GROUP MJ

OLA 2

Contents

Q1 2

Hvad er de primære forskelle mellem Enterprise og Solution Architecture 2

De forskellige roller i designsystem 2

Q2 3

Stefan Tilkovs Syn på Teamets Rolle i Moderne Arkitektur 3

Hvilke Team Topologier Beskriver Bogen *Team Topologies*? 3

Nøglen til Et Succesfuldt Projekt? 3

Q3 4

Hvorfor og Hvilke Fordele Fremhæver Han? 4

Q4 4

Er Hans Argumenter Overbevisende, og Hvorfor? 4

ABCDE 4

Q5 5

På hvilke måder gavner det Saxo Bank ifølge Simon Rohrer 5

Hvordan forbinder han continuous conversation med DevOps infinity loop? 5

Q6 5

5 Messaging-Integrationsmønstre og deres use case 5

Q7 6

Messaging og conversation arkitektur – hvad er forskelle 6

Udfordringer ved conversation-baserede løsninger 6

Pub-Sub vs. Subscribe-Notify: 7

Q8 7

Q9 8

Hvorfor ses dette som et komplekst problem, og hvordan hjælper strangler-mønsteret? 8

Q10 8

Hvad Er De, og Hvilken Rolle Spiller De? 8

Q1

Hvad er de primære forskelle mellem Enterprise og Solution Architecture

Den primære forskel mellem Enterprise Architecture (EA) og Solution Architecture (SA) er, at SA implementerer de løsninger, som EA har fastlagt. SA fokuserer på specifikke projekter eller løsninger og har ansvaret for at designe de tekniske aspekter af en konkret løsning, som skal implementeres, ofte inden for de rammer, EA har defineret.

De forskellige roller i designsystem

- Forretningsarkitekt
  Deres primære opgave er at tilpasse forretningsmål med IT-strategier samt at forstå og dokumentere forretningsprocesser og mål. De skal også sikre, at teknologien understøtter forretningsmålene og bidrager til den overordnede forretningsstrategi. Med andre ord skal de fungere som en bro mellem forretningsdriften og IT-infrastrukturen.

- Informationsarkitekt (eller Dataarkitekt)
  I informationsarkitektur er fokus på håndtering og styring af virksomhedens data. Opgaven er at definere, hvordan data skal opbevares, administreres, integreres og anvendes i organisationen. De skal sikre, at data er struktureret på en måde, der understøtter forretningsmålene og giver indsigt til beslutningstagning, ofte med særligt fokus på metadatahåndtering og datastyring.

- Applikationsarkitekt
  Deres hovedopgave er at designe og administrere softwareapplikationer og systemer. Det er vigtigt, at de integrerede applikationer er i overensstemmelse med forretningsbehovene. De skal også sikre, at systemerne er skalerbare, sikre og nemme at vedligeholde

- Teknologiarkitekt (eller Infrastrukturarkitekt)

Teknologiarkitekten er ansvarlig for at definere den tekniske infrastruktur, såsom servere, netværk og lagring, og sikre, at den understøtter organisationens applikationer og tjenester. Fokus er på hardware, cloud-infrastruktur og tekniske kapaciteter med henblik på at sikre pålidelighed, skalerbarhed og optimal ydeevne

Q2

Stefan Tilkovs Syn på Teamets Rolle i Moderne Arkitektur

Stefan Tilkov understreger, at teams spiller en central rolle i moderne arkitektur, og at det ikke kun handler om teknologiske valg, men i lige så høj grad om organisatoriske beslutninger. Han betoner, at der er en tæt sammenhæng mellem softwarearkitektur og måden, hvorpå teams arbejder og interagerer. Ifølge ham bør arkitektur og teams være spejlbilleder af hinanden. Han argumenterer for, at teams skal være små, selvstændige og have klare afgrænsninger, så de er i stand til at træffe uafhængige beslutninger.

Tilkov påpeger også vigtigheden af at finde den rette balance mellem centralisering og decentralisering i arkitekturen. Moderne arkitekturer skal være fleksible og kunne tilpasse sig forandringer, da de ofte er distribuerede og komplekse. Derfor er det afgørende, at både arkitektur og teams designes til effektivt at håndtere denne kompleksitet.

Hvilke Team Topologier Beskriver Bogen *Team Topologies*?

Bogen *Team Topologies* beskriver en ramme for, hvordan softwareteams kan organiseres effektivt for at opnå hurtigere leverancer og forbedret samarbejde. Den introducerer fire centrale teamtyper og fokuserer på, hvordan disse teams bør interagere for at håndtere kompleksitet og sikre et optimalt arbejdes flow.

