# 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 et systems nytteværdi i høj grad er afhængig af hvordan det driftes. Når der før eller siden opstår et problem i eller omkring det, er muligheden for spore fejlkæder afgørende for driftspersonalet.

## Problemstilling

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 ét system skyldes ofte fejl i systemer placeret tidligere i kæden. Systemer senere i kæden kan lide under det aktuelle systems fejl.

Alle disse systemer 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 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.

## Problemformulering

Jeg foreslår at udvikle en hjælpe-applikation, **LogWatcher**, som skal facilitere et dannelsen af et hurtigt overblik over flere systemers tilstand.

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

## LogWatcher

### Grafisk Brugerflade

LogWatcher 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.

### Database

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

### Søge-træer

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, som 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*

### LogWatcher mål-systemer

Det er potentielt relevant at indsamle log-data fra alle de DR-systemer som generer logs.

# TEKNOLOGI

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 platform skal den køre på? - Windows, \*Nix, OSX ?  
Skal den afvikles lokalt på brugerens klient eller central fra en server?   
Hvem håndterer daglig drift og problemløsning ifald 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 ønsker at brugere med både Win og OSX-klienter skal kunne anvende LogWatcher. 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.

## Udviklings-miljø

### Historisk tilbageblik på udviklingen

I de allerførste computersystemer som opstod i i 1940-50, var der ingen abstraktion mellem hvordan hardwaren var konstrueret, og hvor systemerne skulle programmeres. Colossus, som brød den tyske Kriegsmarine’s kodesystem, Enigma, var specialbygget til dette ene formål, og kunne aldrig udføre andre opgaver end denne ene: at bryde og dechifrere kodede meddelelser. (Colossus, 2022)  
  
Verdens første (elektroniske) ”general purpose” computer er generelt anerkendt til at være ENIAC, som blev sat i drift i 1945. ENIAC blev programmeret direkte i binær assembler. (ENIAC, 2022)  
  
Siden da er der opstået adskillige niveauer af abstraktion mellem hardwaren, og den software som vi skriver og anvender til at løse opgaver. CPU’erne har komplekse og omfattende instruksionssæt, som styresystemet kalder for at udstille højere funktioner til applikationerne. (HAL, 2022)  
  
Moderne softwareudvikling sker hovedsageligt på højt niveau af abstraktion. Her anses det som hovedregel for vigtigt, at kode kan afvikles på så mange mål-platforme som muligt. Dette var bl.a. visionen bag udviklingen af Java, ”Write Once, Run Anywhere” (WORA, 2022)

### 3 kandidater til udviklingssprog

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.

#### 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.

#### 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 udviklings miljøet .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).

#### 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-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)

### Opsummering

Hvert enkelt sprog har specifikke styrker og fordele. Python giver mulighed for meget elegant og effektiv rekursion som er vigtige for at bygge de node-baserede søge-træer som udgør 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)

## .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 MS .NET og C#.  
  
LogWatcher 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 MS .NET som udviklingsmiljø, skabes det bedste potentiale for at forankre værktøjet hos brugerne.

## Overvejelser om driftsmiljø

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

### VM's vs. containers

I moderne IT-drift er det ikke længere praksis at afvikle applikations-instanser på diskret hardware, dvs. èn fysisk server pr. applikation/system. 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**. Container-teknologier som Docker og Kubernetes bliver oftest anset som cloud-baserede. VM’s har hos DR indtil nu været baseret på OnPremise-hardware.  
  
Cloud-computing er et paradigme der faciliterer omfattende adgang til en central pulje af ressourcer. Det har traditionelt været forbundet med VMs, men containers er ved at vinde frem. Man kan betragte containers som letvægts VMs. (Containers Versus Virtual Machine, 2018)

#### 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 en proces eller applikation

#### Isolering som strategi

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

VM: Isolation of the machine  
Container: Isolation of the process

### Virtual Machines

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 produktionssystemer. 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.

