# Diagnose af IT Infrastrukturer baseret på eksplicitte afhængighedsrelationer

Silas Hansen & Morten Fachmann

Kongens Lyngby 2012

IMM-B.Eng-2012-23

# Indholdsfortegnelse

1 Indledning....5
2 Analyse....7
2.1 Formål....7
2.2 Problem....7
2.3 Løsning....8
2.4 Termer....8
2.5 Use cases....11
2.5.1 Aktørbeskrivelse....11
2.5.2 Konsekvensanalyse....11
2.5.3 Systemdokumentation....12
2.5.4 Statusrapportering....12
2.5.5 Systemadministration....13
2.6 Scenarier....14
2.6.1 Systemdokumentation....14
2.6.2 Konsekvensanalyse....14
2.6.3 Statusrapportering....14
2.7 Kravspecifikation....15
2.7.1 Ikke-funktionelle krav....16
2.8 Løsningsbeskrivelse....16
2.8.1 Webbaseret brugergrænseflade....16
2.8.2 SOA API grænseflade....22
2.8.3 Komponenter....22
2.8.4 Struktur på komponentafhængigheder....24
2.8.5 Evaluering af komponenttilstand....25
2.8.6 Advarselstilstand....28
2.9 Mediator....31
2.9.1 Mediator brugergrænseflade....32
2.10 Data Design model....33
2.11 Analysekonklusion....33
3 Teknologier....34
3.1 Teknologier og programmer der behøver yderligere introduktion....34

3.1.1 ASP.NET MVC....34
3.1.2 WCF – Windows Communication Foundation....35
3.1.3 Entity Framework....35
3.1.4 Adobe Flash Builder....35
3.1.5 Kalileo Diagrammer....36
3.1.6 SignalR....36
3.1.7 ANTLR....37
4 Design....38
4.1 Teknologieksperimenter....38
4.2 System model....39
4.3 HTML brugergrænseflade....39
4.4 Model Change Management....40
4.5 SOA API Interface....41
4.6 Diagrambaseret visualisering....42
4.7 Live opdatering af data i HTML grænseflade....45
4.8 Tekst og HTML baseret editering....46
4.9 Model CRUD scaffolding....46
4.10 Evaluering af komponenttilstande....47
4.11 Parsning af boolske udtryk....47
4.12 Grafisk visning af komponenttilstande....49
4.13 SOA API Grænseflade....49
4.14 Konsekvensanalyse....49
4.15 Mediator....51
4.15.1 Datamodel for mediator API konfiguration....51
4.15.2 Datamodel for mediator komponent konfiguration....51
4.15.3 Logik....53
5 Implementering....55
5.1 Parsning af boolsk logik med ANTLR....55
5.2 Flashbaseret diagramkomponent....57
5.3 Visningsalgoritmer....59
5.4 Live data med SignalR....61
5.5 Webservice med WCF....61

5.6 Databaseadgang med Entity Framework ....62
5.7 Mediator kommunikation ....64
5.7.1 PRTG ....64
5.7.2 Infrastructure Diagnosis ....65
5.8 Mediator GUI og model binding ....66
6 Test ....69
6.1 Konsekvensanalyse ....69
6.2 Systemdokumentation ....70
6.3 Statusrapportering ....70
6.4 Konfiguration ....70
6.5 Import af live data ....71
6.6 Test konklusion ....72
7 Konklusion ....73
Appendiks ....74
A. Database model ....74
B. Håndbog ....75
Opsætning ....75
Funktionalitet ....89
Mediator ....95

# 1 Indledning

Vi har igennem to år været ansat i en mindre it-afdeling i et firma som primært beskæftiger sig med drift og hosting af arbejdsløshedskasser. Det er vores vigtigste opgave at sørge for at vores kunders it-systemer fungerer og er tilgængelige døgnet rundt. Vores problem er at it-systemets driftsikkerhed, er truet af uforudsete konsekvenser som indtræffer ved ændringer på it-systemet. Vi har i dette projekt identificeret to problemer, som vi anser som værende de primære trusler imod driftsikkerheden.

Det første problem er at vi ikke er i stand til at forudsige de konsekvenser der opstår, når vi ændrer på vores it-system. Problemet møder vi typisk i forbindelse med ændringer på komponenter i it-systemet. Disse komponenter kan være både software og hardware baserede. Et eksempel på en hardware komponent kunne være en server eller en router.. Problemet opstår typisk som følge af en planlagt ændring på en eller flere komponenter. Ændringen fører efterfølgende uhensigtsmæssige konsekvenser med sig, som skyldes at andre tilknyttede komponenter ikke længere, er i stand til at udføre hele eller dele af deres funktion. Årsagen til problemet er, at der i firmaet ikke er tilstrækkeligt overblik over it-systemets komponenter og afhængigheder, til at kunne forudsige ændringens konsekvens. Dette skyldes manglende dokumentation, men i ligeså høj grad, at den dokumentation der haves, ikke præsenteres på en overskuelig måde.

Det andet problem er at vi ikke er i stand til at skabe os det nødvendige overblik over it-systemets komponenter og dets indbyrdes afhængigheder. Dette problem opstår når der er konstateret fejl på systemet. Definition af fejl omfatter kritiske tilstande såsom fysisk nedbrud, software nedbrud, eller uventet adfærd fra en komponent i systemet. Fejl indrapporteres på to måder, enten via kunden eller via et automatisk overvågningsværktøj. Kunden beskriver typisk fejlen på højt niveau, forstået på den måde at de beskriver en funktionalitet som ikke virker, hvorefter det er op til it-afdelingen at finde ud af hvad der præcist er galt. Det automatiske overvågnings-værktøjet fungerer anderledes, ved at det typisk beskriver fejlen på et meget lavt niveau, forstået på den måde at det er en meget specifik komponent der beskrives, hvorefter det er it-afdelingens opgave at finde årsagen til fejlen, men i ligeså høj grad hvilke afledte konsekvenser det har for it-systemet og for kunden. For at kunne indfri it-afdelingens ønske om hurtigt at kunne identificere årsager og afledte konsekvenser, er det nødvendigt at medarbejderne har et indgående og detaljeret kendskab til it-systemets komponenter og afhængigheder. En afhængighed eksisterer mellem to komponenter når den ene komponents funktionalitet er påkrævet, for at den anden komponent kan udføre sin funktion.

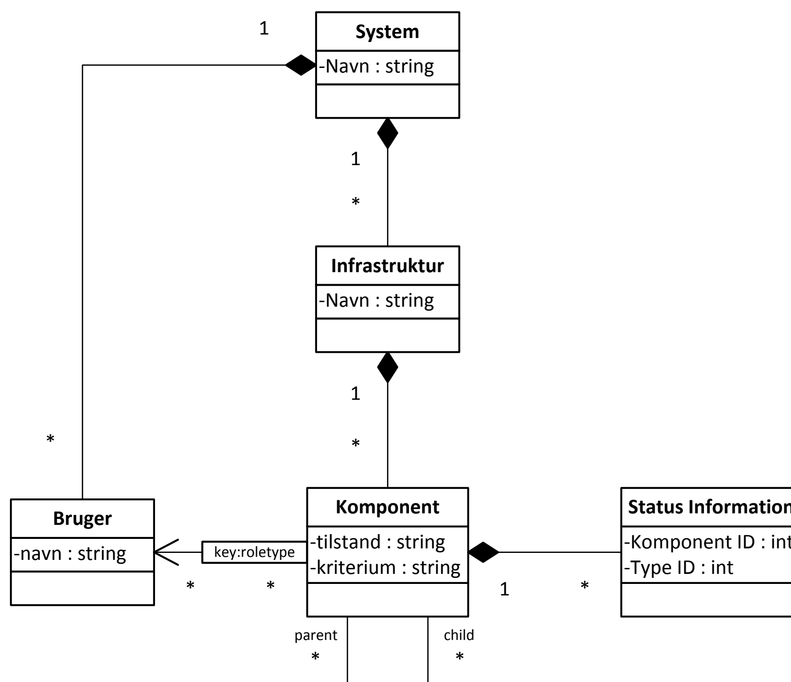
Begge problemer skyldes i høj grad mangel på brugbar dokumentation. Den nuværende dokumentation er på nogle områder mangelfuld, og på andre områder meget omfattende og detaljeret, men ingen af delene løser effektivt problemerne. Med dette projekt introducerer vi tre nye koncepter, som er vores bud på en løsning på de nævnte problemer. Koncepterne har vi kaldt *Konsekvensanalyse*, *Systemdokumentation* og *Statusrapportering*.

***Konsekvensanalyse*** er et værktøj der kan danne en rapport, som beskriver hvordan et fiktivt scenarium vil påvirke it-systemets komponenter.

***Systemdokumentation*** er et værktøj som gør det muligt at danne sig et overblik over it-systemets sammensætning og tilstand, ved hjælp af en grafisk visualisering af komponenter og afhængigheder, samt en tekstbaseret visning af samme information.

***Statusrapportering*** gør det muligt at danne sig et øjebliksbillede af it-systemets generelle tilstand. Informationen i øjebliksbillede vises som aggregerede kategorier af funktionalitet, som udgør et eller flere kerneområder i it-systemet.

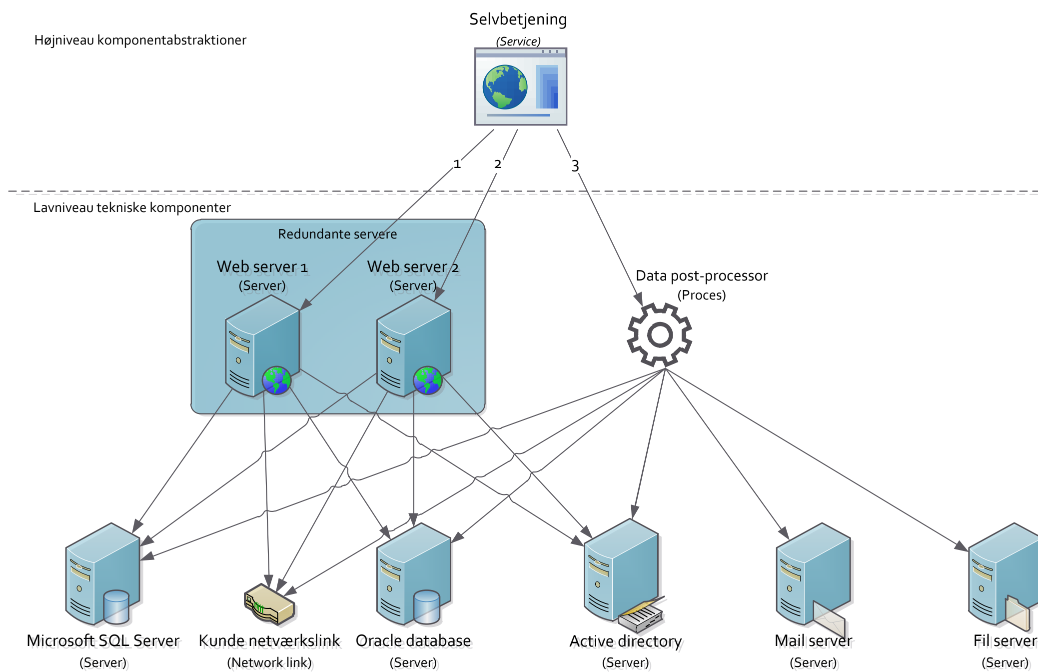
For at give læseren en lidt mere konkret, men dog simpel beskrivelse af løsningen tidligt i projektet, så bringer vi nedenfor i Figur 1 domænemodellen. Komponent-strukturen er statisk, mens *Status Information* opbevarer dynamiske tilstands-informationer. En mindre undtagelse er at *Komponent* klassen dog indeholder den øjeblikkelige tilstand på komponenten.