De fire teamtyper, som bogen præsenterer, er:

1. Stream-aligned teams: Teams, der er dedikeret til et specifikt produkt eller en service og har til formål at levere værdi direkte til kunderne.
2. Platform teams: Disse teams administrerer den underliggende infrastruktur og tjenester for at reducere den kognitive belastning for stream-aligned teams, så de kan fokusere på deres kerneopgaver.
3. Enabling teams: Teams, der hjælper andre teams med at forbedre deres færdigheder, f.eks. ved at introducere nye værktøjer, teknologier eller metoder.
4. Complicated-subsystem teams: Ansvarlige for områder af systemet, der er særligt komplekse og kræver specialiseret ekspertise.

Nøglen til Et Succesfuldt Projekt?

Et succesfuldt projekt kræver ofte forskellige kompetencer, som ingen enkeltperson kan mestre fuldt ud. At samle forskellige færdigheder ét sted gør det muligt at løse problemer på en mere kreativ og helhedsorienteret måde. Et hold der arbejder godt sammen, skaber et miljø, hvor idéer kan deles, udfordres og forbedres. Dette fører ofte til innovative løsninger, som ikke nødvendigvis ville opstå hos en enkeltperson.

I komplekse projekter gør opdelingen af opgaver efter hold medlemmers styrker arbejdet mere effektivt og sikrer, at alle aspekter af projektet bliver behandlet grundigt.

Desuden giver teams også følelsesmæssig støtte, som er afgørende for at holde motivationen oppe, når projektet bliver udfordrende.

Ud fra min erfaring opnår projekter med velstrukturerede og samarbejdende teams generelt bedre resultater end dem, hvor folk arbejder isolere

Q3

Hvorfor og Hvilke Fordele Fremhæver Han?

Tilkov mener, at der bør være en balance mellem centralisering og decentralisering. Mens decentralisering giver teams frihed til at handle selvstændigt, kan centralisering af visse kerneydelser eller beslutninger være med til at sikre systemets overordnede sundhed og skalerbarhed samt forhindre fragmentering og dobbeltarbejde. Centralisering er ofte nødvendig for områder som sikkerhed, compliance og governance, mens decentralisering fremmer udviklingshastighed og innovation. Ved at finde den rette balance kan man sikre både effektivitet og fleksibilitet i systemet. Det handler derfor om at afgøre, hvilke dele af systemet der bedst håndteres centralt for at sikre ensartethed og effektivitet, og hvilke dele der bør decentraliseres for at fremme innovation og tilpasning.

Q4

Er Hans Argumenter Overbevisende, og Hvorfor?

Simon Rohrer gør sine argumenter overbevisende ved at vise, hvordan moderne arkitektur effektivt tackler de problemer, som legacy-systemer står over for især ved hans ABCDE. Hans fokus på tilpasningsevne, forretningsmæssig relevans, samarbejde, distribuerede systemer og integration i et bredere økosystem giver god mening i forhold til de krav, virksomheder møder i dag. Det er netop denne praktiske tilgang, der gør hans pointer, er så stærke og overbevisende for os.

ABCDE

Tilpasningsevne (A): Moderne arkitektur giver organisationer mulighed for hurtigt at tilpasse sig forandringer, mens ældre systemer ofte er stive og svære at ændre. Dette er en stor fordel i dagens teknologimiljø.

Forretningsmæssig tilpasning (B): Rohrer pointerer, at moderne arkitektur er tættere knyttet til forretningsmål, hvilket gør det muligt at skabe mere værdi, i modsætning til legacy-systemer, der ofte mangler denne fokus.

Samarbejde (C): Samarbejde på tværs af teams er centralt i moderne arkitektur, hvorimod ældre systemer ofte fører til siloer. Rohrers fokus på teamwork er i tråd med agile og DevOps-praksis, hvor samarbejde er nøglen til succes.

Distribuerede systemer (D): Moderne arkitektur udnytter distribuerede systemer, som er mere skalerbare og fleksible sammenlignet med monolitiske legacy-systemer. Dette passer godt til cloud-native strategier.

Økosystem (E): Moderne arkitektur er bygget til at arbejde i et bredere økosystem af tjenester og værktøjer, hvilket gør integration med eksterne API'er til en nødvendighed for at forblive konkurrencedygtig.

Q5

På hvilke måder gavner det Saxo Bank ifølge Simon Rohrer

I videoen forklarer Simon Rohrer "Continuous Conversation" som en løbende dialog mellem udviklings- og driftsteams, hvilket hjælper Saxo Bank med at holde deres teknologiske løsninger og forretningsmål i sync. Det gør, at de kan reagere hurtigere på ændringer og løse problemer, inden de vokser sig større. På den måde fungerer samarbejdet bedre, og de kan levere værdi til kunderne mere effektivt.

Hvordan forbinder han continuous conversation med DevOps infinity loop?

