12. DECEMBER 2022

AU I IT - AFGANGSPROJEKT

UDVIKLING AF LOGSEARCHER

JAKOB VIGGO HANSEN
DANMARKS RADIO
Copenhagen Business Academy

INDHOLD

1 INDLEDNING

Jeg arbejder på DR, hvor jeg udvikler applikationer og workflows som understøtter de indholdsskabende afdelinger. Der har jeg haft lejlighed til at erfare at et systems nytteværdi i høj grad er afhængig af hvordan det driftes og supporteres. Når der før eller siden opstår et problem, er muligheden for at spore fejlkæder afgørende for at driftspersonalet hurtigt kan udbedre fejlen.

1.1 Problemstilling

Virksomheder hvis virke er baseret på adskillige samarbejdende IT-systemer, har en stor udfordring med overvågning af drift og stabilitet. Mange systemer kan udsende alarmer i tilfælde af driftsforstyrrelser eller stop. Men enkeltstående alarm-meldinger vil sjældent vise det samlede billede af det påvirkede system. Det er ofte 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 ét system skyldes ofte fejl i systemer placeret tidligere i kæden. Systemer senere i kæden kan lide under det aktuelle systems fejl.

Systemerne genererer drifts-logs, ét sæt pr. applikation, ofte adskillige tusind linjer pr. time. Disse logs er nøglen til at udlede et retvisende billede af systemets tilstand, aktuelt og bagud i tiden.

Men det kan være uoverskueligt 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.

1.2 Problemformulering

Jeg foreslår at udvikle en applikation, **LogSearcher**, som skal facilitere et hurtigt overblik over flere systemers tilstande, ved at gennemsøge og analysere deres logs.

LogSearchers brugere er personer som skal supportere og drifte virksomhedens system-park. LogSearcher skal hjælpe dem med at søge efter tekst-billeder på tværs af mange systemers logs. Baseret på søgeord og tidsgrænser skal LogSearcher gøre det nemt at danne sig et overblik over hændelser og deres tidslige distribution.

1.3 LogSearchers bestanddele

1.3.1 Grafisk Brugerflade

LogSearcher skal have en grafisk brugerflade (GUI). 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.

1.3.2 Søgemotor

Søge-algoritmerne skal sikre god performance når brugeren skal søge over mange tusinde linjer rå log-data. Søge-komponenten bør udstille et API til at facilitere søgning i domænet.

Søge-resultater kan vises som tekst-udsnit fra søge-hits.

Søgeresultaterne skal give brugerne et hurtigt overblik over begivenheder på tværs af systemer, og hjælpe til at danne en forståelse af hvor de underliggende problemer kan være opstået i den lange kæde af indbyrdes forbundne systemer.

1.3.3 Database

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

2 SYSTEM ANALYSE

Systemudviklings-fasen er primært orienteret mod det brugervendte lag af LogSearcher.

Jeg vil til denne facet af udviklingen benytte design-metoden ICONIX (Doug Rosenberg, 2007) som det blev gennemgået i CPH modulet "Systemudvikling".

ICONIX metoden er en minimalistisk, effektiv tilgang baseret på UseCase-drevne UML-modeller. Unified Modelling Language – UML (Fowler, 2003), er et sæt af symboler som er velegnede til, på abstrakt niveau, at beskrive og designe processer i software under udvikling.

ICONIX og UML er tæt knyttet til det objektorienterede udviklings-paradigme (OOP, 2022).

2.1 ICONIX begreber og processer

Første trin består i at samle en såkaldt Funktionel Beskrivelse (trin 1), som i klart sprog beskriver hvordan applikationen forventes af præsentere sig overfor brugeren, og hvordan interaktion foregår.

På basis af den funktionelle beskrivelse, kan der dannes en såkaldt Domæne Model (trin 2). Denne bruges til at identificere objekter som er relevante at modellere i softwaren, og ikke mindst deres indbyrdes relation.

Næste trin er at bygge Use Case Packages (trin 3), som i bulletpoint-form lister de primære trin for hver case. Desuden bygges et diagram over trinenes indbyrdes relation og afhængighed.

Dernæst skabes en design-skitse for softwarens GUI (trin 4), gerne i en forsimplet form som lægger vægt på funktion snarere end form. Overvejelse om grænsefladen kan også påvirke strukturen af domæne-modellen.

Nu skrives komplette Use Cases (trin 5), som i klart sprog og abstrakte detaljer beskriver hvordan en facet af brugerens interaktion med softwaren skal foregå.

Robustness Analysis (trin 6) er et ICONIX-specifict begreb som sigter efter at koble Use Cases og objekter. Vi søger at identificere domæne-modellens objekter i de analyserede Use Cases.

ICONIX-processen lægger op til at hele design-fasen er iterativ, hvor en opdagelse på ét niveau kan påvirke tidligere beslutninger og kræve at de revurderes og tilpasses.

2.1.1 Funktionel Beskrivelse (trin 1)

De med rødt fremhævede navneord er særligt interessante ifht. domæne-modellen:

Vi skal udvikle en applikation som giver sin bruger mulighed for at udvælge et eller flere målsystemer, og gennemsøge logs fra disse.

Brugeren kan oprette nye målsystemer og tilføje dem til en målsystem-liste. Oprettelse indebærer definition af en kildefolder, dvs. hvor på netværket findes logfilerne. Brugeren definerer hvordan en loglinje fra målsystemet er struktureret, dvs. opretter en template for målsystemets loglinjer.
Brugeren kan markere et målsystem og aktivere indlæsning af logs.

Brugeren kan afgrænse en søgning ved at angive ét eller flere nøgleord i en nøgleords-liste som skal forekomme i en loglinje, og angive et tidsinterval som log-begivenhedens tidspunkt skal ligge indenfor.

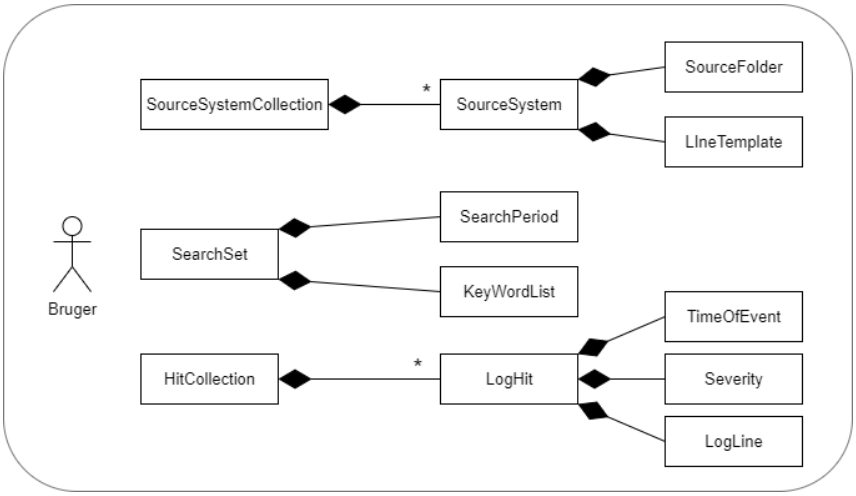
Når søgningen er gennemført, og der er fundet resultater, skal disse loglinjer præsenteres for brugeren i en hitliste. Hvis brugeren klikker på en linje i hitlisten, vises hele linjen.

2.1.2 Domæne Model (trin 2)

Den funktionelle beskrivelse identificerer nøglebegreber som er objekt-kandidater til domænemodellen.

Nøglebegreber	Domæne model objekter
Målsystemliste	SourceSystemCollection
Kildefolder	SourceFolder
Loglinje	LogLine
Template	LineTemplate
Målsystem	SourceSystem
Søgning	SearchSet
Nøgleord	KeyWord
Nøgleordsliste	KeyWordList
Tidsinterval	SearchPeriod
Tidspunkt	TimeOfEvent
Hitliste	HitCollection
Severity	Severity

Selve domænemodellen optegnes på basis af disse domænemodel-objekter. Her bliver objekternes relation tydelig.

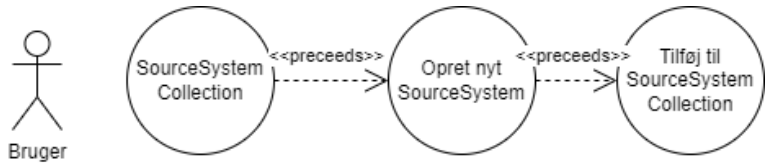


2.1.3 Use Case Packages (trin 3)

Kortfattet overblik over Use Cases, domæne objekter og interne relationer.