Figur 1 – Domænemodel

# 2 Analyse

I dette kapitel beskriver vi vores problem og den tilhørende løsning. Vi har i dette projekt taget udgangspunkt i problemer, som vi har stiftet bekendtskab med på vores arbejdsplads. For at læseren lettere kan sætte sig ind i projektet, vil vi herunder beskrive de primære problemer som vi står overfor i dag, og samtidig introducere vores løsningsbeskrivelse. Et typisk eksempel på en del af firmaets IT-infrastruktur ses i Figur 2.




Figur 2 - Et eksempel på en service (Selvbetjening) og tilhørende afhængigheder som optræder i firmaets miljø

## 2.1 Formål

Vores overordnede mål er at forbedre driftsikkerheden på firmaets it-systemer, gennem udvikling af en række værktøjer som skal hjælpe it-afdelingen med at løse de, i indledningen, nævnte problemer.

## 2.2 Problem

Vores problem er at driftsikkerheden på firmaets it-systemer, er truet af uforudsete konsekvenser som indtræffer ved ændringer på it-systemerne. Disse ændringer kan være udført af os selv med overlæg eller af eksterne begivenheder, som eksempelvis strømafbrydelse.

## 2.3 Løsning

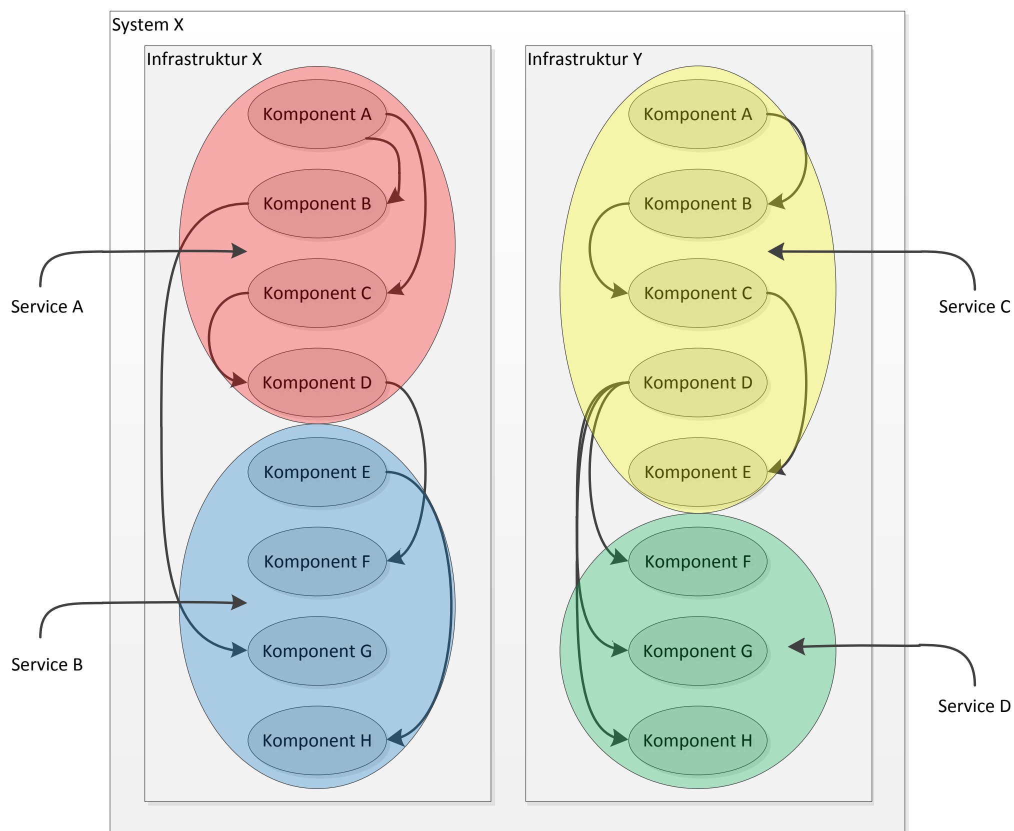
Løsningen omfatter udviklingen af en webside, der skal kunne tilgås af alle typer brugere i en it-afdeling, da alle kan have en interesse i hele eller dele af løsningen. På websiden skal det være muligt at tilgå de tre nøglefunktionaliteter: *Konsekvensanalyse*, *Systemdokumentation* og *Statusrapportering*. Vi vil opbygge en løsning der fokuserer på følgende hovedpunkter:

- Forudsigelse af fejl og medfølgende konsekvenser.
- Hurtig diagnosticering af fejl på it-systemet.

Vi navngiver denne løsning **Infrastructure Diagnosis**.

## 2.4 Termer

Vi vil i dette projekt benytte os af termer, som vi tillægger en specifik betydning. Dette gør vi i mangel af bedre beskrivende ord. Nogle af termerne er allerede tidligere i rapporten beskrevet og benyttet, men vi beskriver dem her alligevel, da afsnittet så kan benyttes som opslagsværk. For at give læseren en hurtig introduktion til de termer vi benytter, har vi lavet Figur 3 hvor de mest grundlæggende termer indgår og umiddelbart herefter, beskrevet de enkelte termer i detaljer.




Figur 3 – Termernes indbyrdes forhold

Vi benytter os i projektet af følgende termer:

**System**
Et system er den overordnede entitet, som kan bestå af en eller infrastrukturer. Systemet vil typisk benyttes som enheden der repræsenterer en specifik virksomhed.

**Infrastruktur**
En infrastruktur er en entitet som via komponenter og deres indbyrdes afhængigheder, er i stand til at repræsentere et it-system. En infrastruktur kan også betragtes som et sub-*system* som typisk repræsenterer en afdeling eller anden afgrænset gruppering i virksomheden.

**Komponent**
En komponent repræsenterer en entitet i systemet. Der er ingen regler for hvad en komponent kan repræsentere, men i vores løsning vil det typisk være hardware eller software. En server eller en switch er eksempler på hardware komponenter, hvor en applikation eller proces er eksempler på software komponenter

**Afhængighed**
En afhængighed beskriver en association mellem to komponenter. Det kan f.eks. være en webserver der lagrer sine data på en SQL-server. I dette tilfælde vil webserveren være afhængig af SQL-serveren. En afhængighed kan være mere kompleks end blot en række associationer. I Figur 2 er illustreret et eksempel hvor en afhængighed som er mere kompleks en blot en række associationer. Figuren illustrerer at *selvbetjening* har to associationer markeret *1* og *2,* som ikke begge kræves at være funktionsdygtige samtidig. Denne situation kan udtrykkes ved hjælp af et kriterium som beskriver forholdet mellem association *1* og *2*.

**Service**
En service komponent er en abstrakt definition af flere komponenter. Den dækker over en eller flere komponenter, som samlet udgør et stykke funktionalitet på højt niveau. Der vil typisk være tale om funktionalitet på et niveau som firmaets kunder kan forstå. Da store dele af vores projekt omhandler services i større eller mindre grad, har vi illustreret konceptet i Figur 2 på side 7.

Hvis man kigger på Figur 2 ses et eksempel på en *Service* som i dette tilfælde er en selvbetjenings-platform som vores firma udvikler og vedligeholder for vores kunder. Komponenten *Selvbetjening* er i det her tilfælde defineret som en *Service*, fordi kundens opfattelse af begrebet *Selvbetjening,* omfatter både efterbehandling (*Data post-processor)* af data, samt indsamling af data via web platformen (*Web server*). På den måde samles kundens forståelse af funktionaliteten under en fælles definition.

**Komponenttilstand**
En komponenttilstand er i vores løsning en indikation af komponentens helbred i forhold til at udføre sin forventede funktion. Der opereres med tilstandene: *Up, Down* og *Warning*.

**Bruger**

En bruger er en person som har en association til en eller flere *komponenter*. En association kan være kvalificeret idet en bruger kan have forskellige forhold til en komponent. I vores system vil der opereres med kvalifikationerne *Owner* og *Interestee*.

## 2.5 Use cases

For at få et mere detaljeret kendskab til vore krav til løsningen, har vi udarbejdet fire use cases. De første tre use cases beskriver nøglefunktionalitet i løsningen, og den sidste use case beskriver initial opsætning og løbende administration af løsningen.