Han forbinder det med DevOps Infinity Loop ved at forklare, hvordan samtalen fortsætter gennem hele udviklingsprocessen – fra planlægning til drift og tilbage igen. Det skaber et konstant feedback-loop, hvor udviklings- og driftsteams lærer af hinanden, hvilket gør systemet mere fleksibelt og i stand til at levere bedre resultater.

Q6

5 Messaging-Integrationsmønstre og deres use case

Beskedkanal (Message Channel):

Mønster: En beskedkanal fungerer som en forbindelse, der lader beskeder blive sendt fra en afsender til en modtager. Det fjerner behovet for, at afsenderen og modtageren kender hinandens præcise placering. Eksempel: I et distribueret system kan applikation A sende data til applikation B, uden at skulle vide, hvor B er. Beskedkanalen sørger for, at A kan sende beskeden, som B henter, når det passer.

Beskedrouter (Message Router):

Mønster: En beskedrouter tillader beskeder at blive dirigeret til forskellige destinationer uden at afsenderen og modtageren behøver at blive enige om en fælles kanal. Eksempel: I et transportsystem kan beskeder om lageropdateringer og leveringsstatus automatisk sendes til forskellige systemer (f.eks. transportfirmaer eller lagersystemer) uden, at systemerne skal være direkte forbundet.

Beskedmægler (Message Broker), f.eks. RabbitMQ:

Mønster: En beskedmægler modtager beskeder fra udgivere og videresender dem til modtagere. Det kan ske via enten punkt-til-punkt-kommunikation eller en publish-subscribe model. Eksempel: I en onlinebutik kan RabbitMQ bruges til at modtage kundeordrer og sende dem videre til forskellige lager- og leveringssystemer baseret på lagerstatus og placering.

Publish-Subscribe Kanal:

Mønster: En publish-subscribe kanal distribuerer beskeder til flere modtagere, som har valgt at abonnere på disse beskeder. Hver modtager modtager en kopi af beskeden. Eksempel: I et aktiehandelssystem kan realtidsopdateringer om aktiekurser sendes ud via en publish-subscribe kanal, hvor flere handelsapps og porteføljeforvaltere modtager opdateringerne samtidigt.

Kø (Queue):

Mønster: En kø gør det muligt at gemme beskeder, indtil en modtager er klar til at behandle dem. Dette understøtter asynkron kommunikation, så afsender og modtager ikke behøver at være tilgængelige på samme tid. Eksempel: I et ordrehåndteringssystem kan indgående ordrer gemmes i en kø, hvis ordresystemet midlertidigt er nede. Når systemet er oppe igen, behandles ordrerne i den rækkefølge, de kom ind.

Q7

Messaging og conversation arkitektur – hvad er forskelle

Messaging Arkitektur:

I denne type arkitektur kommunikerer systemerne gennem individuelle beskeder, der typisk udveksles mellem to parter. Det er ofte struktureret som en request-response model, hvor et system sender en forespørgsel og venter på svar. Hovedvægten ligger på levering af beskeder og at sikre, at hver besked bliver modtaget, behandlet og korrekt besvaret. Denne arkitektur er mere synkron, hvilket betyder, at afsenderen forventer et hurtigt svar fra modtageren.

Conversation Arkitektur:

I modsætning til messaging fokuserer en conversation-baseret arkitektur på længerevarende interaktioner, der involverer en sekvens af sammenhængende beskeder mellem flere systemer. Disse samtaler er ofte asynkrone, hvilket betyder, at beskederne udveksles over tid i flere trin, hvor hver besked kan udløse yderligere handlinger eller svar. Hovedmålet er at koordinere kommunikationen mellem flere deltagere, bevare konteksten undervejs og håndtere tilstanden mellem beskeder for at sikre, at samtalen fortsætter flydende.

Udfordringer ved conversation-baserede løsninger

State Management: Fordi samtaler ofte varer længe og involverer flere beskeder, kan det være svært at holde styr på tilstanden på tværs af systemer. I modsætning til en simpel messaging-tilgang, hvor tilstanden er midlertidig, kræver samtaler, at man konstant opretholder konteksten.

Coordination and Consistency: Når beskeder udveksles mellem flere systemer over tid, kan det være en udfordring at sikre, at alle systemer arbejder sammen og forbliver i sync. Det bliver særligt vanskeligt, hvis nogle systemer fejler, genstarter eller er langsomme til at svare under samtalen.

Pub-Sub vs. Subscribe-Notify:

- **Pub-Sub**: I dette mønster sender udgivere beskeder til et emne eller en kanal, og abonnenter modtager dem uden at være direkte forbundet til udgiveren. Det fungerer mere som en broadcast, hvor beskeder sendes ud uden forventning om specifikke svar.
- **Subscribe-Notify**: Her registrerer abonnenter deres interesse for bestemte begivenheder, og systemet giver dem besked, når disse begivenheder finder sted. Fokus er mere på hændelsesbaserede interaktioner, som ofte bliver en del af længere samtaler fremfor enkeltstående broadcasts.