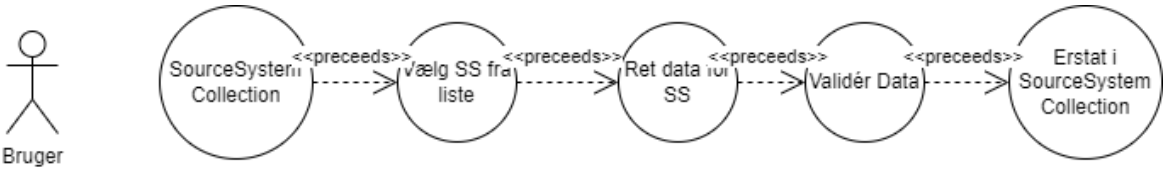
Create New SourceSystem

Opret nyt målsystem



Edit an existing SourceSystem

Rediger data for et eksisterende målsystem



Select an existing SourceSystem

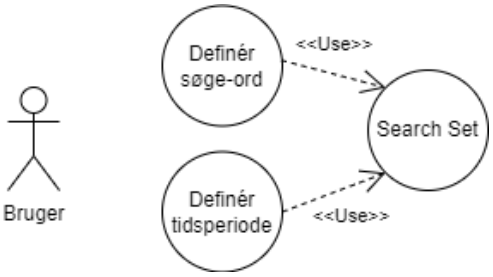
Vælg ét målsystem fra listen

Update logs for a selected SourceSystem

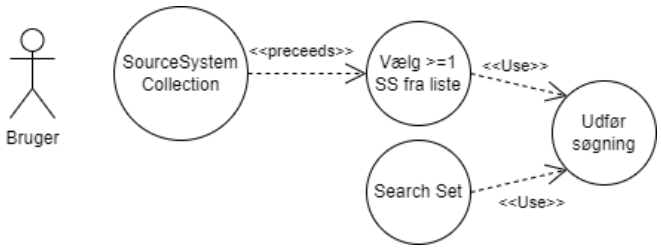
Vælg ét eksisterende målsystem.
Opdatér logs for det pågældende system

Define SearchSet for a search

Indtast 0 eller flere søgeord.
Angiv en start og slut-periode for søgningen.



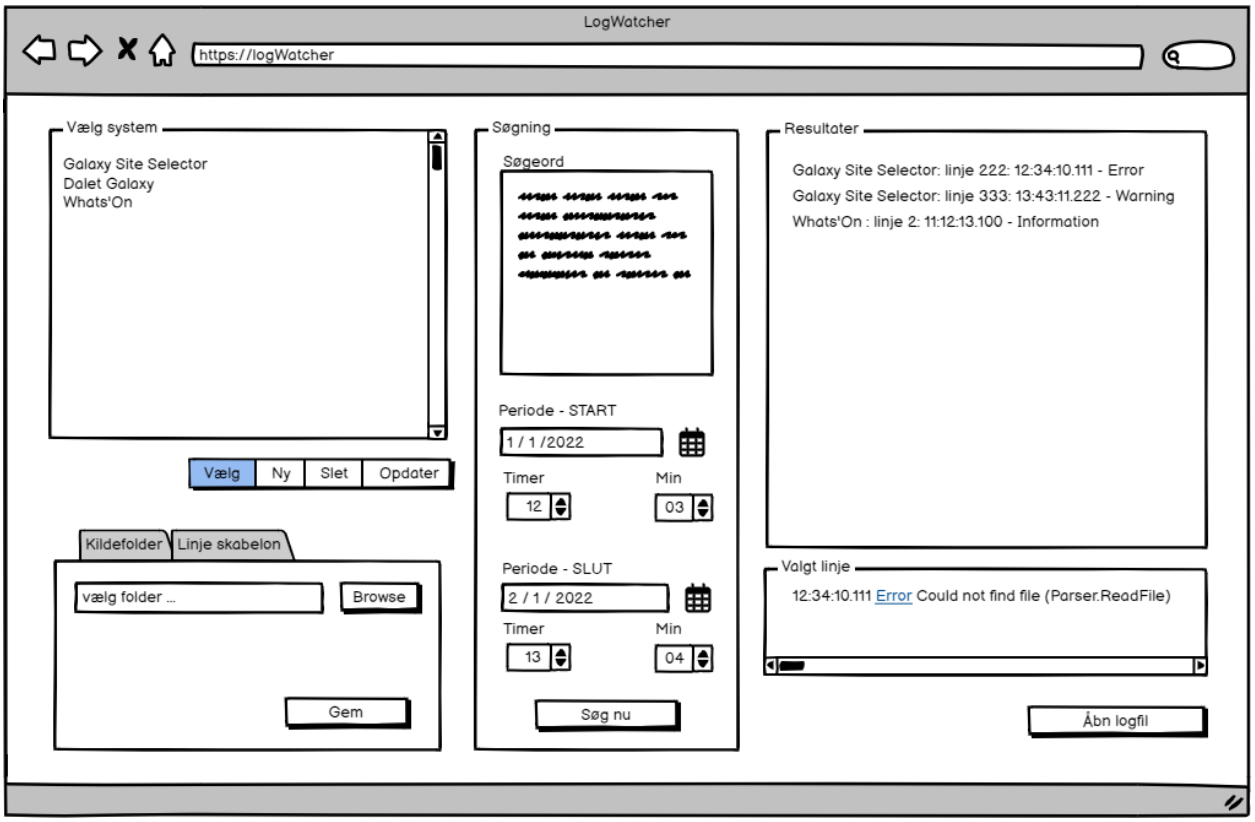
Execute search
Brug de anførte søgekriterier til at gennemsøge de valgte målsystemer



View returned hits from search
Efter gennemført søgning, vises listen over returnerede loglinjer

2.1.4 Grafisk Interface – GUI (trin 4)

Simpel skitse for interaktion med bruger.



(https://balsamiq.cloud/skqgr93/p6284we/r6B57)

2.1.5 Use Cases (trin 5)

Ved at udvælge og beskrive brugerhandlinger kan vi sætte domænemodellen i spil.

Create New SourceSystem

Brugeren ser en liste over allerede eksisterende *SourceSystem's* som er oprettet i *SourceSystemCollection*. Hvis listen er tom, vises intet. Brugeren klikker på knappen "Ny", og der vises felter for indtastning af *SourceFolder*, og definitionen af målsystemets *LineTemplate*.

Hvis de indtastede data passerer validering, aktiveres "Gem"-knappen. Hvis brugeren trykker på "Gem", bliver det ny system tilføjet *SourceSystemCollection*.

Edit an existing Source System

Brugeren ser en liste over allerede eksisterende *SourceSystem's* som er oprettet i *SourceSystemCollection*. Brugeren vælger ved at klikke i listen.

Objekterne *SourceFolder* og *LineTemplate* for den valgte liste vises i et separat vindue. De kan nu redigeres. Når brugeren trykker på "Gem" bliver informationerne valideret, og data skrives tilbage i objekterne. Derefter opdateres *SourceSystemCollection* med det rettede *SourceSystem*.

Select an existing SourceSystem

Brugeren ser en liste af *SourceSystems* indeholdt i *SourceSystemCollection*. Brugeren vælger ved at klikke i listen. Listen giver mulighed for at vælge ét eller flere systemer (multiselect).

Update logs for a selected SourceSystem

Brugeren ser en liste af *SourceSystemer* indeholdt i *SourceSystemCollection*. Brugeren vælger ved at klikke i listen. Hvis kun ét system er valgt, bliver knappen "Opdater" aktiv.

SourceFolder for det valgte system bliver scannet, og evt. nye filer bliver indlæst i databasen.

Define SearchSet for a search

Hvis brugeren har klikket på mindst ét *SourceSystem*, bliver søge-vinduet aktivt.

Brugeren kan skrive et antal ord som bliver tilføjet *KeyWordList*. Der kan også anføres et tidspunkt for tidligste forekomst, og seneste forekomst. Tidspunkter anføres i datoformat, og med mulighed for at angive time/min. Tidspunkter tilføjes til *SearchPeriod*. *KeyWordList* og *SearchPeriod* føjes til et *SearchSet*-objekt.

Execute search

Hvis søgevinduet er aktivt, kan brugeren klikke på knappen "Søg nu". De betingelser som er er defineret i *SearchSet*-objektet bliver anvendt til at begrænse søgeresultatet.

View returned hits from search

Hvis en søgning er gennemført, og hvis *HitCollection* indeholder elementer, bliver hvert *LogHit* i *HitCollection* vist i "Resultater"-vinduet. Et *Loghit* vises i reduceret form.

Brugeren kan klikke på et *LogHit* i vinduet, og så vises hele linjen i vinduet "Valgt linje".

Hvis brugeren har valgt en linje, bliver knappen "Åbn logfil" aktiv. Hvis brugeren klikker, bliver hele kilde-log-filen åbnet i et nyt vindue.