### 2.5.1 Aktørbeskrivelse

Vi arbejder med følgende fire aktørtyper i vores projekt:

| Aktør | Beskrivelse |
|---|---|
| **IT-tekniker** | En it-tekniker er en medarbejder med større teknisk indsigt i software og/eller hardware. |
| **Supporter** | En supporter er en medarbejder med mindre teknisk indsigt i software og/eller hardware. |
| **Leder** | En leder, er en overordnet medarbejder med ansvar for firmaets interesser. |
| **IT-medarbejder** | En IT-medarbejder er en fællesbetegnelse for IT-teknikere, supportere og ledere. |

### 2.5.2 Konsekvensanalyse

Denne use case beskriver brugen af værktøjet konsekvensanalyse.

| Aktør | **IT-tekniker** |
|---|---|
| Overordnet mål | Får kendskab til konsekvenserne for et system, ved en eller flere komponenters fravær. |
| Forudsætninger | Programmet (Infrastructure Diagnosis) er konfigureret så det afspejler det system som IT-teknikeren benytter sig af. |
| Mål: | 1) At gøre IT-teknikeren i stand til at markere en eller flere komponenter og sætte deres tilstand til *Up* eller *Down*.<br>2) Konsekvensanalysen bliver præsenteret for IT-teknikeren som en liste af påvirkede komponenter. |
| Flow: | 1) IT-teknikeren tilgår den komponent, som status ønskes ændret for.<br>2) IT-teknikeren ændrer status til *Up* eller *Down*.<br>3) IT-teknikeren bliver præsenteret for en liste af komponenter som er påvirket af ændringen. |

### 2.5.3 Systemdokumentation

Denne use case beskriver brugen af værktøjet systemdokumentation.

| Aktør | IT-tekniker |
|---|---|
| **Overordnet mål** | Får kendskab til en eller flere komponenter i et forud valgt system. |
| **Forudsætning** | Programmet (Infrastructure Diagnosis) er konfigureret så det afspejler det system som IT-teknikeren benytter sig af, og alle komponenter er blevet beskrevet. |
| **Mål** | 1) Det er muligt for IT-teknikeren indhente information omkring en komponent. |
| **Flow** | 1) IT-teknikeren tilgår den komponent der ønskes information omkring.<br>2) IT-teknikeren indhenter detaljer omkring komponenten ved trykke på den. |

### 2.5.4 Statusrapportering

Denne use case beskriver brugen af værktøjet statusrapportering.

| Aktør | IT-medarbejder |
|---|---|
| **Overordnet mål** | Får kendskab til et IT systems overordnede tilstand, med henblik på information til kunde. |
| **Forudsætninger** | Programmet (Infrastructure Diagnosis) er konfigureret så det afspejler det system som IT-teknikeren benytter sig af, og alle komponenter er blevet beskrevet. |
| **Mål** | 1) At gøre det muligt for en IT-medarbejder hurtigt at danne sig et overblik, over hvilke services der ikke er funktionelle i et it-system.<br>2) At gøre det muligt for en IT-medarbejder at indhente information, til brug i forbindelse med status-meldinger til kunden. |
| **Flow** | 1) IT-medarbejderen tilgår service oversigten.<br>2) IT-medarbejderen tilgår en side, hvor services hvis tilstand er *Error* eller *Warning* er listet.<br>3) IT-medarbejderen tilgår detaljer på de påvirkede services, hvoraf det det blandt andet fremgår hvem der ejer servicen, og dermed skal |

| | |
|---|---|
| | kontaktes vedrørende denne komponent. |
| **Bemærkning** | Informationen som IT-medarbejderen ved hjælp den denne use case kommer i besiddelse af, benyttes typisk til at informere de involverede kunder. |

### 2.5.5 Systemadministration

Denne use case beskriver hvordan vores løsning administreres.

| **Aktør** | **IT-tekniker** |
|---|---|
| **Overordnet mål** | Får mulighed for at opsætte og administrere et eller flere systemer i Infrastructure Diagnosis |
| **Forudsætninger** | Programmet (Infrastructure Diagnosis) er ikke konfigureret. |
| **Bemærkninger** | Da det vil være meget omfattende at beskrive alle de forskellige situationer en IT-tekniker kan befinde sig i, i forbindelse med administration af løsning, vil flowet kun beskrive oprettelse af et *system*. Alle de resterende oprettelser, ændringer og sletninger fungerer efter samme princip. |
| **Mål** | 1) Det er muligt at oprette systemer, infrastrukturer, komponenter, afhængigheder, kriterier og brugere.<br>2) Det er muligt at ændre systemer, infrastrukturer, komponenter, afhængigheder, kriterier og brugere.<br>3) Det er muligt at slette systemer, infrastrukturer, komponenter, afhængigheder, kriterier og brugere. |
| **Flow** | 1) IT- teknikeren tilgår forsiden<br>2) IT- teknikeren trykker på knappen "Create New System"<br>3) IT- teknikeren bliver nu mødt af en side som udbeder sig de krævede information for at et system kan oprettes.<br>4) IT- teknikeren indtaster den af siden krævede information<br>5) IT- teknikeren trykker på knappen "Create System" |

## 2.6 Scenarier

Dette afsnits scenarier beskriver specifikke brugsmønstre i forbindelse med vores løsning. Scenarierne beskriver med vilje de samme arbejdsgange som vores use cases tidligere har beskrevet, da vi ønsker funktionaliteten i vores løsning, beskrevet bedst muligt.

### 2.6.1 Systemdokumentation

Dette scenarie beskriver en situation hvor en medarbejder ved firmaet, drager nytte af systemdokumentationen i Infrastructure Diagnosis.

Det er midt i juli måned og firmaet er på grund af sommerferie bemandet med et minimum af medarbejdere. Martin er, som eneste medarbejder, på arbejde for infrastrukturafdelingen. Han bliver, af overvågningssystemet, gjort opmærksom på at en kundes hjemmeside ikke længere er tilgængelig. Kundens hjemmeside er ikke en del af Martins normale ansvarsområde, og han kender kun perifært til dens eksistens. Martin gør brug af Infrastructure Diagnosis og af serviceoversigten fremgår det, at kundens hjemmeside er gået ned. Martin kan i den detaljerede visning se, at det blot drejer sig om at IIS processen er gået ned. Han trykker på processen, og af den detaljerede proces visning, indhenter han oplysninger som han efterfølgende benytter til at rette fejlen.

### 2.6.2 Konsekvensanalyse

Konsekvensanalyse benyttes af firmaet til at forudsige fejl på deres systemer. Følgende scenarie beskriver en fiktiv men realistisk situation, hvor konsekvens-analyse benyttes.

Som en del af infrastrukturafdelingens daglige drift, skal en SQL-server genstartes på grund af sikkerhedsopdateringer. Infrastrukturafdelingen har ikke detaljeret kendskab til de andre afdelingers brug af SQL-serveren, men har kendskab til at den benyttes i mange sammenhænge. For at finde ud af hvilke services der bliver påvirket af genstarten, benytter infrastrukturafdelingen sig af Infrastructure Diagnosis. Her kan infrastrukturafdelingen teste et scenarie hvor SQL serveren sættes offline og efterfølgende få oplyst hvilke services der påvirkes af ændringen. Infrastruktur-afdelingen informerer nu de involverede afdelinger omkring genstarten af SQL-serveren, sådan at de nu har mulighed for at forberede sig på hændelsen.

### 2.6.3 Statusrapportering

Firmaet benytter statusrapportering til at informere deres kunder i forbindelse med nedbrud. Følgende scenarie beskriver en fiktiv men realistisk situation, hvor statusrapportering benyttes.

En del af firmaets storage løsning er brudt ned, hvilket har resulteret i at et større antal servere og services er nede. Infrastrukturafdelingen er i gang med at løse

problemet ved at starte de ramte servere op på en anden storage enhed. Dette betyder at de ramte dele, en efter en, bliver operative igen. Undervejs i processen benytter en leder sig af Infrastructure Diagnosis mulighed for at få et overblik over de ramte services. Her fremgår det for lederen hvilke dele der er nede og hvorfor. Lederen kan derfor med jævne mellemrum videreformidle driftsstatus til kunden, uden at skulle forhøre sig direkte ved infrastrukturafdelingen.

## 2.7 Kravspecifikation

Vi beskriver først vores overordnede krav til løsningen, som består af følgende tre punkter:

- Konsekvensanalyse
- Systemdokumentation
- Konfiguration

Konsekvensanalyser skal gøre os i stand til at kunne forudsige fejl på it-systemet. Det skal være muligt at analysere et negativt scenarie for et it-system og på den måde få kendskab til de negative konsekvenser ved en eller flere komponenters *Down* tilstand.

Systemdokumentation skal gøre os i stand til at visualisere IT-systemers infrastrukturer på en mere overskuelig måde end vi er vant til fra traditionelle overvågningsværktøjer, der typisk anvender lister og tabeller. Vi ønsker at skille os ud på overskuelighed, ved at man kan tage udgangspunkt i en komponent og derfra gøre det muligt at se andre komponenter der er associationer til.

Sidste punkt skal gøre det muligt at administrere den kommende løsning. Dette punkt dækker over alle de praktiske funktioner, såsom opret, ændre og slette. Dette er trivielle funktioner, men er så absolut en vigtig del af løsningen.

Vi introducerer yderligere to punkter, som ikke direkte øger driftsikkerheden, men som dog vil kunne forbedre hverdagen for både vores kunder og os.

- Statusrapportering
- Import af live data fra eksterne kilder

**Bemærk**: De to punkter er knyttet tæt sammen, men det skal pointeres at funktionaliteten *statusrapportering* vil kunne fungere uden import af live data fra eksterne kilder, og statusrapporteringen vil i dette tilfælde kun rapportere på manuelle input. Det er yderligere værd at bemærke at når en tilstand for en komponent sættes, så vil det øjeblikkeligt slå igennem i resten af løsningen, lige meget om den er sat manuelt eller via live opdatering.

Meningen med statusrapportering er det at skal være let at få et overblik over et it-systems helbred. I stedet for traditionelt at vise alle de komponenter der er fejlbehæftede, ønsker vi kun at de fejlbehæftede services skal figurere. Det vil lette

udmeldingen til kunder, og det vil måske endda være muligt for kunden selv, at benytte sig af denne funktionalitet.

Det sidste punkt, Import af live data fra eksterne kilder, er en ekstrafunktionalitet til det tidligere punkt Systemdokumentation. Det skal være muligt at indhente live data fra tredjeparts overvågningsværktøjer, hvilket så kan bruges til at live opdatere systemdokumentation. Dette vil kræve at vi stiller en mulighed for at kommunikere med systemet til rådighed for eksterne systemer. Til netop den opgave vil det være oplagt at vælge et SOA baseret API, da SOA i sin natur håndterer live kommunikation

### 2.7.1 Ikke-funktionelle krav

Vi har følgende ikke-funktionelle krav til vores løsning.

- Fokus på ikke fungerende komponenter og services.
- Nem og intuitiv brugergrænseflade.
- Simpel opbygning, kun den vigtigste information vises.
- Løsningen skal kunne håndtere optil 250 forskellige *komponenter*.

## 2.8 Løsningsbeskrivelse

Vores løsning vil overordnet komme til at indeholde følgende dele:

**Webside**
En webside, som vil være frontend ud imod brugeren af løsningen. Funktionaliteten som vil kunne tilgås fra websiden vil være følgende: *Systemdokumentation*, *konsekvensanalyse* og *statusrapportering*.