Q8

Centrale krav for nyttige patterens og vores Opsummering

I Gregor Hohpes video *Enterprise Integration Patterns 2* beskriver han begrebet mønstre og mønstersprog, og hvordan de har rødder i Christopher Alexanders arbejde med arkitektur, men også hvordan de nu anvendes i softwareudvikling. Han fremhæver, at mønstre fungerer som en metode til at beskrive gentagne løsninger på velkendte problemer i en specifik kontekst, hvilket skaber et fælles sprog, der kan bruges på tværs af forskellige situationer og systemer.

For at mønstre skal være effektive, er der nogle centrale krav, de skal opfylde:

- **Genanvendelighed**: Mønstre skal beskrive løsninger, der er brede nok til at kunne bruges på tværs af forskellige systemer og situationer, men stadig specifikke nok til at være praktisk anvendelige.
- **Kontekstafhængighed**: Et mønster giver kun mening, hvis det er knyttet til en bestemt kontekst. Konteksten afgør, hvornår og hvor det er relevant at bruge mønstret, så man undgår at anvende det forkert i situationer, hvor det ikke passer.
- **Fokus på problem og løsning**: Mønstre skal klart beskrive det problem, de adresserer, og den løsning, de tilbyder. Det gør det lettere for andre at forstå, hvornår mønstret kan bruges, og hvordan det løser problemet.
- **Fleksibilitet og struktur**: Et mønster skal have tilstrækkelig struktur til at kunne guide implementeringer, men samtidig være fleksibelt nok til at kunne tilpasses forskellige systemdesigns og begrænsninger.
- **Letforståeligt**: Mønstre skal formuleres klart og præcist, så både nye og erfarne brugere kan forstå dem og anvende dem i praksis.

Q9

Hvorfor ses dette som et komplekst problem, og hvordan hjælper strangler-mønsteret?

I artiklen forklarer Simon Rohrer, at det kan være svært at slippe af med gamle systemer, fordi de ofte er tæt integreret i virksomhedens daglige drift og har mange afhængigheder. De har været i brug i mange år og spiller en vigtig rolle, så det kan være risikabelt at ændre dem. Hvis noget går galt, kan det føre til nedetid og andre problemer for virksomheden.

Strangler-pattern hjælper ved at lade os erstatte dele af det gamle system gradvist i stedet for at skifte det hele ud på én gang. Det betyder, at vi kan bygge nye funktioner ved siden af det gamle system og langsomt fjerne de gamle dele, når det nye system er klar til at overtage. Det gør processen mere sikker og minimerer risikoen for større fejl eller forstyrrelser undervejs.

Q10

Hvad Er De, og Hvilken Rolle Spiller De?

I Jesper Lowgrens video *"Solution vs Enterprise Architecture Tutorial"* beskrives tre vigtige diagrammer, som kan bruges til at forklare en arkitektur. Diagrammerne hjælper med at vise forskellige synspunkter og give en bedre forståelse af, hvordan arkitekturen fungerer. Her er der eksempel på tre diagrammer:

1. High-Level Overview (Conceptual View)

Dette diagram giver et overblik over hele systemet og dets omgivelser. Det viser, hvordan systemet taler sammen med brugere, andre systemer eller organisationer uden for systemet.

- **Eksempel:** Tænk på et online banksystem. diagrammet ville vise bankappen og hvordan den snakker med fx betalingsgateways, andre banker og brugerne, der logger ind via en mobil- eller web app.

Dette diagram er godt til at vise, hvem systemet har kontakt med, og hvad der sker omkring det.

2. Technical Components (Functional or Logical View)

Dette diagram går mere i detaljer og viser, hvordan systemet er bygget op indvendigt. Det viser, hvilke mindre dele eller moduler der findes, og hvordan de hænger sammen.

- **Eksempel:** I onlinebanksystemet kunne dette diagram vise, at der er en del til at logge brugere ind, en til at behandle transaktioner og en til at styre kontooplysninger. Diagrammet viser, hvordan de enkelte dele arbejder sammen.

Dette diagram er nyttigt for dem, der arbejder på at bygge eller forstå systemets indre opbygning fx arkitekter, udviklere og IT-ledere, der har brug for at forstå, hvordan løsningen hænger sammen på et funktionelt niveau.

3. Implementation and Infrastructure (Physical View)

Dette diagram viser, hvordan systemet er fordelt fysisk, fx på forskellige servere eller i skyen. Det viser, hvor systemets dele ligger, og hvordan de kommunikerer.

- **Eksempel:** For onlinebanksystemet kunne det vise, at nogle dele kører på en web-server, andre på en database, og at der er kommunikation imellem dem.