2.1.6 Robustness Analysis (trin 6)

Her tilstræber vi at reducere tvetydighed i domænemodellen og Use Cases. Det primære værktøj til formålet er opbygning af et **Robustness Diagram** (RD).
I et RB indgår en række symboler som repræsenterer specifikke aspekter af det modellerede system: Boundary, Entity og Controller.

Boundary Object: Denne objekttype repræsenterer grænseflader. Dvs. entiter som optræder i f.eks. GUI, og udstiller et aspekt af systemets tilstand, og/eller tillader brugeren at interagere med aspektet.

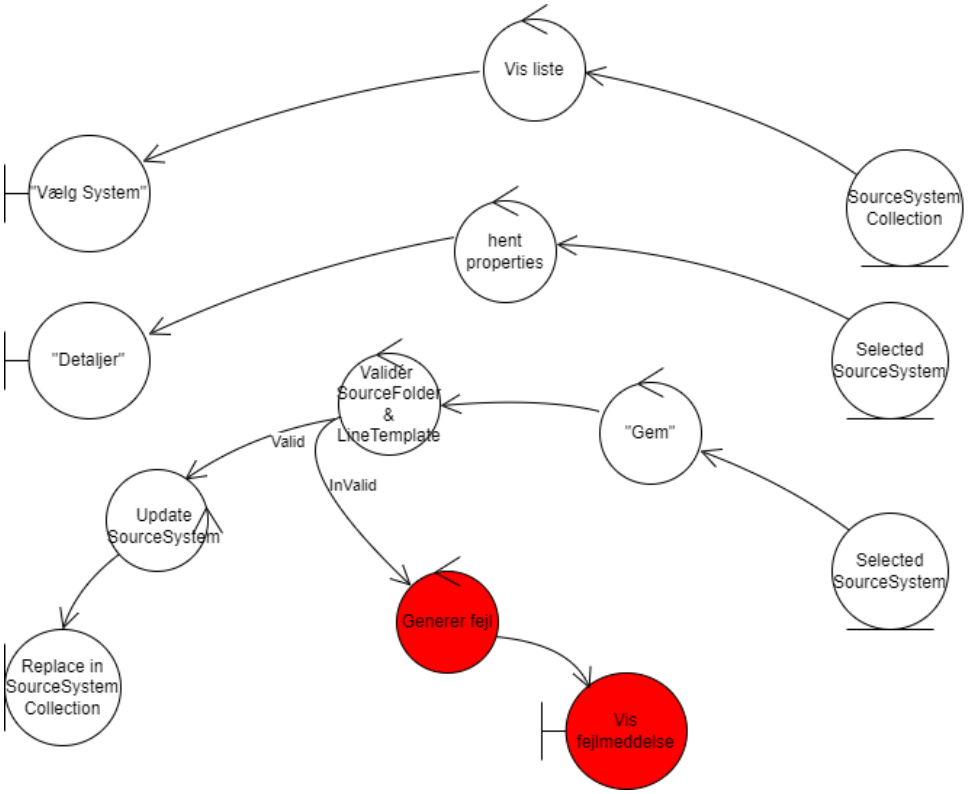
Entity Object: En repræsentation af et domænemodel-objekt, dvs. et objekt som repræsenterer systemets tilstand.

Controller: Repræsenterer typisk en process, dvs. en handling som udføres eller sæt data som transformeres.

2.1.6.1 Robustness Diagram

Jeg har valgt at afgrænse analysen til kun at omfatte casen "**Edit an existing sourcesystem**".

Under analysen viser RD behovet for at håndtere fejl-scenariet hvor de nye data for *SourceFolder* og *LineTemplate* ikke kan valideres. Ved fejl bliver der genereret en bruger-rettet fejlmeddelelse.



2.2 System analyse - konklusion

Med analysen i hånden er opgaven med at kode LogSearchers front-end blevet reduceret til et spørgsmål om hvordan selve implementeringen skal realiseres. De svære beslutninger om struktur og interaktivitet er taget, og grænsefladerne mod applikationens øvrige komponenter er blevet klarere.

Næste skridt er at belyse de teknologiske rammer som omgiver applikationen.

3 TEKNOLOGI OVERBLIK

3.1 Moderne udviklingssprog – de 3 kandidater

Fordelt over de 3 programmerings-moduler jeg har taget under 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. (Top Programming Languages, 2022)

De tre sprog har som fællestræk at programmets forbrug af hukommelse under kørsel håndteres i baggrunden, af det miljø som sættes op for at afvikle koden. Denne proces kaldes Garbage Collection (GC), og betyder at allokeret hukommelse automatisk frigøres når et objekts livs-cyklus er slut. (Automatic Memory Management, 2022) (Languages, 2022)

Eksempler på sprog-miljøer uden GC er C og C++, som derfor også foretrækkes til problem-domæner hvor der er begrænset hukommelse eller krav om tids-kritisk eksekvering.

3.1.1 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 indholds karakter 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 som de facto normen for at kode interaktivitet på web (sammen med html/css) bl.a pga. dets kompakte og fortættede syntax. I den kontekst køres JavaScript i brugerens webbrowser.

3.1.2 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. (Java, 2022)
C# er knyttet til kørsel- og udviklingsmiljøet .NET, som er Microsofts bud på et kode-økosystem til Windows-platformen (.NET, 2022). 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).

3.1.3 Python

kan placeres imellem JavaScript og C#. Det kan kodes både i et objektorienteret og et funktionelt paradigme, og er typesvagt. Python-syntax er en meget kompakt kodestil, og er særligt populært til at processere store datamængder i få linjer kode. Python-fortolkere findes i varianter egnet til de mest populære miljøer, Windows, OSX og Unix/Linux. (C# vs Python, 2022), (C# vs Python 2, 2022), (Python for Science, 2022)

3.1.4 Udviklingssprog - opsummering

Hvert enkelt sprog har specifikke styrker og fordele. Python giver mulighed for meget elegant og effektiv rekursion som kan være nyttigt til at bygge søgemotoren. C# understøtter multi-threading (Multithreading in .NET, 2017) indenfor samme app-instans, men det gør Python til gengæld ikke. (TowardsDataScience, 2020) (Python code parallelization, 2016)

3.2 Database teknologier - RDBMS vs. NoSQL

Inden vi beslutter hvilken database-teknologi LogSearcher 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.

3.2.1 Relationel Database Management Systems

Begrebet RDBMS opstod i 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 eller flere elementer i en eller flere andre tabeller. 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.

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 og robusthed.

3.2.2 NoSQL

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

NoSQL er mere dynamisk orienteret end SQL, og er mere modstandsdygtigt overfor ændringer og tilpasninger i systemets datastruktur, 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.

3.2.3 Skalering – RDBMS vs. NoSQL

I takt med at performance-kravene vokser, kendetegnes de to typer (RDBMS/NoSQL) 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. (SQL vs. NoSQL vs. NewSQL, 2016)

3.2.4 Valg af database - konklusion

På baggrund af den datastruktur jeg vil beskrive i afsnit 4.2, en velnormaliseret struktur med minimalt behov for tabel-joins, vurderer jeg at et RDBMS vil være bedst egnet til løsningen.

3.3 Microsoft .NET i DR

På DR anvendes flere forskellige teknologier. Som Public Service Broadcaster (Public Broadcasting, 2022) midt i en digital transition mod web, er en meget stor del af husets seer/bruger-rettede 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 de fleste af de fundamentale produktions-systemer i huset baseret på off-the-shelf produkter. Men integrationslagene mellem produktion-systemerne er i høj grad egenudviklede. 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 Microsoft .NET og C#.

LogSearcher vil indgå naturligt som et støtteværktøj for 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 Microsoft .NET som udviklingsmiljø, skabes det bedste potentiale for at forankre værktøjet hos brugerne.

3.4 Brugerplatform

En væsentlig overvejelse i forbindelse med valget af teknologi-platform for LogSearcher, vil 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 platform skal den køre på? - Windows, *Nix, OSX ?
Skal den afvikles lokalt på brugerens klient eller centralt fra en server?
Hvem håndterer daglig drift og problemløsning hvis der opstår fejl?
Kan den virtualiseres eller måske køres i en cloud-løsning?

I DR eksisterer Windows, Linux og Apple OSX side-om-side, og bliver anvendt af overlappende brugergrupper. Alle servere er enten *Nix (Unix-like, 2022) eller varianter af Windows Server, mens klienterne enten kører Windows eller OSX.

Jeg anser det for et krav at både Windows og OSX-brugere skal kunne anvende LogSearcher.

3.4.1 Brugerplatform - konklusion

Det gør det meget oplagt at beslutte at kernen af applikationen skal afvikles centralt fra en server, og at grænsefladen præsenteres via en web-browser, baseret på en HTML-side. Derved undgår jeg at skulle udvikle OS-specifikke klienter. Vedligehold og fejlretning kan også bedre fokuseres, hvis applikationen ikke er distribueret over en antal klienter, som alle skal holdes opdateret.

