
        - Cozi de mesaje
        - Apeluri asincrone
        - Cache local
- La instalare:
    - MS nu au cuplare la instalare ca monolitii

        - Cand un MS se modifica, atunci doar el trebuie reinstalat, nu si toate celelalte
        - Cand un modul din monolit se modifica se reinstaleaza toata aplicatia
    - Codul e unic in fiecare MS
- Cuplare de domeniu
    - Designul arhitecturilor MSA imita structura organizationala a intreprinderii
    - Informatia curge prin tot sistemul
    - Solutii:
        - Reanalizare si simplificare mesaje
        - Aparitie evenimente asincrone
        - Introducere agregat
            - Colectioneaza mesaje
            - 1 MS gestioneaza minim 1 agregat

Context marginit (bounded context)

Arie dintr-o organizatie mare

- Trebuie sa aiba responsabilitati clare
- Trebuie ascunse detaliile
- Poate contine mai multe agregate
- Poate depinde de mai multe contexte marginite

API (Application Program Interface)

- Forteaza interoperabilitatea
- Adaptare mai usoara
- Vazut ca model de proiectare

Curs 7

Principiile SOLID

- Responsabilitate unica
    - 1 MS trebuie sa indeplineasca o singura functie
    - Trebuie sa fie suficient de modular pentru reutilizare
    - Ar trebui sa existe orchestrator
- Inchis-deschis

    - Un MS nu trebuie modificat pentru oferire de functionalitati situationale/marginale
    - Ar trebui sa fie apelat de catre alt MS care adauga acele functionalitati
    - Proces asemanator cu derivarea de la OOP
- Substitutia Liskov
    - O versiune noua de MS ar trebui sa poata inlocui versiunea veche
    - Apelantii nu trebuie sa se modifice
- Segregarea interfetelor
    - Un MS nu ar trebui sa expuna metode care nu au legatura directa intre ele (ex: facturare si plata, comandare si plata)
- Control invers (la OOP parca era Dependenta inversa…)
    - Un MS nu ar trebui sa apeleze direct alt MS
    - Se poate folosi un modul de descoperire a MS
    - Se poate utiliza o coada de mesaje pentru apelare

Cubul scalarii MS

- Model de scalabilitate pentru MS
- Are 3 axe:
    - X: descompunere orizontala
    - Y: descompunere verticala
        - Cand se utilizeaza mai multe resurse
        - In functie de cat de mult sunt utilizate/cat de multe calcule fac
    - Z: descompunere functionala
        - Seamana cu X
        - Diferenta: selectia mesajelor pe care un MS le primeste
        - Se utilizeaza topic-uri (uneori si geolocatia)

Steaua mortii

- Anti-model de proiectare
- MS devin extrem de interdependente
- Intarzieri mari
- Fragilitate
- Inflexibilitate

Caching

- Intermediar in care raspunsurile sunt interceptate si stocate in memorie

- Reaccesarea rapsunsurilor e mai mult mai rapida
- Cache-ul gaseste raspunsul pentru cerere in memorie => HIT
- Cache-ul nu gaseste raspunsul pentru cerere in memorie => inainteaza cererea spre destinatar => MISS
- Tipuri:
    - Incapsulat (embedded)
        - Vin cereri la Load Balancer
        - Sunt redirectionate catre un serviciu
        - Serviciile verifica daca nu au deja raspunsul in cache-urile proprii
        - Daca da, intorc valoarea din cache
        - Daca nu, trateaza cererea (proces care dureaza ceva mai mult), stocheaza rezultatul in cache si il intorc
    - Comun (incapsulat distribuit)
        - Toate cache-urile grupate intr-un cluster
        - Ex: Hazelcast Cluster (scris in Java, interopereaza cu Spring)
    - Cache client-server
        - Vin cereri la Load Balancer
        - Sunt redirectionate catre un serviciu
        - Serviciul foloseste un Cache Client pentru a se conecta la Cache Server
        - In cazul in care raspunsul e in cache, acesta este intors direct
        - Daca nu se proceseaza cererea, se stocheaza raspunsul in cache si se intoarce raspunsul
    - Cache in nor
        - Ca la client-server
        - Serverul este in nor
        - Ex: Hazelcast Cloud
    - Atasamentul (sidecar)
        - Cererea vine la LB (Kubernetes) si e redirectionata catre un POD
        - Cererea ajunge in containerul aplicatiei
        - Se foloseste cache client pentru utilizarea cache-ului
        - Cache Server-ul e pe aceeasi masina, intr-un alt container => Cache local => latenta scazuta, descoperirea nu mai e o problema (mereu disponibil la localhost)
    - Proxy invers