**Database**
En database, hvor alle informationer vil blive lagret.

**API**
Et API som skal stå for kommunikationen med eksterne aktører.

**Mediator**
Yderligere udvikles en mediator som et proof-of-concept på, at kommunikation mellem overvågningsværktøjer og vores løsning, Infrastructure Diagnosis, er mulig og anvendelig.

### 2.8.1 Webbaseret brugergrænseflade

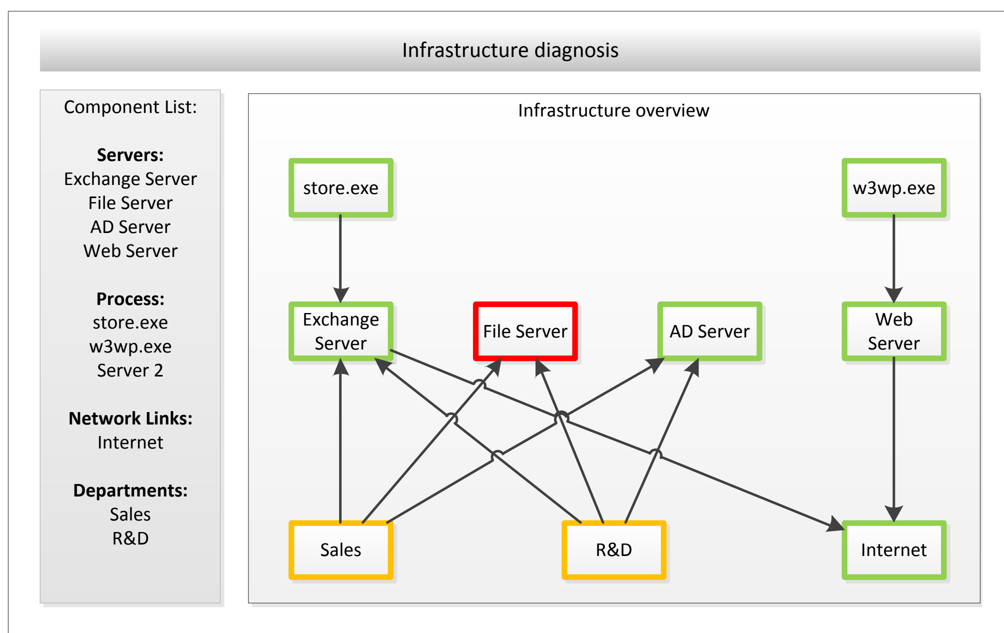
Systemets grænseflade implementeres som en webbaseret applikation og vil være slutbrugerens primære måde at interagere med systemet på.

#### 2.8.1.1 *Systemdokumentation*

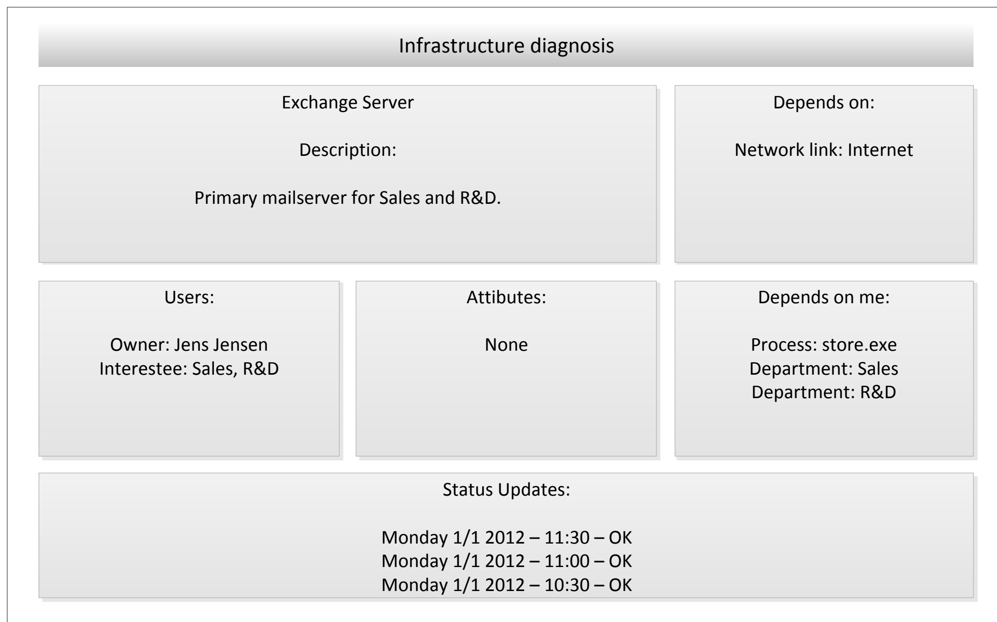
Et ønske om dokumentation af komponenter og afhængigheder er en af de grundliggende koncepter i vores løsning. Hvis systemet kender til alle komponenter i firmaets infrastruktur og samtidig har information om hvilke afhængigheder der er

mellem komponenterne, så giver det mulighed for at løsningen kan give brugerne af systemet, en række muligheder såsom visning af aggregerede helbredsdata, grafisk repræsentation af afhængigheder og specifikke overbliksbilleder, der kun handler om enkelte komponenter. Det har også den fordel at brugere i virksomheden kan bruge systemet som opslagsværk, når der ønskes indblik i specifikke sammenhænge i infrastrukturen. Man kan vel næsten tillade sig at sige, at ethvert firma bør have den slags information dokumenteret et eller andet sted. Vi giver her et bud på en mulig løsning der ikke er baseret på gammeldags tekstdokumenter. Dette stiller selvfølgelig nogle krav til disciplin og processer omkring brugen af systemet, men det må være et område der ligger uden for omfanget af denne rapport.

Billede 1 (s.17) viser en tegning over hvordan vi forestiller os at system-dokumentation kunne se ud. I venstre side af tegningen er der en menu som lister alle komponenter, der er knyttet til den valgte infrastruktur. I højre side af tegningen er der en graf som illustrerer infrastrukturens komponenter og hvor pilene imellem komponenterne repræsenterer afhængigheder. Det er muligt for brugeren at trykke på komponenterne i menuen, og på den måde få præsenteret en detaljeret visning af komponenterne. Med Billede 2 har vi illustreret hvordan vi forestiller os at det kunne se ud. Detaljerne vil omfatte beskrivelser af komponentens afhængigheder, involverede brugere og tilstandsopdateringer.

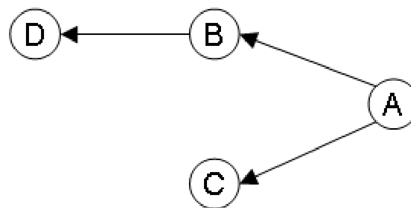



Billede 1 – Systemdokumentation, infrastruktur oversigt




Billede 2 – Systemdokumentation, komponent i detaljeret visning

Det giver mening at forsøge at repræsentere afhængighederne i computernetværket ved hjælp af en grafisk fremstilling fordi vi her har mulighed for at lege med visuelle indikatorer såsom størrelser, farver og retningspile. Det er klart en fordel frem for en tekstbaseret fremstilling. Se Figur 4 for et simpelt eksempel på en grafisk repræsentation af afhængigheder. I den nævnte figur er **A afhængig af B og C** og **B er afhængig af D**.




Figur 4 – Eksempel på visuel repræsentation af afhængigheder

Man kan også bruge farver til at præsentere den status som komponenten har, tidligere illustreret på Billede 1 (s. 17)

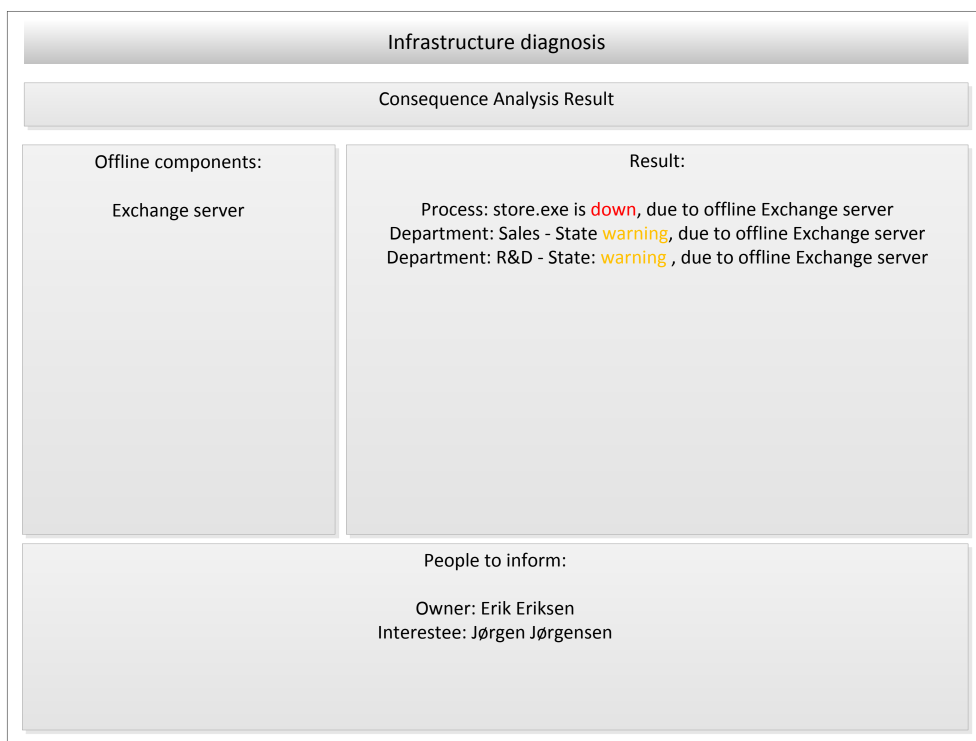
#### 2.8.1.2 *Konsekvensanalyse*

Muligheden for at få vist hvilke konsekvenser en ændring i systemet vil medføre, vil være et yderst brugbart værktøj for en bruger som ikke nødvendigvis har det fulde kendskab til de afhængigheder der findes i systemet. Her kan en bruger, inden faktisk udførelse, afprøve forskellige scenarier og kortlægge konsekvenserne for resten af it-systemet. I en typisk virksomhed som vores, vil komponenterne i IT-infrastrukturen have en eller flere interesserede eller ansvarlige personer. Ud over at informere om hvilke dele af systemet der er berørt, vil konsekvensanalysen også synliggøre hvilke ansvarlige personer der bør informeres, hvis det tænkte scenearie blev realiseret. For at kunne give et retvisende billede, kræver at det modelerede system afspejler den virkelige verden.