3.5 Hostingmiljø

Med en klar beslutning om at basere LogSearcher på en klient/server arkitektur, er næste spørgsmål hvordan miljøet omkring serveren skal bygges.

3.5.1 VM's vs. containers

I moderne IT-drift er det ikke længere praksis at afsætte én komplet server pr. applikation. I stedet har man en indført lag af abstraktionsniveauer over hardware-niveauet og introducere en generaliseret arkitektur.

Overordnet set er der i dag 2 grene af abstraktion, Virtual Machine (VM) og containers. Container-teknologier som Docker og Kubernetes bliver oftest anset som cloud-baserede.

Virtual Machine: 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 VM vil typisk være vært for ét komplet system, på tværs af mange processer.

Container: 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. Èn container vil som hovedregel kun være vært for én proces eller applikation

3.5.2 Virtual Machines @ DR

Sammendrag af interview med Thomas Borup, DR, MS Infra-team.
Thomas og hans team håndterer drift af alle IT-systemer i DR, både administrations- og produktions-systemer. Deres domæne er Virtual Machines.

DR i skyen: *"DR har for nyligt besluttet af være "Cloud First", efter at have været OnPremise-orienteret."*
OnPremise indebærer at den fysiske hardware er servere placeret i bygningen. Cloud er i DR-kontekst lig med Microsoft Azure. Det indebærer at nye systemer i videst muligt omfang skal deployes til Azure. Azure giver mulighed for både at instantiere komplette VMs, eller modulære komponenter som f.eks en SQL-database. Komponent-tilgangen er favoriseret, fordi der er mindre ressource-overhead på et drifte en komponent snarere end end komplet VM.

Hardware: *"Mit team håndterer 442 fysiske serverer, og 1065 virtuelle maskiner. Af de 442 fysiske servere er 40 værter for virtualiseringsmiljøet. Vi anvender Microsoft HyperV til drift af windows-systemer, og VMWare til linux. I dag bruger vi MS Virtual Machine Manager til håndteringen"*

3.5.3 Containere @ DR

Sammendrag af interview med Sigurd Kristensen, DR, Linux-team.

Ny platform: *"Vi er ved at at bygge vores nye platform. Det bliver Kubernetes-baseret, med adskillige clusters til test, produktion osv.*"

Kubernetes er et system til håndtering og drift af container-samlinger og Docker er det mest udbredte container-system. Tilsammen udgør de en platform som giver stor fejl-tolerance, horisontal skalérbarhed og isolation mellem klient system-domæner. (Kubernetes: Up and Running, 2022)

Stateless vs. Statefull: "*DRs container-strategi indebærer at vi foretrækker 'StateLess' fremfor 'StateFull'. Vi ønsker at kunne rive hele containeren ned når vi re-deployer en app via vores pipeline*"

Stateless-paradigmet betyder at en applikation altid deployes i samme tilstand, dvs. med nøjagtigt samme data. Det gør modellen uegnet til f.eks databaser, som netop er kendetegnet ved at skulle reflektere et systems aktuelle tilstand. (Stateless Vs. Stateful, 2019)

3.5.4 VM's vs. Containers - Konklusion

Kubernetes og Docker-containere giver mulighed for at skalere drift ifht. brugsbelastning. Det er derfor oplagt at drifte LogSearchers komponenter på en Kubernetes docker-platform. Derimod bør databasen hostes på en Azure VM (eller komponent), qua det beskrevne 'stateless'-paradigme.

4 TEKNOLOGI I LOGSEARCHER

4.1 Data-repræsentation – fra kilde til database

Den fælles kilde-type for alle målsystemer vil være de ustrukturerede rå-data som logfilerne udgør. Dvs. LogSearcher har ikke adgang til strukturerede metadata om logfilens format.

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

En system-log består af én eller flere linjer af log-events, som mål-applikationen genererer under drift. og de forventes at karakterisere applikationens tilstand. Dvs. at der typisk skrives i loggen, når en væsentlig hændelse har fundet sted.
F.eks. gennemført brugerinteraktion, eller indlæsning af eksterne data.
Eller når der opstår en fejl eller fejltilstand.

På basis af logs fra de 3 eksempler, har jeg identificeret 4 primære elementer som karakteriserer et log-event.

TimeOfEvent: hvornår fandt begivenheden sted
Severity: hvor alvorlig vurderes hændelsen at være, fra simpel information til fatal fejl
Event-description: en mere detaljeret beskrivelse af fejlen og evt. relateret tilstand
Source-module: navnet på det modul som fejlen eller meddelelsen stammer fra.

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.

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***

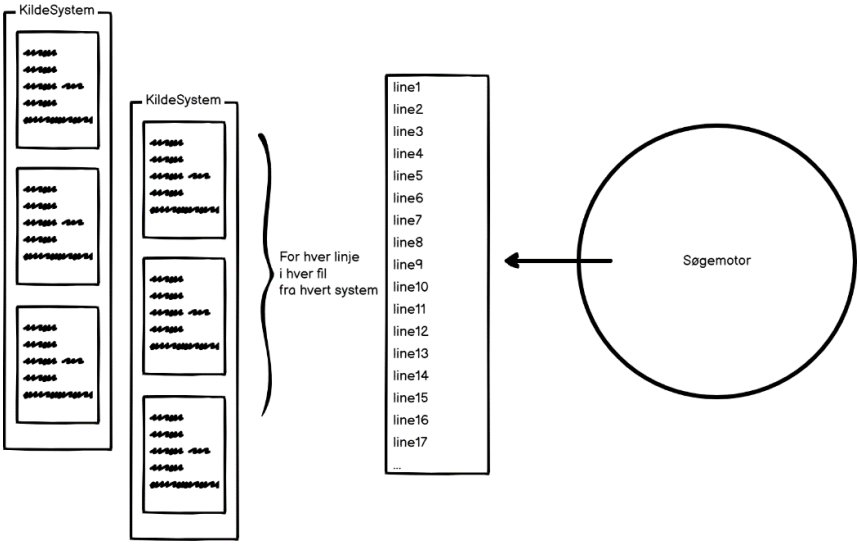
Tabel: Her er 3 eksempler fra kurante DR-systemer * (ISO 8601, 2022)

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 derfor nødvendigt at filtrere og re-formatere log-data inden persistering.

4.2 Database struktur

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



For at optimere denne proces og minimere flaskehalsen, skal databasens struktur understøtte en effektiv transformation fra kildefiler til datarecords og facilitere hurtig og smidig tilgang fra søgemotor.

Fra logfil til søgbart indhold:

For hvert **kildesystem** er givet et antal **kildefiler** som indeholder et antal **textlinjer**.

Søgemotoren arbejder på ord-forekomster i tekst-linjerne, og når der er fundet et match, skal det kunne spores tilbage til kildefil og -system.

Det medfører denne tabel-struktur:

Table: **SourceSystems**

ID (int)	Name (string)	SourceFolder (string)	LineTemplate (string)
10	MA-Ingest	C:/logs	0-15^

Table: **LogFiles**

ID (int)	SourceSystemID (int)	FileName	FileHash
1	10	Logfile.txt	xxxxxx

Table: **Loglines**

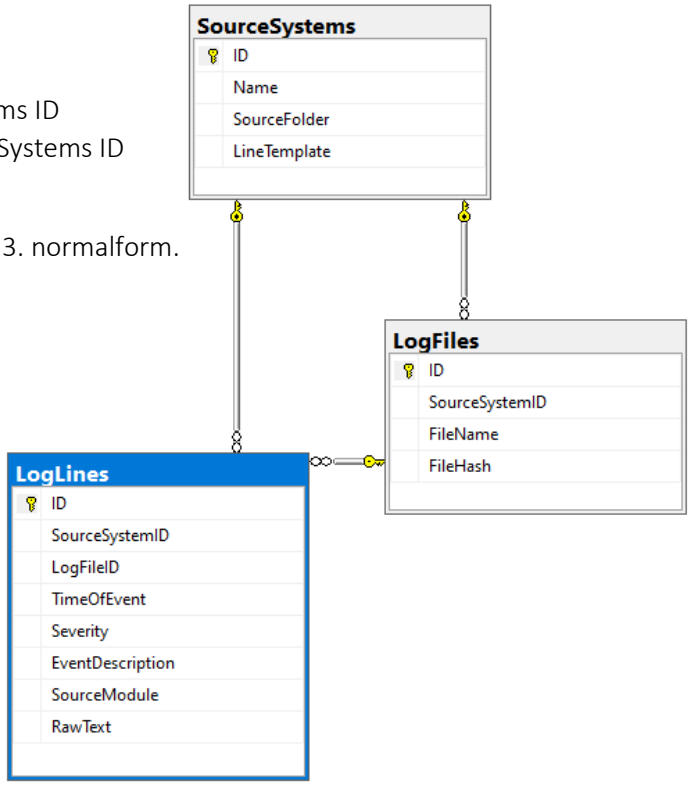
ID (bigint)	SourceSystemID (int)	LogFileID (int)	TimeOfEvent (dateTime)(nullable)	Severity (string)(nullable)	EventDescription (string)	SourceModule (string)(nullable)	Rawtext (string)
10000	10	1	---	ERROR	Failed while uploading user data	Main.GetData	

Når der er fundet et match i en log-linje, kan kildesystem og -fil udledes fra ID-referencerne i LogLInes-tabellen.

