Afgangsprojekt for Informationsteknologi
Jakob Viggo Hansen

1.1 Problemformulering

I store virksomheder hvis virke er baseret på adskillige samarbejdende IT-systemer, er overvågning af drift og stabilitet en udfordring. Mange systemer kan udsende alarmer i tilfælde af driftsforstyrrelser eller stop. Men en enkeltstående alarm-melding vil ofte ikke afsløre det samlede billede af det påvirkede system. Det er vanskeligt at udlede kontekst og underliggende drift status.

Desuden er diskrete systemer næsten altid blot komponenter i en længere kæde, et workflow eller data-flow. En fejl i et enkelt system skyldes ofte fejl i systemer som er placeret tidligere i kæden. Eller systemer senere i kæden kan lide under det aktuelle systems fejl.

Alle disse systemer genererer drifts-logs - ofte adskillige tusind linjer pr time. Logs er indgangen til at udlede et billede af systemets tilstand, aktuelt og bagud i tiden.

Men det kan være vanskeligt og tidskrævende at gennemlæse mange tusind loglinjer, og vanskeligt at sammenstille de informationer som fremsøges, på tværs af tid og systemer. Overblikket kan drukne i mængden af data.

Jeg foreslår at udvikle en hjælpe-applikation, LogWatcher. **LogWatcher er målrettet til personer som skal supportere og drifte virksomhedens system-park.**

1.2 Indledning

Til daglig arbejder jeg på DR, hvor jeg udvikler applikationer og workflows som understøtter de indholdsskabende afdelingers arbejde. Der har jeg haft lejlighed til at indse at en applikations nytteværdi i høj grad er afhængig af hvordan den driftes. Når der før eller siden opstår et problem i eller omkring den, er muligheden for spore fejlkæder afgørende for driftspersonalet.

LogWatcher skal kunne hjælpe brugeren med at søge efter tekst-billeder på tværs af mange systemers logs. Baseret på søgeord og tidsgrænser skal LogWatcher hjælpe sin bruger med at danne sig et overblik over hændelser og deres tidslige distribution.

LogWatcher skal have en **grafisk brugerflade** baseret på en HTML-side, tilgængelig via en standard web-browser. GUI'en skal facilitere **indsamling** af logs fra forbundne systemer, **gennemsøgning** af logs på basis af søge-termer og tidshorisonter, og **udstille** resultaterne fra søgningerne.

Forud for selve søgningen skal de indsamlede logs persisteres til en **database**, som derefter vil udgøre det data-grundlag som LogWatcher skal traversere.

Søge-algoritmerne skal være baseret på trie's og binære søgetræer, for at sikre god performance når brugeren potentielt skal søge over mange tusinde linjer rå log-data. Søge-komponenten bør udstille et ***eDSL** til at facilitere søgning i domænet.

Søge-resultater kan udstilles som tekst-udsnit fra søge-hits. F.eks

- top-5 af systemer med søgeordene repræsenteret i deres logs.
- et grafisk billede af den tidslige distribution af et sæt søgeord (f.eks. over de seneste 48 timer)

Ved hjælp fra LogWatcher kan brugerne altså hurtigere danne sig et overblik over begivenheder på tværs af systemer, og få hjælp til at danne en forståelse af hvor de underliggende problemer kan være opstået i den lange kæde af indbyrdes forbundne systemer.

**embedded Domain Specific Language*

2.1 Teknologivalg - mulige retninger

Fordelt over de programmerings-specifikke moduler under uddannelsen, har jeg arbejdet med først Javascript, derefter C#/.NET og senest Python. De tre sprog ligger pr. 2022 blandt Top 7 Mest populære/anvendte sprog. (* https://spectrum.ieee.org/top-programming-languages-2022)

Javascript repræsenterer det type-svage sprogdomæne, også kaldet dynamisk typing. Det er et fortolket sprog, dvs. det omsættes til maskinvendt kode linje-for-linje under kørsel. En deklareret variabel kan ændre indholdskaraktér i løbet af sin levetid, dvs. igennem programmets køretid. Variablen A kan starte med at refererer heltals-værdier, dernæst en tekst-streng, og senere et komplekst objekt. Javascript er i særdeleshed udbredt fordi det er de facto normen til at kode interaktivitet på web, sammen med html/css. Javascript har traditionalt været påskønnet til webudvikling pga. sit kompakte og fortættede syntax.

C# repræsenterer det "modsatte" domæne, det type-stærke, kaldet static typing. Et C#-program bliver kompileret til maskine-nær kode før eksekvering. I C# kan alle variable udelukkende referere den datatype de deklareres mod. C# er stærkt object-orienteret. Det blev udviklet som Microsofts

svar på det oprindelige objektorienterede Java fra Sun/Oracle. (https://en.wikipedia.org/wiki/Java_(programming_language))
C# er knyttet til kørsel- og udviklings miljøet .NET, som er Microsofts bud på et kode øko-system til Windows-platformen. I de senere år er .NET Framework blevet afløst af .NET Core, som også er egnet til udvikling mod OSX (Apple), Linux og Android (Google).