Billede 3 (s. 19) er en tegning der viser hvordan vi forestiller os at en konsekvens-analyse opstilles. I venstre side er der en liste over komponenter, de komponenter der markeres, er de komponenter der vil blive simuleret som havende tilstanden *Down*. Billede 4 (s. 20) viser hvordan vi forestiller os at resultatet af en konsekvens analyse, kan se ud. I venstre side er der en opremsning af de valgte komponenter med tilstanden *Down* og i højre side en liste hvor konsekvenserne for systemet fremgår. I bunden af billedet listes de personer som har en interesse i at kende til konsekvenserne, hvis scenariet senere skulle realiseres.

Infrastructure diagnosis
Consequence Analysis
Component List:
**Servers:**
☑ Exchange Server
☐ File Server
☐ AD Server
☐ Web Server
**Process:**
☐ store.exe
☐ w3wp.exe
☐ Server 2
**Network Links:**
☐ Internet
**Departments:**
☐ Sales
☐ R&D
Generate Report
Guideline:
Please select those components which, in the analysis, will be simulated as offline

**Billede 3 – Konsekvensanalyse, konfiguration**




**Billede 4 – Konsekvensanalyse, resultat**

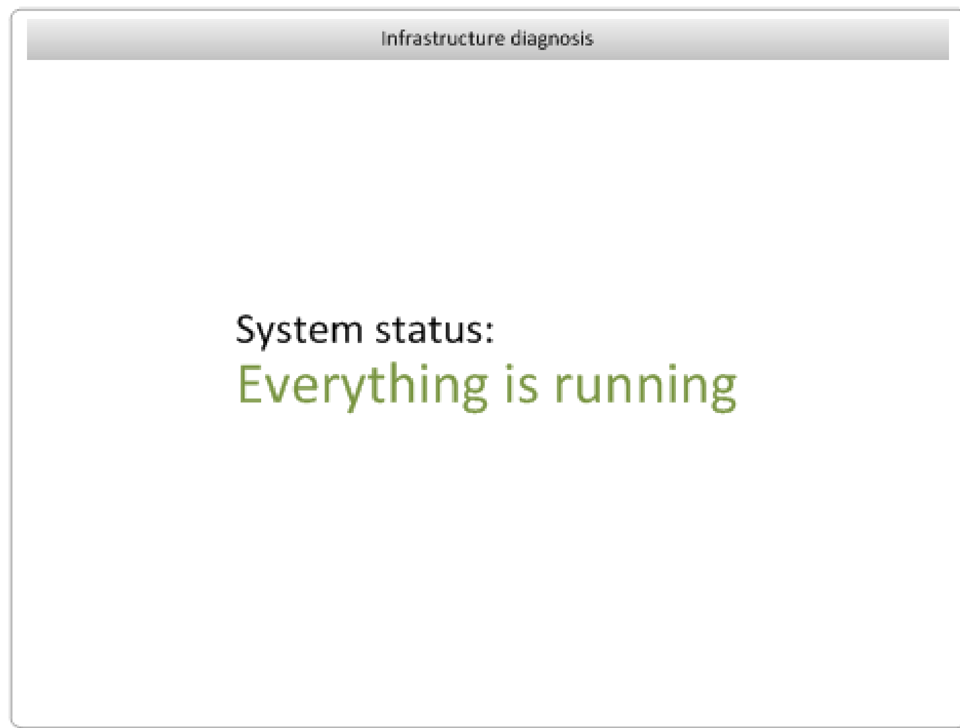
### 2.8.1.3 *Statusrapportering*

For at de ansatte i firmaet kan få ro til at udføre deres arbejdsopgaver, er det vigtigt at den enkelte medarbejder på en let måde, kan få et overblik over hvilke komponenter der på et givent tidspunkt ikke fungerer efter hensigten. (Se Figur 5 og Figur 6) Overblikket skal være så simpelt som muligt og vil derfor som udgangspunkt kun indeholde informationer om de services der er fejl på. Hvis der ikke er nogen fejl, så vil der blot gives besked om at systemet kører. Sådan et overblik vil kunne bruges af alle firmaets roller og vil have en værdi på forskellige niveauer:

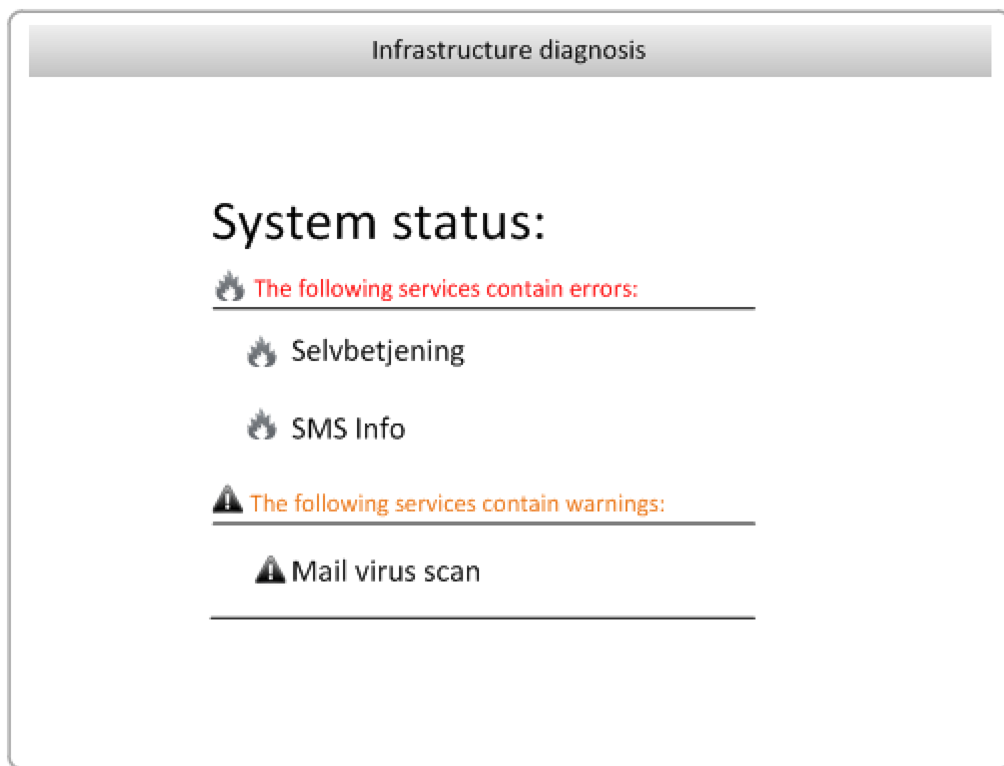
**Lederen** vil kunne danne sig et overblik over hvilke services der ikke fungerer, altså kun orientere sig om de komponenter der har betydning i en større forretnings-mæssig sammenhæng.

**Teknikeren** vil kunne danne sig et overblik over hvilke specifikke komponenter i infrastrukturen der skal problemsøges, sådan at lederens overblik igen bliver fejlfrit.

**Supporteren** vil kunne orientere sig omkring systemets tilstand ved kundehenvendelser og derved blive i stand til at give kunden et mere konkret og brugbart svar end før. Supporteren vil også være bedre klædt på til at viderebringe en sag til teknikeren nu hvor årsagen til problemet er mere tydelig.




Figur 5 – Eksempel på at der ingen fejl er på systemet




Figur 6 – Eksempel på at systemet har fejl og advarsler

### 2.8.2 SOA API grænseflade

API grænsefladen vil kunne bruges af eksterne systemer som ønsker at interagere med løsningen. API'et vil tilbyde muligheden for at hente metadata omkring systemets komponenter, systemer og infrastrukturer, samt muligheden for at fortælle systemet om ændringer på specifikke komponenter, således at et eksternt system kan levere online data til systemet automatisk. Når data leveres vil det være muligt at fortælle systemet hvilken type data der er tale om, således at der kan indrapporteres forskellige typer af data som kan bruges ved udtræk af data til statistik eller komponentdiagnose.

#### 2.8.2.1 Udtræk af komponentdata fra systemet

Udtræk af model oplysninger via API fra vores løsning er et krav, fordi eksterne systemer skal have mulighed for at danne en relation fra det eksterne system til vores løsning, uden menneskelig indblanding. For at eksterne systemer skal kunne foretage en korrekt analyse af systemet udefra, skal følgende muligheder være til stede i API'et:

- Udtræk af liste over alle *Systemer*
- Udtræk af liste over alle *Infrastrukturer* på et givent *System*
- Udtræk af liste over alle *Komponenter* på en given *Infrastruktur*
- Udtræk af liste over datatyper

#### 2.8.2.2 Indsendelse af komponentdata til systemet

Indsendelse af data via API er den del af systemet der gør at øvrige systemer har mulighed for at levere data. Det er derfor vigtigt at systemet stiller den mulighed til rådighed. Hertil er der brug for følgende funktionalitet:

- Indsendelse af data for en given *Komponent*

Systemet kan sættes op sådan at der kan sendes et vilkårligt antal forskellige datatyper ind omkring hver komponent. En datatype kunne være information om komponentens status (Up/Down/Warning), information om generelle system statistikker såsom hukommelses forbrug, harddisk forbrug osv. Internt vil nogle af disse typer være faste typer som er defineret af systemet fordi de indgår i systemets logik. Andre typer vil være brugerdefinerede og indeholde overfladiske data som kun bruges til at gemme historik på komponentens tilstand, i forbindelse med senere diagnosticering af fejl.

### 2.8.3 Komponenter

For at kunne repræsentere de enheder der optræder i det system som virksomheden ønsker at repræsentere, er der brug for en fælles definition der beskriver de enkelte enheder der indgår i systemet. Til dette formål har vi valgt, at laveste fællesnævner for en enhed der kan indgå i systemet, kaldes en *komponent*. En *komponent* er en abstrakt definition af de mulige typer af elementer der kan indgå i systemet. Det vil sige at alt fra servere og processer til services, alle kan betragtes som komponenter i systemet.

Alle komponenter deler konceptet om at kunne antage forskelige tilstande, sådan at algoritmer kan arbejde på et abstraktionslag, hvor der ikke indgår specifikke informationer om hvilken type af komponent der er tale om. For at understøtte en mulighed for at systemet kan fortælle hvilken bruger der er ansvarlig for at reparere eller problemsøge komponenten, skal komponenten også indeholde associationer til en eller flere brugere.