        - Solicitarea vine la LB
        - LB verifica daca raspunsul e in cache
        - …
        - Avantaj: configurare cache din exterior, fara a modifica serviciile
        - Dezavantaj: serviciile nu pot invalida memoria cache
    - Proxy invers + sidecar
        - Cererea vine la LB (Kubernetes) si e redirectionata la POD-uri
        - In POD, containerul proxy invers (nu containerul de aplicatii) primeste cererea
        - …

Descoperire

- La nivel de client:
    - Prin Service Registry
    - Clientul obtine detaliile de conectare la unul dintre serviciile dorite
- La nivel de serviciu:
    - Serviciul client trimite o cerere de conectare la alt serviciu catre LB/API Gateway,
    - LB/API obtine adresa serviciului destinatie folosind Service Registry
    - LB/API inainteaza cererea

Autoinregistrarea unui serviciu

- La pornirea unui serviciu, acesta se inregistreaza in Service Registry
- La inchidere, acesta cere Service Registry-ului sa il scoata din lista de servicii disponibile

Urmarire distribuita

- Capacitatea unei solutii de urmarire de a urmari si observa cererile
- Compune stratul de instrumentatie
- Prezinta puncte de culegere de informatii (ex: API-uri)

Ordonarea temporala Lamport

- Algoritm simplu utilizat pentru a determina ordinea evenimentelor dintr-un sistem distribuit
- a->b = a s-a intamplat inainte de b

- Ex: Docker-Jaeger, Spring-Trace

Observabilitate intr-o plasa de MS

- Masurarea, colectarea si analiza diferitelor semnale de diagnosticare dintr-un sistem

Upfront sampling

- Bazata pe cap (head-based/upfront sampling), o decizie de esantionare va fi aplicata unei singure urmariri la initierea acelei urmarire
- Datele de urmarire vor fi fie incluse in setul de esantioane, fie eliminate pe baza logicii initiale, indiferent de ceea ce este inregistrat in urmarirea completata.

Probabilistic sampling

- In Jaeger, sampling-ul probabilistic ia o decizie de esantionare aleatorie cu probabilitatea esantionarii egala cu valoarea proprietatii sampler.param.

Sampling-ul cu limitarea ratei

- In Jaeger, sampling-ul cu limitarea ratei foloseste un limitator de rata pentru a se asigura ca urmaririle sunt esantionate cu o anumita rata constanta.

Cei 3 piloni ai observabilitatii

- Metricile (metrics)
- Log-urile (logs)
- Urmarirea distribuita (distributed tracing)

Planul Brown

- Format din:
  - Stratul Cross-Cutting (Cross-Cutting Tools, Baggage Definition Language), accesat de developeri Cross-Cutting Tools
  - Stratul Baggage (Data Type Encodings, Nested data, Multiplexing), accesat de developeri Cross-Cutting Tools
  - Stratul Atom (Cross-Cutting Tools, Merging, Trimming), accesat de developeri App & Framework
  - Stratul Transit (Instrumental Systems, Propagate opaque Baggage Context), accesat de developeri App & Framework

Pipeline cu data mining

- Componente:
  - Microserviciile
  - Urmarirea backend-ului (Tracing backend)
  - Declansatorul finalizarii urmaririi (Trace completion trigger)
  - Extractorul de caracteristici (Feature extractor)
  - Agregatorul (Aggregator)
  - Spatiul de stocare

Modele de migrare a aplicatiei

- Cei 6 de R:
  - Rehost
    - “Lift and shift”
    - Rapid
    - Se muta totul in nor
    - Costuri initiale mici
    - Costuri mari pe termen lung datorita incarcarii
  - Replatform
    - Apar anumite actualizari in timpul migrarii:
      - Schimbare SGBD cu cel nativ al norului
    - Acestea favorizeaza gazduirea aplicatiei de catre nor => scalabilitate, securitate, profit
  - Repurchase
  - Refactor
    - Modificare (rearanjare) cod sursa fara a altera comportamentul
    - Se reproiecteaza aplicatia pentru a putea fi implementata nativ in nor
  - Retire
  - Retain

Curs 8

Docker

- Containere Linux (LXC) executate simultan, izolate

- Incarcare minima
- Foarte utilizat
- LXD = generatia urmatoare de containere Linux (dupa LXC)
- Se utilizeaza imagini precompilate de Linux + REST API
- Fiecare container e executat intr-un singur proces
- Se creeaza stive de File Systems
- Scade granularitatea proiectarii aplicatiilor
- Containerele incapsuleaza servicii => microservicii
- runc e utilizat pentru executarea fiecarui container
- Foloseste Linux namespaces:
    - PID (Process ID)
    - NET (Networking)
    - IPC (Inter-Process Communication)
    - MNT (Mount)
    - UTS (Unix TimeSharing => izolare kernel)
- Se folosesc *cgroups* pentru limitarea resurselor utilizate
- Este asigurata echilibrarea incarcarii