Entity Relations diagrammet for databasen illustrerer tabellernes relation.

Tabellen **LogFiles** har en FK til SourceSystems ID
Tabellen **LogLines** har en FK til hhv. SourceSystems ID og LogFiles ID.

Databasen er således fuldt normaliseret til 3. normalform.



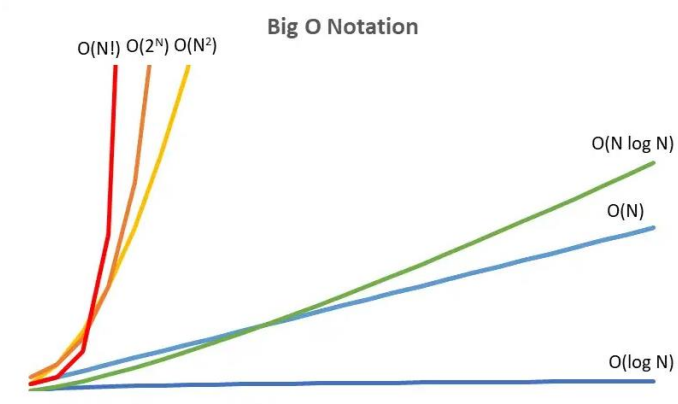
4.3 Søgning i store datasæt

LogSearchers relevans er baseret på at kunne finde specifikke ord i store datasæt, og knytte hvert sæt af forekomster til en kildefil/kildesystem.

Den naive tilgang til denne søgning er at gennemgå hvert linje ord-for-ord og danne en forekomst-liste. Hver gang der søges, skal alle linjer checkes for forekomst. Hvis ét check tager x millisekunder, vil y antal linjer tage x*y millisekunder. Altså en lineær relation mellem søgesættets størrelse og varigheden af søgning. Til at udtrykke denne type relationer anvendes normalt **BigO** notation.

4.3.1 BigO notation & TimeComplexity

"Big O" notation er en måde at måle en algoritmes effektivitet. Det udtrykker køretid i forhold til inputmængde. Altså hvordan funktionen skaleres.
Der er 2 primære aspekter af notationen: TimeComplexity og SpaceComplexity (hhv. køretid og hukommelsesforbrug). De angiver den øvre grænse for for vækstraten af funktionen.



Big O introduction

BigO er altså en måde at tale om en algoritmes effektivitet, da det giver en målestok for køretid/hukommelsesforbrug i forhold til det datasæt som anvendes til input. **Som regel er vi mest interesseret i Time Complexity**. (Big O introduction, 2021)

BigO versionen af eksemplet fra 5.2 er: O(n), hvor n er antallet af linjer.

Her illustreres gængse forekomster af relationen mellem input-mængde og processtid. X-aksen repræsenterer input, Y-aksen tidsforbrug.

4.3.2 Kandidater til indekseringsstrategi

Der findes et antal strategier til at håndtere sortering og søgning af datasæt. Jeg har vurderet de 3 følgende som kandidater til LogSearchers søgemotor.

4.3.2.1 Binary search på en sorted list

Udgangspunktet er en liste af værdier, **sorteret i størrelsesorden** (enten størst-først eller størst-sidst). Ved at starte i midten af listen, kan vi konstatere om søge-værdien er i øvre eller nedre halvdel. Derefter reduceres søgesættet til denne halvdel. Processen gentages indtil værdien enten er fundet, eller ikke eksisterer i sættet.
TC, Sortering af liste: O(n log n), søgning: O(log n)

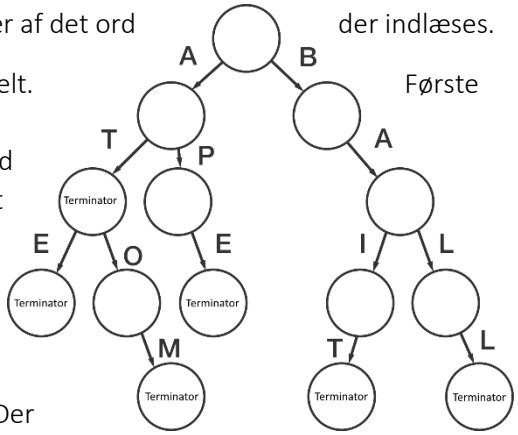
4.3.2.2 Hashtable

Hvert ord i søgesættet omsættes til en talværdi(hash) og placeres i en tabel af typen linked list. En linked list har for hvert element i listen en pointer til det næste element. Ved søgning omsættes søgenøglen til en hash efter samme metode. Tabellen traverseres indtil den søgte hash er fundet. Afhængig af hashing-metoden kan der forekomme kollisioner, hvor 2 input tildeles samme hash.
TC, Insert/retrieve: O(1)

4.3.2.3 Trie (prefix tree)

Ordet trie kommer af re**trie**ve, at hente. Det er en træ-baseret datastruktur, som er velegnet til at organisere store data-mængder, typisk tekst-strenge. Et trie har en tom rod-node (root) og referencer til under-noder (child-nodes).

Hver enkelt node-forbindelse (edge) repræsenterer en karakter af det ord der indlæses.



Når træet **POPULERES**, indlæses hvert enkelt bogstav sekventielt. Første sæt edges efter roden checkes for forekomst af det aktuelle bogstav. Hvis det allerede eksisterer, fortsætter processen med næste karakter i næste lag af træet. Ellers indsættes bogstavet i det aktuelle lag, og resten af ordets karakterer kædes efter hinanden i denne nye gren. Når det sidste bogstav indsættes, markeres dennes node som ord-grænse (terminator)

Når træet **LÆSES**, foregå traverseringen efter samme princip. Der startes fra rodnoden, og bogstav-for-bogstav undersøges om der findes edges i den aktuelle node som matcher bogstavet. Hvis travseringen ender på en slutnode, betyder at ordet er fundet. Et komplet ord indikeres altså ved at sidste node markeres som **terminator**/slut-node. I figuren ser vi at ordene {AT, ATE, APE, ATOM, BAIT, BALL} er markeret som afsluttede ord. Hvorimod {AP, ATO, BA, BAI, BAL} ikke er.

Tries anvendes ofte som erstatning for hashtables, bl.a. fordi de undgår problemet med kollisioner, og er nemme at ændre størrelse på.
TC, Indlæst: O(mn), søgning: O(an) (m = karakterer i længste ord, n=antal ord, a=længden af længste ord)

4.3.3 Konklusion: Indekseringsstrategi

Jeg vil vælge at implementere tries til søgemotoren, primært for at undgå hashtable-kollisioner og problemer med at ændre størrelsen. Da søgetiden i et indlæst trie er meget svarende til en hashtable, giver dette valg både god performance og stabil drift.

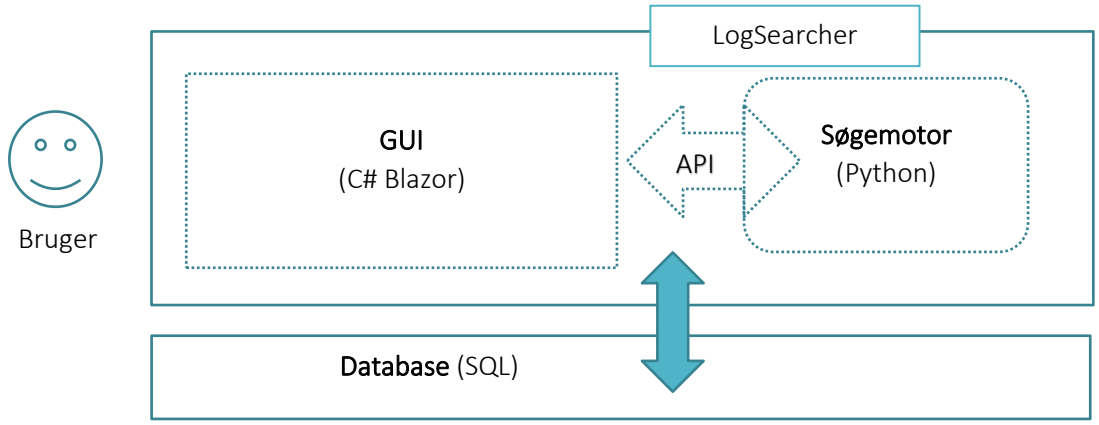
4.4 LogSearcher udviklingskomponenter

På basis af de foregående undersøgelser har jeg tegnet en profil af LogSearcher ud fra brugerbehov, driftskontekst, anvendt teknologi i DR, datastruktur og søgealgoritmer.