Python kan med god vilje placeres imellem JavaScript og C#. Det kan kodes både i et objektorienteret og et funktionelt paradigme, og er typesvagt. Python tilbyder en meget kompakt kode-stil som gør det særligt populært til at processere store datamængder med få linjer kode.

https://hackr.io/blog/c-sharp-vs-python
https://litslink.com/blog/csharp-vs-python-choosing-right-language-for-your-project
https://developer.ibm.com/blogs/use-python-for-scientific-research/

Hvert enkelt sprog har specifikke styrker og fordele. Python giver mulighed for meget elegante og effektive recursioner som er vigtige for at bygge de node-baserede søge-træer som udgør søgemotoren. C# understøtter multi-threading indenfor samme app-instans, i modsætning til Python.

2.2 .NET i DR

På DR anvendes flere forskellige teknologier. Qua sin rolle som Public Service Broadcaster* midt i en digital transition mod web, er en meget stor del af husets seer/brugerrettede kodebase i Javascript og understøttende frameworks som React og Redux. Men de fleste af de internt rettede egenudviklede applikationer er baseret på MS .NET og skrives i C#.

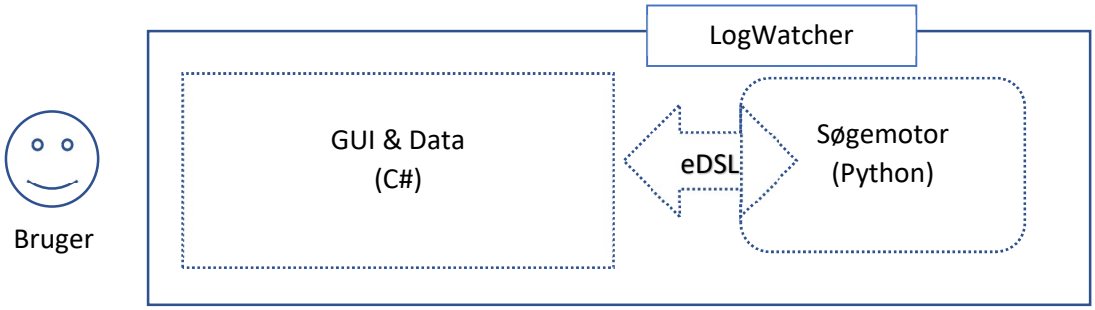
Med få undtagelser, er fundamentale produktions-systemer i huset baseret på off-the-shelf produkter. Men integrationslagene mellem produktion-systemerne er i høj grad egenudviklet. Når data flyttes fra ét system, f.eks. programplanlægningsværktøjet WHATS'ON til programafviklingen (publiceringsplatformen), sker det igennem værktøjer og processer som DR selv udvikler. Den udvikling er baseret på MS .NET og C#.

LogWatcher vil sidde naturligt som et støtteværktøj for det personale som varetager driften af integrationslagene. Et godt værktøj skal kunne tilpasses løbende til en dynamisk og foranderlig kontekst, og der må forventes løbende udvikling og tilpasning, også efter idriftsættelse.
Ved at vælge MS .NET som udviklingsmiljø, skabes det bedste potentiale for at forankre værktøjet hos brugerne.

*https://en.wikipedia.org/wiki/Public_broadcasting

2.3 Kerne-teknologi: C# & .NET Core, Blazor og Python

Til udvikling af LogWatcher vil jeg kombinere C# og Python. Jeg vil bruge Python til at facilitere de centrale aspekter af søgning i datasæt, søgekomponenten. Jeg vil bruge C# til grafisk brugerflade (GUI) og data-persistering mod databasen.



Ved at indbygge et eDSL i søgekomponenten, kan jeg udstille de primære søge-værktøjer mod det omsluttende C#/.NET lag som udgør den brugervendte applikation.

Med .NET Core har Microsoft introduceret et web-orienteret lag til C#, som gør miljøet meget egnet til at udvikle web-sider. Den specifikke komponent hedder Blazor, og giver mulighed for en eventdrevet, dobbeltrettet binding mellem grafik-objekter og de underliggende tilstande i applikationen. Denne teknologi vil jeg benytte til LogWatcher GUI.