Filozofie Docker

1. Build
    - Creare/distrugere de imagini
2. Ship
    - Imaginea containerului e trimisa la un server
3. Run
    - Din imagine se creeaza un container iar acesta e mutat in productie

Imagini

- Indexate intr-un catalog:
    - Local
    - Public (al Docker)
- Push/Pull

Barfa in Docker (Docker Gossip)

- Exista retele Docker suprapuse
- Acestea au minim 1 nod comun
- Acesta disemineaza informatii de la o retea la alta (“barfa”)

POD

- Definitii:
  - Performance optimized data center (introdus de catre Sun)
  - Proces care se executa intr-un cluster

Lansare

- Pe baza instructiunilor din manifest
- In el:
  - Imagine Docker
  - Nr. instante
  - Nr. resurse
  - Reguri particulare (ex: nu pune X si Y pe aceeasi gazda)

Docker-compose

- Foloseste fisiere de configurare .YML, mentionandu-se:
  - Serviciile
  - Imaginile
  - Volumele
- Descrie aplicatii multi-container
- Automatizeaza lant DevOps
- Pe un server cu CI/CD se utilizeaza pentru a evita interferente intre builduri
- Reutilizeaza containere deja existente daca MS nu s-a modificat
- Se foloseste pentru:
  - Dezvoltare
  - Testare automata
  - Gazda unica (se poate instala la distanta un container) ???

Cum se creeaza arhitecturi MSA

- Creez MS
- Testez
- Creez imagine
- Integrez in arhitectura

Orchestrarea containerelor

- Proces de instalare automat si optimizare a mai multor containere
- Folosind Docker Swarm
- Cluster => mai multe gazde => roi

- Gazdele pot fi:
  - Worker
  - Manager
  - De ambele (nu se recomanda, daca moare se pierde tot + incarcare mare)
- Se definesc numarul de replici, resurse, proturi expuse
- Exista retele de intercomunicatii:
  - Managerul asociaza adrese continue
  - Gestionarul roi asociaza un DNS fiecarui server
  - Se expun porturi pentru echilibrarea incarcarii ???
- Nod = instanta a masinii Docker care se executa in cluster
- Filtre:
  - Constrangere sau marcaj nod
  - Afinitate
    - Se specifica daca un containere se poate executa in aceeasi retea cu altul
  - Portul (resursa unica)
  - Dependenta
    - Daca exista dependente intre MS, atunci se executa in acelasi nod
  - Sanatatea
    - Ex: ca la HeartBeatMonitor din laboratoare
    - Verifica doar ca un MS “mai traieste, nu si cat de bine o duce”
- Alte solutii de orchestrare:
  - Kubernetes
  - Apache Mesos

Docker PRO

- Configurare simpla ??? (doar 40 pagini de PDF…)
- Incarcare minima
- Nu exista dependente de locul unde se executa imaginea
- Codul scris de dezvoltatori se va potrivi la instalare
- Productivitate: fiecare MS intr-o masina virtuala
- Izolare
- Grupare servere pentru reducere costuri
- Depanare ??? (nu prea cred…)
- Dezvoltare rapida

Docker Contra

- Nu doar pentru crestere performanta (utilizeaza doar cate resurse primeste de la Kernel)
- Nu e ok in cazul unor constrangeri serioase de securitate (probleme cu File Systems)
- Nu utilizati Docker daca doriti GUI ???
- Nu pentru mediu de dezvoltare simplu, facil si depanare pas cu pas
- Nu pentru alt SO/Kernel decat cel de baza
- Nu pentru salvat multe date importante
    - Deobicei moare serviciul care face procesari
    - Cele de persistenta nu ar trebui sa aiba probleme => Slide in bataie de joc…
- Greu de controlat

Curs 9

Modele de proiectare MSA

Sugrumatorul (*Strangler*)

- Realizeaza tranzitia de la un monolit la MSA
- Se tot extrag servicii din monolit
- Apare cate un proxy care decide care cereri se trimit la monolitul original si care se trimit la noile MS

Compartimentat (*Bulkhead*)

- Ofera toleranta la erori
- Retine conexiuni organizate in pool-uri
- Daca o conexiune pica, este inlocuita cu alta din pool

API

- Ajuta la apelarea in mod simplificat a serviciilor
- Daca exista canale de comunicatii diferite => probleme
- Cereri/raspunsuri in formate diferite
- Cand se doreste utilizarea unor protocoale nesuportate de servicii
- Securitate slaba

Agregator

- Apare un gateway care decide cui inainteaza cererea
- Poate interpreta si modifica raspunsul