Det fører frem til beslutningen om arkitekturen for applikationen.

4.4.1 Grundlæggende arkitektur

Et grafisk overblik over komponenterne og deres relationer.



4.4.2 Udviklingssprog

Til LogSearcher vil jeg anvende C#, Python og MSSQL.

Python til de centrale aspekter af søgning i datasæt, i søgekomponenten.
C# til grafisk brugerflade (GUI) og persistering mod databasen.
MSSQL til databasen.

4.4.3 REST API

Ved at indbygge et REST API i søgekomponenten kan jeg udstille de primære søge-værktøjer mod det omsluttende C#/.NET lag som udgør den brugervendte applikation. API vil udgøre grænsefladen mellem GUI og søgemotor.

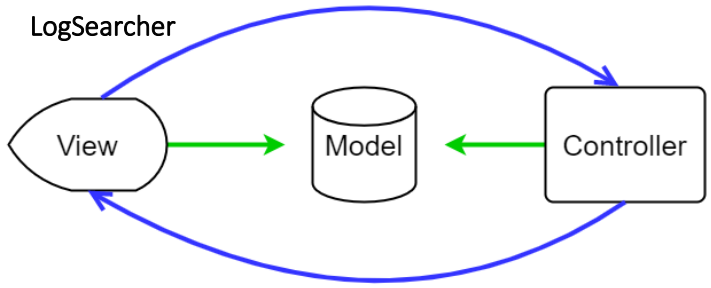
4.4.4 Blazor Server til GUI

Med .NET Core har Microsoft introduceret et web-orienteret lag til C#, som gør miljøet meget egnet til at udvikle websider. Komponenten hedder Blazor (What is Blazor, 2022), og giver mulighed for en eventdrevet, dobbeltrettet binding mellem grafik-objekter og de underliggende tilstande i applikationen. Blazor er designet til at underbygge SPA-modellen (Single Page Application), hvor brugeren ikke navigerer mellem forskellige sider på samme site, men hvor indholdet af siden i stedet udskiftes og opdateres dynamisk. Jeg vil anvende den hostede model, **Blazor Server**, hvor appen hostes på en IIS eller Kestrel-instans.

4.5 LogSearcher – en applikation i tre dele

LogSearcher skal udgøres af 3 distinkte komponenter, eller lag. Et brugervendt lag, et persisterings-lag, og et søge-lag. Denne struktur flugter med MVC-modellen, Model-View-Controller.

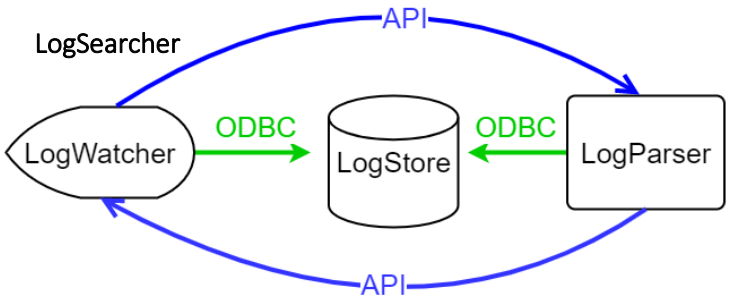
View: det brugervendte lag, GUI, som udstiller applikationens funktionalitet.
Controller: det lag som håndterer selve databearbejdningen.
Model: det lag som repræsenterer data og deres struktur.



View og controller-lagene bruger den fælles database til at persistere og hente data.

Når data er indlæst, starter brugeren søgningen. Derved sendes en kommando til controlleren, som henter det relevante søgesæt, og behandler det. Når søgningen er klar, sendes søgeresultatet til brugeren.

Det giver mening at navngive komponenterne ifht. deres rolle i denne illustration.



View: **LogWatcher**
Model: **LogStore**
Controller: **LogParser**

Både LogWatcher og LogParser vil anvende ODBC-protokollen til at skrive og hente data hos LogStore.

Interaktion mellem LogWatcher og LogParser faciliteres af et API.

ODBC er en udbredt standard for tilgang af RDBMS-systemer (ODBC, 2022). API (Application Programming Interface) refererer til en http-baseret udveksling af data via REST API (REST, 2022).

LogWatcher bliver i denne sammenhæng forsiden af LogSearcher-applikationen, som udgøres af 3 sammenkædede komponenter. Denne struktur giver flere fordele, bl.a. stor kompatibilitet med den foreslåede drifts-model. Hver enkelt komponent kan afvikles i et diskret miljø, som kan være fysisk og logisk afkoblet fra de øvrige.

4.5.1 LogParser

Jeg vil udvikle LogParser i Python, fordi det er velegnet til den type datahåndtering Tries kræver, som er baseret på rekursive algoritmer.

Når der sendes en søgning til LogParser, bliver der først dannet en liste over samtlige loglinjer som er indeholdt i det aktuelle søgesæt. Der søges i databasen på de valgte kildesystemer og den valgte periode, og alle linjer som matcher indeholdes i denne liste.

For hver linje indlæses hvert ord i en trie-struktur. Hvert komplet ord ender i en terminator. Terminatoren skal rumme en reference til ordets kilde, dvs. en pointer i formatet [kildesystem:logfil:loglinje]. Derved kan vi præcist identificere kilden til hvert ord-hit.

Når der søges, traverseres det data-bærende trie, og for hvert match tilføjes den tilsvarende terminator til en resultat-liste. Når alle grene er undersøgt, er søgningen afsluttet og resultat-listen er klar.

4.5.2 LogWatcher

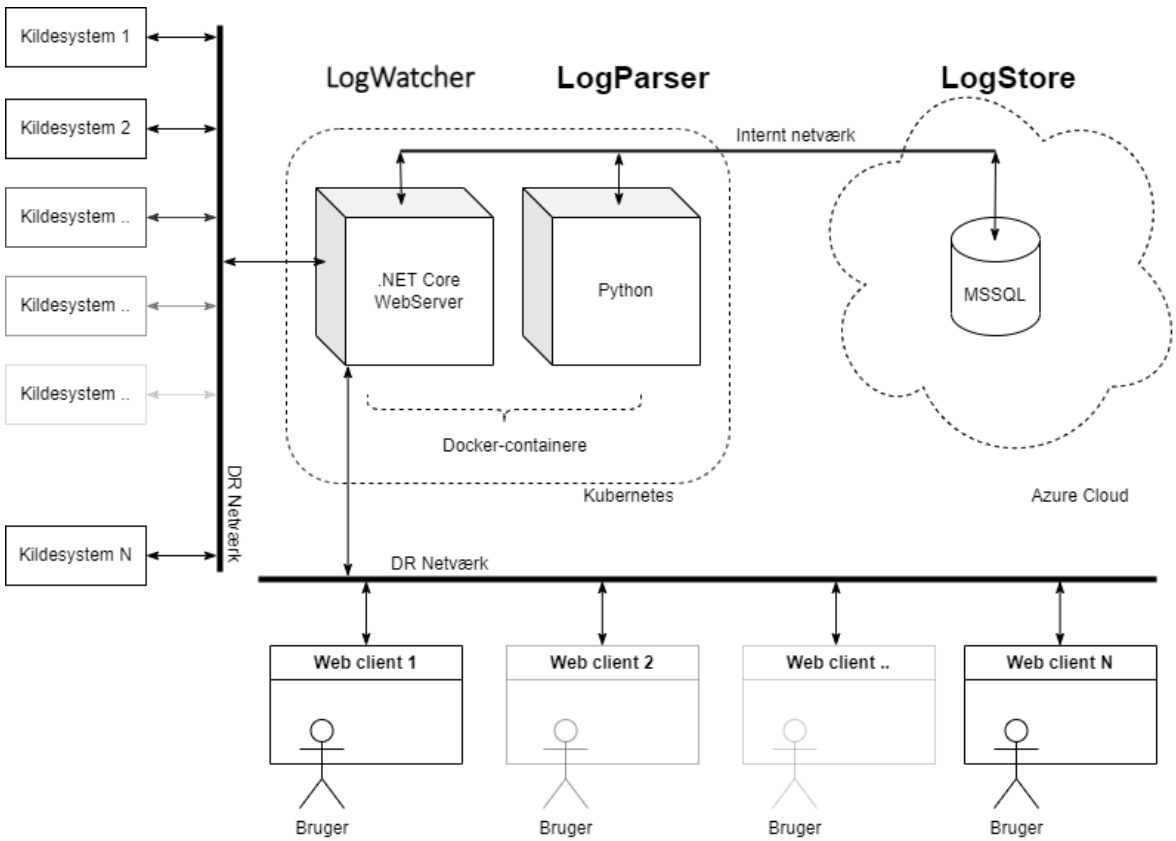
Det brugervendte aspekt af applikationen er i store træk velbeskrevet i kapitel 2 – SYSTEM ANALYSE.