For at systemet i andre sammenhænge kan kende forskel på typerne af komponenter, defineres der en række komponenttyper som de enkelte komponenter kan antage form som. For hver type vil der også være brug for at kunne definere forskellige metadata, hvorfor det er valgt at der kan defineres et ubegrænset antal metadatatyper som knyttes på den enkelte komponenttype. I praksis betyder det f.eks. at der kunne findes en type metadatatype *procesnavn* som kun kan tilføjes på komponenter af typen *proces*.

En kort gennemgang af de krav der er til en komponent listes her:

- Type
- Tilstand
- Tilstandskriterium
- Bruger association
- Generisk metadata
- Associationer til andre komponenter

**Type**
Typen af komponent giver mulighed for at differentiere komponenterne i den visuelle præsentation. Ydermere giver det mulighed for at definere hvilke metadata der skal knyttes til en given type.

**Tilstand**
Tilstanden bruges til at indikere komponentens øjeblikkelige helbred. Hvis en anden afhængig komponent ønsker at reevaluere sin status ud fra tilhørende afhængigheder, så er det tilstanden på den enkelte komponent der kigges på.

De tilstande en komponent kan antage i denne løsning fremgår af Tabel 1.

| Tilstand | Beskrivelse |
|---|---|
| Up | Kan udføre sin primære funktion |
| Down | Kan ikke udføre sin primære funktion |
| Warning | Kan stadig udføre sin primære funktion men komponenten bør kontrolleres |

Tabel 1 - Komponenttilstande

**Tilstandskriterium**
Med kriterium menes der en forudsætning for komponentens tilstand. Ved at tilføre et kriterium som beskriver hvornår komponenten har en given tilstand, får brugeren af systemet mulighed for selv at definere hvordan tilknyttede afhængigheder skal fortolkes nå komponentens tilstand skal evalueres. Der kunne f.eks. være tale om en afhængighed til et eller flere redundante systemer, og det ville derfor give mening at komponenten ikke blev tilskrevet et dårligt helbred, blot fordi en af flere redundante afhængigheder var ude af drift, men derimod kun hvis alle var det. Et krav til sådan et kriterium, er at det skal være let forståeligt for en bruger med mild teknisk baggrund.

**Bruger association**
En bruger association i forbindelse med en komponent er en person i firmaet eller organisationen som har en rolle i forhold til komponenten. Roller kan defineres dynamisk, men løsningen vil som udgangspunkt indeholde rollerne: *Ejer* og *Interessent*. I forbindelse med fejlrapportering vil det være de tilknyttede personer der anføres som kontaktpersoner. Rollen *Ejer* tilføres for at lette systemadministratoren arbejde, med at finde ud af hvem der skal kontaktes i tilfælde af problemer. Rollen *Interessent* er tænkt som en person der ønsker at blive holdt ajour omkring komponenten, men ikke har noget ansvar for komponentens drift.

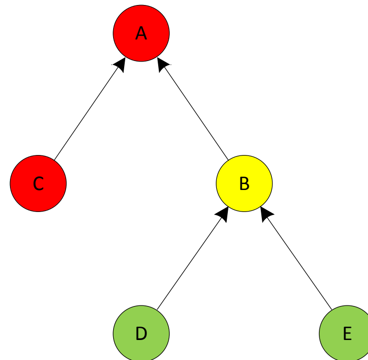
### 2.8.4 Struktur på komponentafhængigheder

Der er brug for en måde at hvorpå vi kan strukturere afhængighederne mellem komponenterne. Det kan gøres på mange måder, men vi er kommet frem til at en træstruktur minder meget om den måde komponenterne i virkeligheden er forbundet på. Konkret vil det betyde at afhængighederne mellem komponenterne opbygges som en retningsbestemt graf, herefter nævnt som afhængighedsgrafen. Retningen i forbindelserne fortæller om der er tale om at komponenten selv afhænger af den modstående komponent, eller om det er den modstående komponent der afhænger af komponenten, hvilket er illustreret i Figur 7.

**Figur 7 – Enkelte afhængighed med retning (A afhænger af B)**

Ved at opbygge afhængighedsgrafen som en acyklisk retningsbestemt graf (engelsk: *acyclic directed graph*) kan vi simplificere måden hvorpå data kan delegeres ud i grafen af komponenter. Det er simplere at traversere en acyklisk struktur rekursivt, end det er at traversere en cyklisk struktur. Inden beslutningen blev taget gennemgik vi mange scenarier og fandt ikke at der, i vores system, var nogen årsag til at vælge en cyklisk graf med den ekstra kompleksitet det ville medføre.

### 2.8.5 Evaluering af komponenttilstand




Figur 8 – Afledte tilstande i afhængighedsstrukturen.

For at kunne give et billede af systemets tilstand er det vigtigt at have defineret hvilke tilstande en komponent i systemet kan befinde sig i. For at minimimere kompleksiteten i forhold til udarbejdelse af rapporter og diagnoser, har vi valgt at fokusere på tre tilstande: *Up*, *Down* og *Warning*.

Tilstanden på en komponent skal kunne sættes på følgende tre måder:

1. Komponenten får sat tilstanden gennem webbaserede brugergrænseflade.
2. Status sættes gennem det eksterne API.
3. En komponent som der afhænges af, ændrer status hvilket medfører reevaluering af komponentens egen tilstand.

Alle komponenter indeholder et eller flere kriterier, i vores tilfælde et logisk udtryk, som fortæller hvornår hvilke specifikke tilstande er opnået for den givne komponent. Komponentens tilstand, og dens tilhørende kriterier, evalueres når en af følgende tre situationer fra listen ovenfor optræder.

**Bestemmelse af tilstand ud fra kriterium**

Med et logisk udtryk menes der et boolsk udtryk der kunne have dette format:

$$A \wedge (B \vee \neg C)$$

Reglerne for systemets boolske udtryk, er udtrykt i EBNF grammatikken på Figur 9 for at give et bedre indtryk af hvilke muligheder udtrykket indeholder.

```
program         :       expression EOF!
                ;

expression      :       orexpr
                ;

orexpr          :       andexpr (OR^ andexpr)*
                ;

andexpr         :       term_negate (AND^ term_negate)*
                ;
term_negate     :       NEGATION^? term
                ;

term            :       VARIABLE
                |       INTEGER
                |       '('! expression ')'!
                ;


NEGATION        :       '^';
OR              :       'OR' | 'or';
AND             :       'AND' | 'and';
VARIABLE        :       ('A'..'Z' | 'a'..'z')('A'..'Z' | 'a'..'z' | '0'..'9')*;
INTEGER         :       ('0'..'9')+;
WS              :       ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+
```

Figur 9 – Grammatik til parsning af boolsk udtryk

Kigger man på Figur 9 igen, så vil man også kunne se at vores logiske udtryk benytter sig af operatorerne i Tabel 2 (s. 23). Det er kolonnen *Tekst version* som systemet benytter sig af, da de mere korrekte matematiske operatorer i kolonnen *Operator*, ikke er tilgængelige på et normalt keyboard layout.

| Operator | Tekst version | Beskrivelse |
|---|---|---|
| ∧ | "^" | Logisk AND |
| ∨ | "AND" | Logisk OR |
| ¬ | "OR" | Logisk NOT |

Tabel 2 – Logiske operatorer

I forbindelse med evaluering af logiske udtryk, anvendes oversættelsen i Tabel 3 på komponenternes tilstand for at kunne oversætte tilstanden til true/false.

| Tilstand | Boolsk værdi |
|---|---|
| **Up** | True |
| **Warning** | True |
| **Down** | False |

Tabel 3 – Oversættelse mellem tilstand og logisk værdi

Grunden til at vi har valgt at bruge almindelige boolske udtryk, er primært fordi vi opfatter simple boolske udtryk som noget en stor del af tekniske IT administratorer i dag kan finde ud af at udtrykke og beskrive. Det er også fordi det giver os mulighed for at sammensætte udtrykket automatisk, såfremt der ikke er angivet noget. I det tilfælde at intet udtryk er angivet, vil udtrykket for *Up/Down* tilstanden blot fortolkes som en serie af AND operationer hvor alle komponentens afhængigheder indgår. I tilfælde af manglende *Warning* udtryk vælger vi blot at udelade evaluering. En anden, endnu mere teknisk årsag, er at det blev indviklet at beskrive et boolsk udtryk med mere end to udfald, da vi ønsker at udtrykke tre tilstande: *Up*, *Down* og *Warning*. Der findes teknisk materiale[i] der beskriver multi-valued logic, men det ville ikke passe sammen med ønsket om at gøre det simpelt for almindelige brugere at arbejde med.

**Automatisk re-evaluering af associerede komponenter**

For hver tilstandsændring der sker, så evalueres komponenterne i afhængigheds-grafen rekursivt fra den berørte komponent og i retning af de komponenter der afhænger direkte af komponenten. Hvis man tager udgangspunkt i Figur 8 (s. 25), så betyder det at hvis *B* skifter status, så evalueres *D* og *E's* status igen. Hvis *D* eller *E* opnår en ny tilstand efter *B's* statusændring, så vil deres nærmeste afhængigheder også gen-evalueres og sådan fortsættes rutinen indtil enden af træet. Rutinen er altså den samme for hver komponent gennem hele træet. Det man dog skal være opmærksom på, i forbindelse med altid at videreformidle en tilstandsændring, er at det ikke er alle tilstande der må ændres, som en konsekvens af en associeret tilstandsændring. Lokalt opståede *Down* og *Warning* tilstande, må ikke overskrives med resultatet af en gen-evaluering. Definitionen af *lokale tilstande* er tilstande der er opstået direkte på komponenten. Det kunne eksempelvis være en server der pludselig rapporteres som ude af drift. Af den årsag er der i komponentmodellen indført et felt *LocalError* der fortæller om en eventuel *Down* tilstand er opstået lokalt eller ej. Følgende flow forklarer problemstillingen som *LocalError* flaget skal løse.

**Uden *LocalError* flag:**

(*A* er af typen server og har tilstanden: *Up*)

1. *A* oplever en strømafbrydelse og antager tilstand: *Down*
2. En association til A ændrer tilstand til *Up*
3. *A* gen-evaluerer sin status med resultatet *Up*.
4. *A* antager status *Up*.