2.4 Overvejelser om driftsmiljø

En væsentlig overvejelse i forbindelse med valget af teknologi-platform for LogWatcher, må være at undersøge hvilke miljøer den skal eksistere i. En applikation består ikke bare af sin kode, men eksisterer også i kraft af det miljø som understøtter den.
Hvilken afviklingsplatform kører den på - windows, linux, OSX ?
Kan den virtualiseres eller måske køres i en cloud-løsning?
Deployes applikationen via Continuos Integration(CI)?
Hvem håndterer daglig drift og problem-løsning ifald der opstår fejl?

https://www.atlassian.com/continuous-delivery/continuous-integration
https://en.wikipedia.org/wiki/Continuous_integration

2.4.1 VM's vs. Containers

I moderne IT-drift er det ikke længere standard praksis at afvikle applikations-instanser på diskret hardware. I stedet har man en indført op til adskillige lag af abstraktionsniveauer over hardware-niveauet. Det giver mulighed for at afkoble afhængigheden til specifik hardware, og i stedet introducere en generaliseret arkitektur.

Overordnet set er der i dag 2 grene af abstraktion, **Virtual Machine** (VM) og **containers**.
En *VM* er i realiteten en simulering af en komplet PC, dvs. hardware som CPU, RAM, storage osv. bliver simuleret for den applikation som skal køres. Ovenpå hardware-simulationen afvikles værts-styresystemet (OS), typisk en variant af MS Windows. Denne løsning giver mulighed for at afvikle adskillige, isolerede instanser af en VM mod samme fysiske hardware.
En *container* er et isoleret og relativt letvægts miljø som afvikles på en til formålet egnet platform. Denne platform betegnes som kernel, og sidder oven på enten fysisk eller simuleret hardware. En container sidder så at sige oven på værts-OS, uden at have sit eget dedikerede OS. En container-instans vil normalt kræve færre ressourcer fra værts-systemet end en VM. En meget brugt teknologi til containers er Docker.

En stor del af pointen med disse abstraktioner er, udover at optimere brugen af hardware-ressourcer, at opnå isolation mellem instanserne. Dvs. at undgå at problemer som en opstår i én instans kan have effekt på nabo-instanser. Man taler om at VM vs. containers tilbyder forskellige typer af isolation:
Container: Isolation of the process
VM: Isolation of the machine

2.4.2 DR's best-practice (baseret på interviews)

(Snak med Thomas Borup)

2.5 LogWatcher mål-systemer

(Skriv om hvilke systemer som kunne være kilder til logfiler)

3.1 Data-repræsentation – fra kilde til database

Den fælles kilde-type for alle målsystemer vil være de ustrukturede rå-data i form af log-filerne. Dvs. LogWatcher har ikke adgang til metadata om log-filens struktur og format.

For at sikre en homogen og ensartet kilde til data for søgemoteren, vil jeg vælge at persistere data fra alle målsystemer til en fælles database.

Målsystemerne anvender forskellige formateringer for logningen. Event-tid (TimeOfEvent) kan f.eks være angivet i UTC eller DK-tid. Niveauet for log-linjens alvorlighed (Severity) kan være angivet umiddelbart efter event-tid, eller slet ikke. Event-beskrivelsen kan indeholde tekst. Der kan evt. optræde en reference til det kode-modul som er kilden til log-linjen.

Her er 3 eksempler fra kurante DR-systemer:

Galaxy Site Selector	*2022-11-03 13:05:34.3721081 [INF] @GetDaletServiceIsInstalled, looking for service: "DaletService" (SiteSelector.Domain.Session.SharedAgents.CheckGalaxyInstalled)* *2022-11-03 13:05:34.3738653 [WRN] Couldn't find service: "DaletService" - "Service 'DaletService' was not found on computer '.'." (SiteSelector.Domain.Services.ServiceHandler)*
	TimeOfEvent er angivet med to komponenter, dato og tidspunkt, adskilt af mellemrum. Tidszone er ukendt. **Severity** er angivet i næste kolonne, adskilt af mellemrum og omgivet af klammer []. **Event-beskrivelsen** er i næste kolonne, adskilt af mellemrum. **Kildemodul** er sidste kolonne, adskilt af mellemrum og omgivet af parenteser ().
MA Ingest	*2022-10-04 00:14:29,010 INFO Common.Comms.MaPersist.MaCom.MaRequest [0] - <?xml version="1.0" encoding="UTF-8"?>* *<fault><reason>Error 401: HTTP 401 Unauthorized</reason><detail></detail></fault>* *2022-10-04 00:14:29,010 ERROR Common.Comms.MaPersist.MaCom.MaRequest [0] - MaCom, Un-authorized* *2022-10-04 00:14:29,011 INFO MaSync.ProcessTasks.Domain.ProcessTask.ProcessNew [0] - Failed to authorize with user superadmin* *2022-10-04 00:14:29,011 INFO Common.DbOps.FailedTasksDbOps.StoreFailedTaks [0] - Storing 0898b419-7cfb-4f39-b044-dba2edda9c94 in failed Tasks.*
	TimeOfEvent er angivet med to komponenter, dato og tidspunkt, adskilt af mellemrum. Tidszone er ukendt. **Severity** er angivet i næste kolonne, adskilt af mellemrum. Kilde-modul er næste kolonne, adskilt af mellemrum. **Event-beskrivelsen** er resten af linjen.
VizController	*2022-11-01T19:25:41.207Z superState Handling state: resultat.sceneOutDone* *2022-11-01T19:25:41.209Z superState Error in checkForBlokke: Error: Failed to match currentResultGuid in active playlist.* *2022-11-01T19:25:41.209Z superState Checking for next result...*
	TimeOfEvent er angivet med to komponenter, dato og tidspunkt, adskilt af mellemrum. Tidszone er UTC* **Kilde-modul** er næste kolonne, adskilt af mellemrum **Event-beskrivelsen** ***er resten af linjen***