MS inlantuite

- Produce un singur raspuns
- Cam ca la Chain of responsability

Ramificatie

- O combinatie intre lant de MS + API/agregator

Compunere interfata utilizator pentru client

- Pentru aplicatii mobil/web
- Fiecare element e manageriat de un MS
- Daca 1 MS pica => restul contiuna sa functioneze

Baza de date pentru un MS

- 1 DB/MS
- Redundanta mare
- Mai multe MS pot dori aceeasi BD => probleme
- BD trebuie replicat in maniera Master-Slave (trebuie asigurata replicarea)

Baza de date comuna

- Nerecomandata
- Se pot folosi cache-uri, cozi de mesaje pentru cresterea de viteza

Impartirea responsabilitatilor in crearea unei interogari

- ?????????

Saga

- Gestioneaza consistenta datelor intre microservicii in scenarii de tranzactii distribuite
- Reprezinta o secventa de tranzactii care actualizeaza fiecare serviciu si publica un mesaj sau eveniment pentru a declansa urmatorul pas de tranzactie

- Cand 1 tranzactie esueaza, tranzactiilor realizate anterior li se da undo (*rollback*)

Compunerea jurnalelor
- Se combina mai multe jurnale
- Ex: serviciu de colectare de date

Metrici de performanta
- ??????????????

Urmarire distribuita
- Se proiecteaza un strat de instrumentatie pentru urmarire
- Se atribuie UUID (ID-uri unice) pentru fiecare MS

Verificarea sanatatii
- Verifica daca inca mai functioneaza un MS
- Nu se stie cu exactitate daca functioneaza corect (ex: poate inca ruleaza dar e blocat si nu mai trateaza cereri => nu i se mai inainteaza cereri si e restaurat)

Configuratie externalizata
- Se externalizeaza credentials, endpoints etc
- Se incarca la pornire de la un alt MS

Descoperirea MS
- Exista un MS (ex: Service Registry) care arata ce alte MS sunt disponibile

Proiectare cu siguranta (intrerupator de circuit)
- Erorile se pot propaga prin sistem
- Acest model poate opri un sistem din a trata o cerere daca observa ca aceasta ar duce la picarea inlantuinta a mai multor MS

Verde-albastru
- Modeleaza ciclul de viata al unei aplicatii
- 2 copii ale aplicatiei pe HW identic
    - 1 in productie
    - 1 in dezvoltare (se modifica serviciile)

- Dupa ce se modifica si se testeaza, aceasta din urma ajunge in productie

Calcul evolutiv
- Pas spre obtinerea HW reconfigurabil automat/manual
- Contine algoritmi inspirati din evolutia organismelor biologice
- Suporta dezvoltare incrementala pe diferite dimensiuni

Fitness
- Functii care sumarizeaza cat de aproape e proiectul de teluri
- Stratul de instrumentatie ajuta
- Maxim 80 de dimensiuni :))
- Se aleg cele mai importante
- Se urmareste echilibrarea lor

Utilizarea pipe-urilor de instalare pentru automatizarea functiilor de fitness
- Definirea etapelor
- Alegerea functiilor de fitness
- Alegerea locului de aplicare a acestora
- Ciclul de modificare a aplicatiei
- Viteza acestui ciclu

Necunoasterea problemelor necunoscute = nu se pot cunoaste toate problemele din start (ce mama naibii???)

Curs 10

Topics ~= cozi de mesaje
Evenimente complexe
- Mai multe atribute
- Ca la Kafka
- Atomic, imutabil
- Inrudire (cu alte evenimente din acelasi grup)
- Cea mai importanta parte in proiectare si utilizare e comportamentul

Abordari
- Evenimentul primul

        - Procesare la server
        - Presupune schimbarea completa a gandirii de la REST
        - PRO
            - Decuplare
            - Incapsulare
            - Inversarea responsabilitatii
            - Flexibilitate sporita
        - Contra
            - Gestiunea pierderii controlului e mai complicata
- Evenimentul comanda
    - Procesare la client
    - REST clasic
    - Nedeterminare mica

Kafka

- Platforma de comunicare, urmarire, procesari big data
- Pleaca de la RabbitMq
- RabbitMq tine mesajele in memorie => de multe ori acestea sunt duse in memoria swap (memorie RAM virtuala, care e de fapt pe HDD/SSD)
- Util in aplicatiile cu procesare de fluxuri
- Scalabilitate mare (utilizat de Twitter, Spotify etc)
- Mecanism Publish-Subscribe
- Mesajele sunt puse in topics

Arhitecturi orientate pe evenimente

- Se mai numesc “Message Driver Architectures” sau “Stream Driver Architectures”
- Comunicatii broadcast
- Reactie eficienta
- Evenimente cu granularitate mica