Problemet her er at *A's* tilstand overskrives med *Up*, selvom *A's* virkelige tilstand er *Down* fordi der fysisk set stadig er strømafbrydelse. Hvis man derimod kigger på *LocalError* inden der evalueres, så kan der tages højde for lokale tilstande som ikke må overskrives. Her er et eksempel på hvordan det løses:

**Med *Local Error* flag:**

(*A* er af typen server og har tilstanden: *Up*)

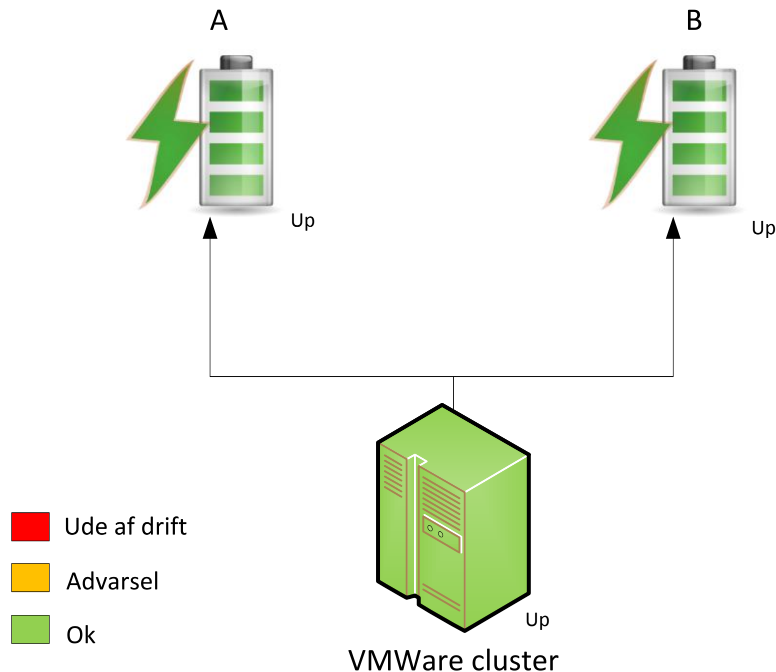
1. *A* oplever en strømafbrydelse og antager tilstand: *Down* og *LocalError: true*.
2. En association til *A* ændrer tilstand til *Up*.
3. *A*'s *LocalError* flag tjekkes og operationen stoppes da værdien er *true*.

Det er med andre ord kun direkte tilstandsændringer der kan ændre ved *A*'s tilstand når *LocalError* er *true*.

### 2.8.6 Advarselstilstand

En af årsagerne til at der indføres kriterier der beskriver komponentens tilstand, er at det skal være muligt at udtrykke hvornår en komponent er helt ude af drift og hvornår den blot bør signalere en advarsel. En advarselstilstand er derfor en proaktiv forudsigelse af en mulig fejl, sådan at brugere kan få en tidlig notifikation af systemet.

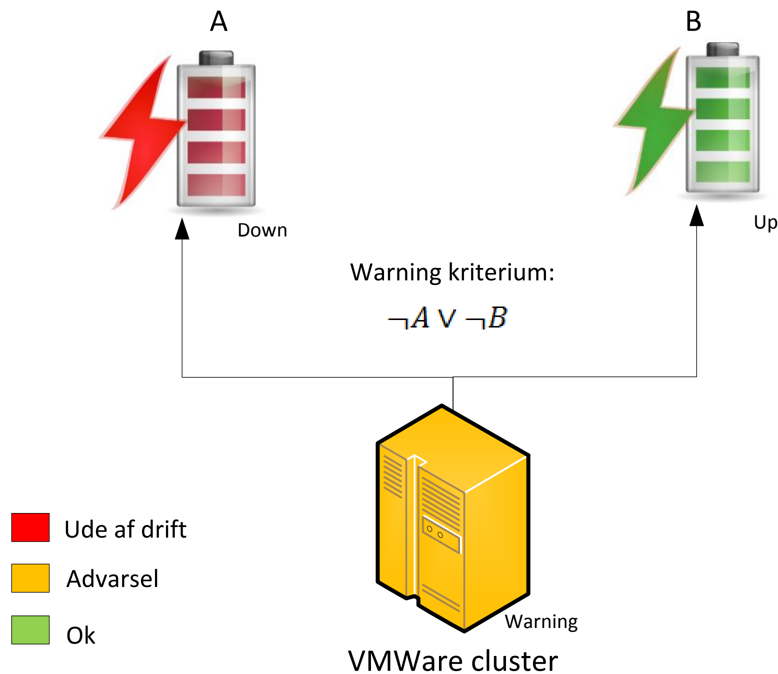
Lad os først kigge på Figur 10 som viser en situation hvor vi har tre komponenter: *VMWare cluster* og to redundante strømkilder. Hvis man kigger på pilene i billedet, så viser den at *VMWare cluster* afhænger af både strømkilde *A* og strømkilde *B*.




Figur 10 – Alt kører efter planen

Virkeligheden er selvfølgelig sådan at *VMWare cluster* fra figuren kun behøver at enten strømkilde *A* eller strømkilde *B* er til fungerende. Derfor giver det ikke mening

at komponenten rapporterer en fejl hvis kun den ene strømkilde er fungerende (læs: *Up*). I den situation er der mere tale om at der bør rejses en advarsel, netop som det ses i Figur 11 (s. 29).



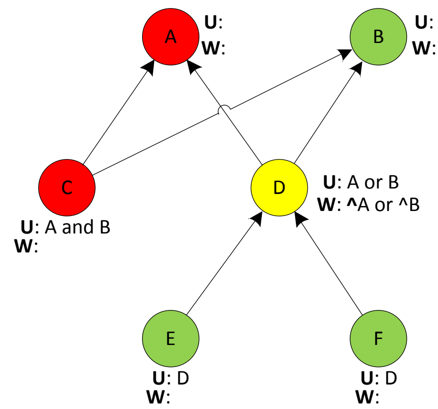

**Figur 11 – Advarselstilstand er opnået fordi *A* ikke er tilstede (komponent: funktionsdygtig)**

Et logisk udtryk der beskriver tilstanden for hvornår denne advarselstilstand er opnået, vil derfor se således ud:

$$\neg A \lor \neg B$$

For læserens skyld er udtrykket allerede anført i Figur 11.

*Up* udtrykket har altid højere prioritet end *Warning*. Det skal forstås sådan at hvis *Up* er false, så evalueres *Warning* udtrykket ikke. Det fungerer sådan fordi at en advarselstilstand ikke skal kunne tage præcedens over en fejltilstand. En anden måde at sige det på er at der kun tjekkes for eventuelle advarsler når komponenten ikke er i fejltilstand.




Figur 12 – Eksempel på afhængighedsgraf med logiske tilstandsudtryk

Hvis man kigger på komponent *D* på Figur 12 og anvender oversættelserne i Tabel 3 (s. 26), får vi følgende evaluering:

1. D's *Up* udtryk evalueres som:

$$1 \lor 0 = \boldsymbol{True}$$

2. Fordi D's *Up* udtryk er true, evalueres *Warning* udtrykket som:

$$\neg 1 \lor \neg 0 = 0 \lor 1 = \boldsymbol{True}$$

3. D's endelige komponenttilstand bliver *Warning*.

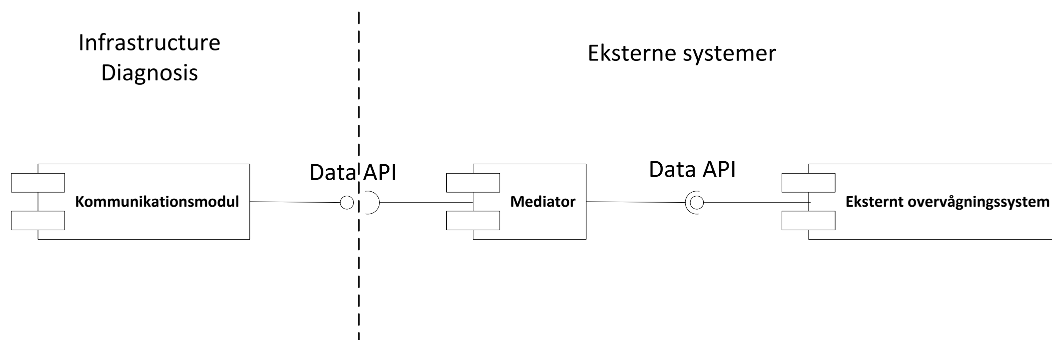
## 2.9 Mediator

I Infrastructure Diagnosis er det gjort muligt at komponenterne kan opdateres med live tilstandsinformation, via løsningens API. Tilstandsinformationer kan tilegnes fra forskellige former for overvågningsværktøjer. Det der mangler, er et program der kan stå for at indhente tilstandsoplysningerne fra overvågningsværktøjer og føre dem ind i Infrastructure Diagnosis. Dette program kalder vi en mediator, grundet dets funktionalitet. Vi har valgt at udvikle et eksempel på en mediator, da vi så får nemmere ved at illustrere hvilken positiv indvirkning live tilstandsinformationer giver vore løsning.

Vi har besluttet os for at tage udgangspunkt i overvågningsværktøjet PRTG. Det har vi gjort da firmaet vi arbejder for allerede benytter sig af det, og vi derfor ikke skal bruge tid på at installere og opsætte det. PRTG stiller et API til rådighed, igennem hvilket vi kan tilgå alle informationer som PRTG har og vil indhente.

Opgaven står i at få oprettet associationer mellem de komponenter som PRTG og Infrastructure Diagnosis har til fælles, og yderligere implementere en logik der står for, på basis af associationerne, overføre tilstandsinformation imellem programmerne. En association er et link mellem en komponents repræsentation i PRTG og den selv samme komponents repræsentation i Infrastructure Diagnosis. Der er yderligere tilknyttet data, som skal bruges af Mediatoren i forbindelse med tilstandsopdateringer.

På Figur 13 nedenfor, har vi illustreret Mediatorens plads mellem vores løsning og et overvågningsværktøj. Den vertikalt stiplede linje markerer opdelingen mellem vores løsning og de eksterne komponenter, og her ses det også at vi ikke anser Mediatoren som værende en del af Infrastructure Diagnosis.