”Problemet med Cloud instanser er at latency (svartider) er markant højere mellem lokal klient og cloud-server. Dvs. meget granuleret trafik med mange forespøgsler/svar akkumulerer forsinkelse. Det betyder at vi skal være opmærksomme på om cloud-servicen har meget trafik fra OnPremise systemer, fordi vi i så fald kan opleve nedsat performance.”

#### 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”

Der er planlagt et teknologiskifte, som indebærer at det eksisterende miljø udskiftes til fordel for MS Azure Stack HCI (Microsoft, 2022), og så kan ressourcer provisioneres mod både Cloud og OnPremise i samme driftsmiljø. Dermed bliver det praktiske skel mellem de to domæner gradvist udvisket.

### Containere

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. Vi bygger vores deploy-pipeline sammen med DevOps-teamets, dvs. vi får fordel af fælles infrastruktur. Vi vil tilbyde både OnPremise og Cloud-hosting”.  
  
Kubernetes er 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)

#### Foreslået arkitektur

”Jeg vil foreslå at du baserer LogWatcher på en .NET Core container, og en anden container til din database. Brug Docker for Desktop til udvikling og debugging.”

## Database - 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.

### 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.

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

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 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.  
  
NoSQL databaser kan under-grupperes i 4 strukturer:

* Kolonne-orienterede: data grupperes i kolonner, med frit antal rækker
* Key/Value: HashMap-baseret lookup
* Dokument-orienterede: egnet til JSON, XML ol.
* Graph-orienterede: repræsenterer data i node-baserede strukturer.

### Skalering

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)

# LOGWATCHER 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.

GUI & Data  
(C# & Blazor)

Søgemotor  
(Python)

eDSL

Bruger

LogWatcher

### eDSL – embedded Domain Specific Language

Ved at indbygge et eDSL i søgekomponenten (eDSL, 2022) ( Embedded Domain Specific Languages, 2022), kan jeg udstille de primære søge-værktøjer mod det omsluttende C#/.NET lag som udgør den brugervendte applikation.

### Blazor til GUI

Med .NET Core har Microsoft introduceret et web-orienteret lag til C#, som gør miljøet meget egnet til at udvikle websider. Den specifikke komponent 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.

## Drifts-miljø

Jeg vil anvende Docker for desktop. En .NET Core container, og en MongoDB-container.

## Data-repræsentation – fra kilde til database

Den fælles kilde-type for alle målsystemer vil være de ustrukturerede rå-data som logfilerne udgør. Dvs. LogWatcher 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.  
  
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*** |

*\** (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.

## Overvejelser om database og performance for LogWatcher

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 søgemoteren.

# SYSTEMUDVIKLING OG -ANALYSE

Jeg vil til denne facet af udviklingen benytte design-metoden ICONIX (Doug Rosenberg, 2007), sådan 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).

## Iconix begreber og processer

Første trin består i at samle en såkaldt **Funktionel Beskrivelse**, 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**. Denne bruges til at identificere objekter som er relevante at modellere i softwaren, og ikke mindst deres indbyrdes relation.  
  
Herefter kan der opbygges **Use Cases**, som i klart sprog og abstrakte detaljer beskriver hvordan en facet af brugerens interaktion med softwaren skal foregå og udspille sig. Hver distinkt interaktions-type kan beskrives i en diskret Use Case.

Det kan også være oplagt at udvikle skitser til hvordan softwarens **GUI** kan designes, gerne i en forsimplet form som lægger vægt på funktion snarere end form. Overvejelse om grænsefladen kan også påvirke strukturen af domæne-modellen. IConix-processen lægger op til at hele design-fasen er iterativ, hvor en opdagelse på ét niveau kan påvirke tidligere beslutninger.  
  
**Robustness Diagram** er et Iconix-specifict begreb som sigter efter at koble Use Cases til deres objekter. Vi søger at identificere domæne-modellens objekter i de analyserede Use Cases.