** https://en.wikipedia.org/wiki/ISO_8601*

Vores database bør anvende ét TimeOfEvent-format, ét Severity-format, ét format til event-beskrivelse og ét format til data om kilde-modul.

Hvis målsætningen om at databasen skal indeholde homogene data skal mødes, er det nødvendigt at filtrere og re-formatere log-data inden persisteringen. Persisterings-processen skal varetage denne arbejdsgang.

3.2 RDBMS vs. NoSql

Inden vi beslutter hvilken database-teknologi LogWatcher skal benytte, er det passende at diskuterede de grundlæggende, dominerende typer af databaser. På tværs af de mange forskellige databaser-produkter ses der 2 grundlæggende typer: relationel og ikke-relationel.

Relational Database Management Systems (RDBMS) har været kendt siden 1970'erne. De relationelle databaser betegnes også SQL-databaser (Structured Query Language). **SQL** er interaktions-syntaxen for RDBMS. SQL er opstået i samme periode som modellen for relationelle databaser og normalisering, men siden forfinet og udviklet.

Det relationelle aspekt opstår ved at alle data gemmes i tabeller, og hvert data-element tildeles en nøgle. Nøglen anvendes til at udtrykke ét tabel-elements **relation** til et andet. RDBMS/SQL stammer fra en periode hvor storage var en bekostelig ressource, og det var vigtigt at reducere data-redundans mest muligt.

Normalisering sigter efter at minimere **redundans** (data-duplikering) og data-skrøbelighed. Data er skrøbelige, hvis ændringer i databasen kan føre til inkonsistens og lign. anomalier, dvs. ukomplette og "fragmenterede" datasæt. En vel-normaliseret database har meget høj grad af data-integritet, dvs. at den er robust.

Hvor effektivt data-duplikering begrænses, kan beskrives af graden af normalisering i et RDBMS.
1. NF (Normalform): ingen kolonner i tabellen gentager en anden kolonnes værdi
2. NF: Overholder 1NF og indeholder kun kolonner som afhænger af primær nøgle (PK)
3. NF: Overholder 2NF og ingen felter udenfor PK er indbyrdes afhængige

En database som er tilstrækkeligt normaliseret vil være nemmere at vedligeholde, ændre og udbygge. Fuld normalisering kan have en negativ effekt på performance, fordi data-sæt er fordelt over mange tabeller, og et meningsfuldt svar skal sammensættes på tværs af disse. Men opvejes til gengæld af fleksibilitet.

NoSQL begrebet dukkede op i begyndelsen af vores årtusinde. Det betegnes som et ikke-relationelt database-paradigme som ikke er afhængigt af tabeller og deres nøgler. Det betyder at en NoSQL-db potentielt kan være bedre egnet til data-sæt som tekst-dokumenter, graph-data o.l.

NoSQL er mere dynamisk orienteret end SQL, og er mere modstandsdygtigt overfor ændringer og tilpasninger, fordi det er knap så betinget af rigide relationer. En NoSQL-baseret database tåler gerne løbende strukturelle modifikationer, i takt med at kravene fra aftager-applikationen ændres. De kan også være meget hurtige, fordi svar ikke skal sammensættes på tværs af mange tabeller.

NoSQL databaser kan under-grupperes i 4 strukturer:

- Kolonne-orienterede: data grupperes i kolonner, hvor antallet af rækker kan varies frit
- Key/Value: HashMap-baseret lookup
- Dokument-orienterede: egnet til JSON, XML ol.
- Graph-orienterede: repræsenterer data i node-baserede strukturer.

I takt med at performance-kravene vokser, kendetegnes de to typer ved forskellig skalerings topografi. Man kan tale om at RDBMS/SQL skal skaleres vertikalt, dvs. at hvis der kræves mere performance, skal der flyttes til en kraftigere server. NoSQL er egnet til horisontal skalering, dvs. over flere diskrete instanser/servere.

3.3 Overvejelser om database-performance og normalisering

Der er to distinkte faser af database-interaktion. Først skal log-data pre-processes og indlæses i databasen. Derefter skal der læses fra databasen ind i søgemoteren.