4.5.3 LogStore

LogStore er velbeskrevet i kapitel 4.2 – Database Struktur.

4.6 Driftsmodel

Driftsmodellen viser arkitekturen og komponenternes relation til det omgivende miljø, hosting-platforme, kildesystemer og brugere



5 IMPLEMENTERING

I kapitel 1-4 har jeg defineret problem-domænet, og beskrevet rammerne for en mulig løsning ift. problemformulering. I denne proces har jeg tegnet både et konceptuelt og et praktisk billede af LogSearcher som en løsning.

Den nærværende opgave beskriver den fuldt udfoldede implementering, samt overvejelser derom og visse konkrete, praktisk realiserede aspekter. Det medfølgende produkt repræsenterer derimod en tilnærmelse til den fuldt implementerede version.

5.1 En prototype - afgrænsning af opgavens omfang

Til min opgavebesvarelse har jeg udviklet en prototype af den fulde funktionalitet og implementeringsmodel som indtil nu er beskrevet.

Den foreslåede struktur og jobfordeling mellem komponenterne er bevaret i denne prototype.

5.1.1 LogSearcher prototype @ Github

Denne opgavebesvarelse er tilgængelig på https://github.com/FaderVader/CPH_Afgangsprojekt_2022

Selve kildekoden findes under hhv. /src/LogWatcher og /src/LogParser

5.2 LogParser

Komponenten er stort set realiseret i sin endelige form.

LogParser udstiller mere funktionalitet end der bringes i spil i denne prototype. Bl.a. giver den mulighed for at finde start/slut perioder for forekomst af sæt af søgeord, f.eks. for at finde start/stop af længere-kørende processer.

I den endelige implementering skal LogParser kobles sammen med en webserver, for at kunne udstille sit API mod LogParser. Denne feature er ikke udviklet til prototypen.

Her følger et overblik over klasserne og deres metoder.

Workers: *Shell, QueryParser, Query, PrepareTrie, Loader*.
De 5 primære workere arbejder i en kæde hvor 1. led starter og konsumerer de følgende:

Shell modtager søgekommandoerne, og delegerer konstruktionen af den aktuelle søgning som brugeren sender. ***QueryParser*** modtager søgningen, og aktiverer de relevante metoder. ***Query*** er wrapper for selve søgefunktionaliteten. ***PrepareTrie*** kalder Loader for at populere søge-træerne. ***Loader*** henter data fra LogStore.

LogParser anvender 2 forskellige trie-træer samt et binært søge-træ.

SearchTrie holder styr på antallet af hits når der søges efter flere ord.
LogTrie indeholder for hver forekomst af et ord, pointere til kilde-loglinjen (1 eller flere).

```
# actual trie builder
def addWord(self, word, terminator):
    def inner(word, node):
        if len(word) == 0:
            if node.value is None:
                node.value = [terminator]
            else:
                node.value.append(terminator)  # elements are concatenated: multiple hits are expected
            return
        elif word[0] not in node.children:
            newNode = Node(word[0])
            node.children[word[0]] = newNode
            return inner(word[1:], newNode)
        else:
            return inner(word[1:], node.children[word[0]])
    return inner(word, self.root)
```

*(vist kildekode: class **Logtrie** fra Tries.py)*

Metoden **addWord()** rummer en indre metode som kaldes rekursivt for hvert bogstav i ordet som indlæses. LogTrie bruger terminatorer til at markere ordgrænser. En terminator definerer en reference til kildesystem:logfil:loglinje. For hvert hit på samme ord, tilføjes et terminator-element.

Det binære søgetræ (BST), anvendes kun til at sortere hits efter tid. BST traverseres InOrder for at udtrække det sorterede sæt.

Models: ***LogLine, SearchSet, SourceSystem, SearchPeriod.***
De databærende klasser. ***LogLine*** rummer metoder til at udtrække tidspunkt og tekst-elementer fra en loglinje. ***SearchSet***, ***SourceSystem*** og ***SearchPeriod*** repræsenterer samme objekttype som i LogWatcher.

Wrappers: ***Database, Api.***
Database udstiller ODBC-værktøjer til database-interaktion med LogStore.
Api tilføjer en http-grænseflade til LogParser.

5.3 Søgning via LogParser

LogParser bliver udstillet til LogWatcher via et minimalt API. Dette API er grundlæggende en wrapper for indgangspunktet for LogParser (Shell).

API'et udstiller 2 endpoints: /search og /research.

/search anvendes hvis søgningen er ny, dvs. hvis gruppen af loglinjer er ny.
/research anvendes hvis det er samme sæt loglinjer, men nye søgeord.

LogWatcher sender LogParser et json-dokument som en serialiseret version af LogWatchers SearchSet-objekt. På basis af søge-argumentet, startes en forespørgsel mod LogStore. De modtagne loglinjer loades i LogTrie. Derefter forespørges LogTrie på hits for søgeordene, og de tilsvarende terminatorer parses for at identificere loglinjerne.

Når søgningen er afsluttet, dannes et svar-datasæt: et array af (sourcesystemid, logfileid, loglineid).

Svaret udstilles via API, som LogWatcher query'er og derefter danner en liste over hits. LogStore forespørges på elementernes ID, og LogWatcher danner hitlisten og udstiller dem for brugeren via GUI.

5.3.1 Komponerede søgekriterier

QueryParser anvendes ved at kalde en gruppe en liste af metoder og argumenter til at komponere den endelige forespørgsel. QueryParsers metoder kaldes via refleksion. (Reflective Programming, 2022)

Shell bygger en sekvens af query-elementer som sammenstilles før de endeligt gives som argument til QueryParser, som derefter udfører søgningen. Denne struktur udnyttes f.eks. til at differentiere søgninger fra Api, mellem at anvende hhv. nyt søgegrundlag og et eksisterende.

@ API, ny søgning (/search)

```
def Search(self, searchSet: SearchSet):
        self.results = None
        self.shell = Shell(searchSet)
        self.shell.do_find(searchSet.KeyWordList)
        self.shell.do_sort('True')
        self.results = self.shell.do_run()
```

@ Api, eksisterede (/research)

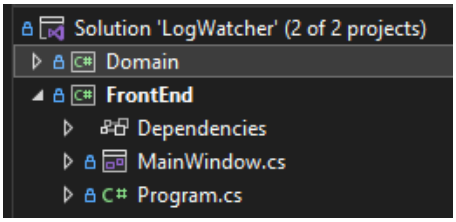
```
def ReSearch(self, searchSet: SearchSet):
        if (self.shell == None):
            raise HTTPException(status_code=400,
detail="Previous search not valid")
        self.results = None
        self.shell.do_find(searchSet.KeyWordList)
        self.shell.do_sort('True')
        self.results = self.shell.do_run()
```

Kørsel af komponent: For at kunne simulere funktionaliteten og interaktionsmodellen for komponenterne af LogSearcher, har jeg valgt at bruge uvicorn (Uvicorn, 2022) som stand-in webserver.

5.4 LogWatcher

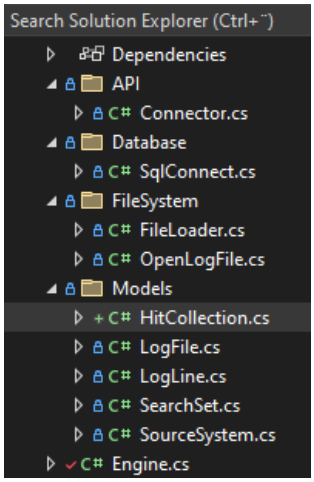
LogWatcher prototypen er realiseret i Windows Forms .NET 6.

Funktionaliteten er distribueret mellem projektet **FrontEnd** og library'et **Domain**. FrontEnd varetager GUI og brugerinteraktion, og Domain indkapsler bl.a. database interaktion og et minimalt http-interface.



Domæne-typerne som blev beskrevet i afsnit 4, er implementeret i Domain/**Models**.

Klassen API.**Connector** håndterer det minimale http-interface.



Database.**SqlConnect** udgør database-grænsefladen, og FileSystem.**FileLoader** udstiller metoder til at indlæse logfiler.

Klassen **Engine** orkestrerer og udstiller funktionalitet mod FrontEnd.

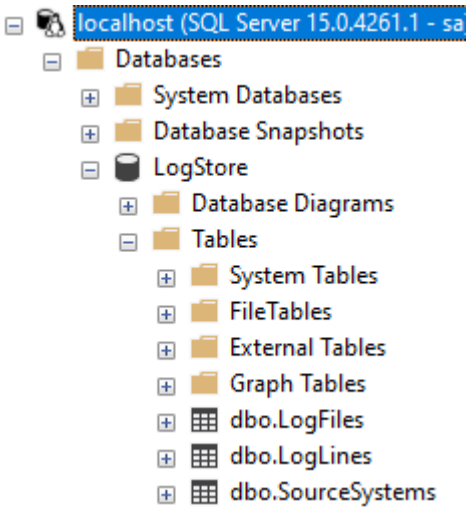
Koden i FrontEnd består primært af event-håndtering (når brugeren klikker på knapper osv).

Jeg valgte at udføre protypen i WinForms, fordi formatet er meget velegnet til hurtigt at bygge et grafisk interface og understøtte bruger-interaktion. Det giver mig mulighed for at fokusere på at udvikle modellen for hvordan komponenterne skal fungere sammen, uden at skulle opbygge et fuldt website.

Kørsel af komponent: Til brug for prototypen startes blot build-outputtet (.exe) af FrontEnd i et windows10 miljø.

5.5 LogStore

LogStore er realiseret i en MSSQL-instans.



I realiteten er databasen umiddelbart klar til produktionsbrug. De tre tabeller som systemet er afhængigt af, kræver meget lidt tilpasning før udrulning.

Kørsel af komponent: Jeg har til prototypen valgt at afvikle en MSSQL-instans i en Docker container i mit udviklings-miljø.

Instansen afvikles via Docker Desktop.

6 AFSLUTTENDE KONKLUSION

Med nærværende opgave, samt det tilhørende stykke praktisk implementering, har jeg demonstreret hvordan en løsning for applikationen LogSearcher kan udformes. Jeg har vist hvordan applikationens elementer kan bygges, og hvordan de bringes i samspil.

Prototypen realiserer 100% af den funktionalitet som jeg foreslog i problemformuleringen.

Selve driftsmodellen for prototypen afviger for den foreslåede model, hvor LogWatchers server-komponent og LogParser afvikles som docker-instanser, og LogStore på en VM.

Til prototypen har jeg valgt at hoste databasen i en Docker-container, bl.a. fordi det gør det meget nemt og hurtigt at nulstille hele databasen når der under udviklingen opstår fejl 😊 Det har også givet mig erfaring med og forståelse for drift af Docker-containere, som er nyttig hvis løsningen skal skaleres op til den foreslåede implementering i produktion.

6.1 Perspektivering – fra prototype til fuld funktionalitet

Prototypen anvender en meget rudimentær proces til pre-processering og udskilning af loglinje bestanddele. Når LogSearcher skal behandle data fra forskellige systemer, skal denne proces udvides.

Det er nærliggende at udvide LogWatcher til at udstille LogParsers mulighed for at lede efter sæt af start/slut søgeord, og deres tidslige distribution, f.eks Top5/Bottom5.

Muligheden for at vise et grafisk overblik over den tidslige distribution af søge-hits er også en oplagt feature.

Da produktions-løsningen skal understøtte et flerbruger-scenarie, skal LogWatcher som Blazor Server udbygges med navngivne bruger-sessioner. LogParser skal instantieres individuelt for hver bruger-session, hvor hver session har eget diskret søgesæt.

Det vil også introducere behovet for at håndtere sikkerhedsaspekter, f. eks at integrere og håndtere brugeradgang mod DRs Active Directory.

Jakob Viggo Hansen
Nørrebro, 2022

6.2 Referencer

.NET. (7. 11 2022). Hentet fra Microsoft: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet

Automatic Memory Management. (7. 11 2022). Hentet fra Microsoft: https://learn.microsoft.com/en-us/dotnet/standard/automatic-memory-management

Big O introduction. (22. 3 2021). Hentet fra Towards Datascience: https://towardsdatascience.com/introduction-to-big-o-notation-820d2e25d3fd

C# vs Python 2. (7. 11 2022). Hentet fra litsLink: https://litslink.com/blog/csharp-vs-python-choosing-right-language-for-your-project

C# vs Python. (7. 11 2022). Hentet fra Hackr.io: https://hackr.io/blog/c-sharp-vs-python

CI. (7. 11 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Continuous_integration

Colossus. (7. 11 2022). Hentet fra Britannica: https://www.britannica.com/technology/Colossus-computer

Comparative Analysis of Horizontal And Vertical Scaling. (6 2016). Hentet fra Academia.edi: https://d1wqtxts1xzle7.cloudfront.net/52711973/IJSARTV2I63514-with-cover-page-v2.pdf?Expires=1668261640&Signature=fRPR9P9Aa47XxQD0tVflF9eApzeR4eSNZRhtU9u8j2hKPBgIlvpEHqup4HvwC61uoHVM7~FoogiglDN5vnFucZOz3FCFnHEDDoXhn2AgdyiRkTOZA8LTrC-QYMot9ZgVRdgoztBy1yNmP

Containers Versus Virtual Machine. (2. 9 2018). Hentet fra Springer Link: https://link.springer.com/chapter/10.1007/978-981-13-1501-5_12#citeas

Continuous Integration. (7. 11 2022). Hentet fra Atlassion.com: https://www.atlassian.com/continuous-delivery/continuous-integration

Doug Rosenberg, M. S. (2007). *Use Case Drive Object modelling with UML.* Apress.

Fowler, M. (2003). *UML Destilled: A Brief guide to Standard Object Modelling Language.* Addison-Wesley Professional.

HAL. (7. 11 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Hardware_abstraction

ISO 8601. (8. 11 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/ISO_8601

Java. (7. 11 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Java_(programming_language)

Kubernetes: Up and Running. (2022). Hentet fra Google Books: https://books.google.dk/books?hl=en&lr=&id=KeB-EAAAQBAJ&oi=fnd&pg=PT17&dq=kubernetes&ots=V9SXEOolU6&sig=jBTY4WiUsz-tDeVhPTa19Pp582o#v=onepage&q=kubernetes&f=false&redir_esc=y#v=onepage&q=kubernetes&f=false

Languages. (8. 11 2022). Hentet fra Memoryy Management: https://www.memorymanagement.org/mmref/lang.html

Microsoft. (2022). Hentet fra Azure Stack HCI: https://azure.microsoft.com/en-us/products/azure-stack/hci/

Multithreading in .NET. (17. 7 2017). Hentet fra jcomputers: http://www.jcomputers.us/vol13/jcp1304-07.pdf

ODBC. (03. 12 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Open_Database_Connectivity

OOP. (12. 11 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Object-oriented_programming

Public Broadcasting. (7. 11 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Public_broadcasting

Python code parallelization. (2016). Hentet fra Eso.org: http://www.eso.org/~jagonzal/ADASS-2016/P6-11.bkp/P6-11.pdf

Python for Science. (7. 11 2022). Hentet fra IBM: https://developer.ibm.com/blogs/use-python-for-scientific-research/

Reflective Programming. (4. 12 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Reflective_programming

REST. (3. 12 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Representational_state_transfer

SQL vs. NoSQL vs. NewSQL. (10 2016). Hentet fra COE: https://caeaccess.com/archives/volume6/number1/binani-2016-cae-652418.pdf

Stateless Vs. Stateful. (2019). Hentet fra Forbes: https://www.forbes.com/sites/forbestechcouncil/2019/09/18/stateless-vs-stateful-the-devils-in-the-details/

Survey on Horizontal and Vertical Scaling . (6 2016). Hentet fra Academia.edu: https://d1wqtxts1xzle7.cloudfront.net/52711973/IJSARTV2I63514-with-cover-page-v2.pdf?Expires=1668261499&Signature=A5-xcyvRc2PkRD7UbiO9ZdGyQeSr9xC2NxKy77eV3f3zXprb26J1u9J9l6w4F94SouZm9nBNVOW8s3ubmiEnPM2rS3W~qM10bWe5rAbQ6SPbFLhLwkCUhQL85ZLUZGps3jU1XUFqW9sE0

Top Programming Languages. (7. 11 2022). Hentet fra IEEE: https://spectrum.ieee.org/top-programming-languages-2022

TowardsDataScience. (9. 7 2020). Hentet fra Python multi-processing: https://towardsdatascience.com/python-multi-threading-vs-multi-processing-1e2561eb8a24

Unix-like. (8. 11 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Unix-like

Uvicorn. (4. 12 2022). Hentet fra Uvicorn: https://www.uvicorn.org/deployment/

What is Blazor. (7. 11 2022). Hentet fra Blazor University, Microsoft: https://blazor-university.com/overview/what-is-blazor/

WORA. (7. 11 2022). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Write_once,_run_anywhere