Figur 13 – Kommuniation med 3. parts produkter

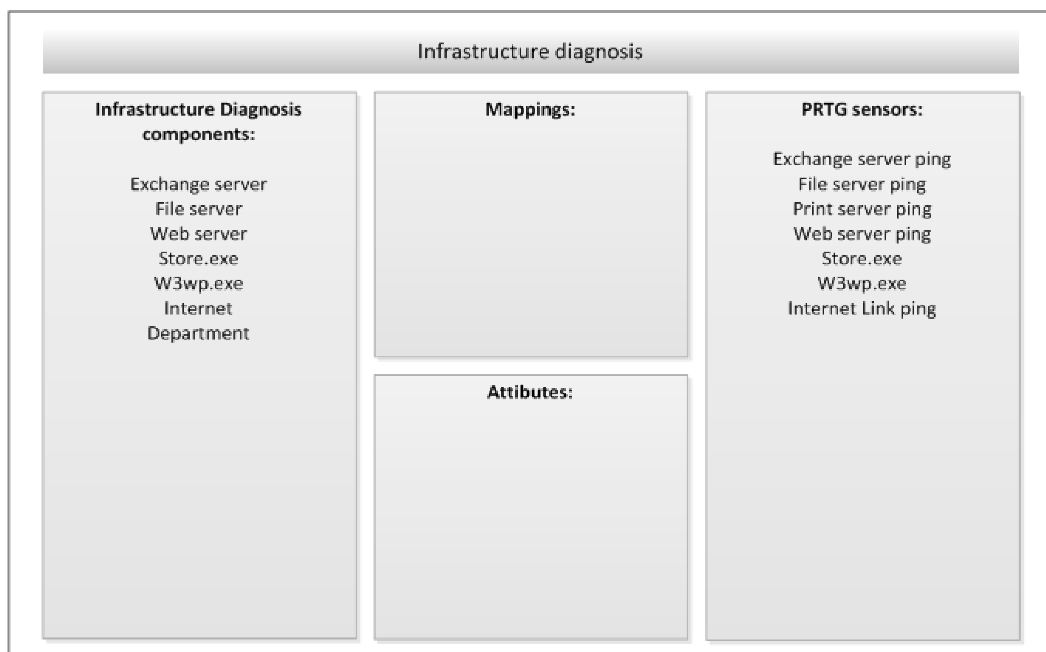
Da overvågningsværktøjernes API'er ikke er ens, vil man være nød til at lave en ny mediator for hvert overvågningsværktøj man ønsker at hente live tilstands-informationer fra. Alternativt kunne det laves i en samlet mediator, som så blot skulle have kendskab og funktionalitet, til at fungere med flere forskellige API'er.

Vore krav til Mediatoren er følgende:

- Skal kunne indhente komponenter fra PRTG.
- Skal kunne indhente komponenter fra Infrastructure Diagnosis.
- Skal kunne skabe en association mellem to relaterede komponenter fra henholdsvis PRTG og Infrastructure Diagnosis.
- Skal kunne overføre tilstandsinformationer automatisk

### 2.9.1 Mediator brugergrænseflade

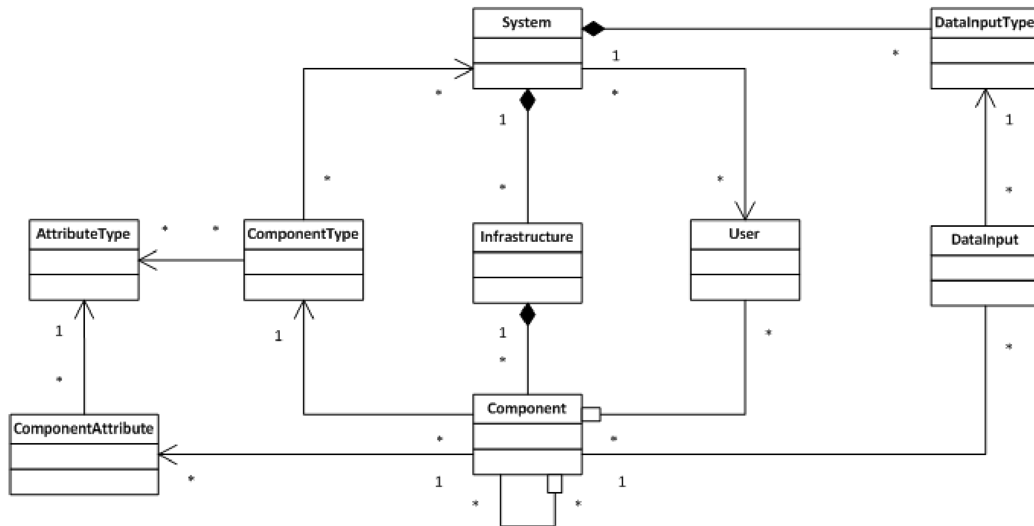
I forbindelse med analysen af Mediatoren, tegnede vi et udkast til hvordan vi kunne forestille os at GUI'en skulle se ud. Dette udkast fremgår af Figur 14. I venstre side vil komponenterne fra vores løsning blive listet. I højre side vil komponenterne fra PRTG fremgå. Associationerne vil fremgå af den øverste boks i midten og nedenfor denne vil egenskaber omkring associationerne fremgå. Det vil f.eks. være komponent id'er, navn, opdateringsinterval og datatype.




Figur 14- Første udkast til Mediator GUI

## 2.10 Data Design model

Designmodellen indeholder alt hvad systemet behøver for at kunne implementeres og opfylde de krav til funktionalitet som der er stillet i analysen. Forskellen her er, at modellen nu indeholder konkrete klasser som behøves for at lave en konkret implementering af systemet.




Figur 15 – Design model

## 2.11 Analysekonklusion

Vi har på baggrund af vores analyse, udarbejdet en løsningsbeskrivelse som opfylder de, i kravspecifikationen, stillede krav.

Vi fandt undervejs i analysen ud af at det var vigtigt at udvikle en prototype af en mediator til projektet. Dette skyldes at vi ønsker at illustrere hvilken positiv effekt import af live tilstandsinformationer har på vores løsning.

Vi mener det er vigtigt at påpege at dette system ikke kun vil have sin berettigelse inden for IT systemer. Vi forestiller os at vores løsning lige såvel med fordel kunne benyttes i andre miljøer, som f.eks. et hospitals miljø med ansatte og udstyr. Domænemodellens struktur tillader en sådan brug, selvom man så ville kunne diskutere navngivningen af nogle System og Infrastruktur klasserne. Mediatoren ville så kunne tilpasses så den i stedet for at indhente status information fra et overvågnings værktøj, i stedet hentede informationer omkring medarbejderes status fra f.eks. et tidsregisteringssystem eller en Exchange server.

# 3 Teknologier

Løsningen vil primært implementeres i C#. Løsningen vil være web baseret og være drevet af ASP.NET MVC web platformen som kører under IIS 7.5 på Windows Server 2008. Det er en fordel for os at bruge denne teknologi da vi allerede har kendskab til platformen og ved at den er veldokumenteret og har et stort community bag sig.

Herudover vil der blive brugt en række øvrige teknologier som skal give os muligheder som ikke er bygget direkte ind i ASP.NET MVC. Den største mangel er muligheden for at danne interaktiv grafik som brugeren kan bruge som overblik og navigation. I tråd med den manglende interaktive grafik, mangler også muligheden for at lave en grafisk editor. Her vil vi benytte os af en flash baseret løsning da denne platform er kendt for at levere rige grafiske effekter i web baserede løsninger. En flash løsning kræver dog et kendskab til ActionScript som ingen af os har tidligere erfaring med.

Her følger en liste af teknologier der er brugt i projektet:

- **C#** - *Programmeringssprog brugt i ASP.NET delen*
- **MSSQL** – *Fysisk SQL baseret data lager*
- **ASP.NET MVC** – *Framework brugt til Web applikationen*
- **WCF** – *Framework brugt til webservice lag*
- **Entity framework** – *ORM Framework brugt til data forespørsler og persistering*
- **Adobe Flash Builder** – *IDE brugt til udvikling af ActionScript og Flex applikationer*
- **Kalileo Diagrammer** – *Flashbaseret diagram komponent.*
- **SignalR** – *Teknologi til to-vejs kommunikation over HTTP*
- **ANTLR** – *Værktøj til generering af parser.*
- **WPF** – *GUI framework benyttet til Mediator konfigurator.*

## 3.1 Teknologier og programmer der behøver yderligere introduktion

Vi vil her komme med en meget kort introduktion til et par af ovenstående teknologier som vi ikke mener at man kan forvente at læseren har forudgående kendskab til.

### 3.1.1 ASP.NET MVC

Microsoft's framework til udvikling af webapplikationer der deler implementeringen op i Model, View og Controller. Frameworket indeholder rige muligheder for at arbejde med data på objektniveau, gennem introduktionen af *Model Binding* som automatisk oversætter form-data til komplekse objekter, ud fra HTTP data. Denne del af frameworket passer rigtig godt sammen med objekt orienteret data pesistens gennem Entity Framework, som også nævnes i dette kapitel. MVC lægger i høj grad

op til *Configuration By Convention* hvilket betyder at systemet kræver et minimum af tilpasning hvis man blot følger de strukturregler og designkonventioner der lægges op til fra Microsoft's side.

### 3.1.2 WCF – Windows Communication Foundation

WCF står for Windows Communication Foundation og er Micosofts officielle webservice framework. Frameworket tilbyder at udvikleren blot skal koncentrere sig om at definere en kommunikationskontrakt og en eller flere konkrete implementationer af denne. Derfra sørger WCF for alt hvad der har med kommunikation og sikkerhed at gøre. Det betyder at man som udvikler på implementeringstidspunktet, ikke nødvendigvis behøver at bekymre sig om hvordan servicen skal tilgås. Der kan senere vælges at eksponere servicen gennem eksempelvis Named Pipes, Binær TCP eller blot HTTP. Samme service kan eksponeres gennem flere forskellige kommunikationskanaler samtidigt, med forskellig konfiguration af sikkerhed. Samme filosofi gælder for opsætningen af sikkerheden for hver kommunikationskanal.

### 3.1.3 Entity Framework

Entity Framework er Microsofts bud på en enterprise ORM der understøtter komplekse associationer og avancerede caching strategier. Entity Framework bygger på ideen om at usynliggøre hvordan data hentes, ved at antage en deklarativ tilgang til hvordan der forespørges og arbejdes med data. Hvis man benytter Entity Framework med SQL som datakilde, så betyder det at udvikleren aldrig ser en linie SQL, men istedet beskriver hvilke data der ønskes hentet eller manipuleret, hvorefter Entity Framework omsætter den abstrakte forespørgsel til faktisk SQL. Det er især fordelagtigt i scenarier hvor der er tale om lazy loading af associeret data, hvilket Entity Framework håndterer automatisk. Eager loading er implementeret mere konservativt, ved at udvikleren på den enkelte forespørgsel definerer om der skal eager loades en eller flere associationer i det samme roundtrip til det fysiske lager. Entity Framework stiller en avanceret grafisk editor til rådighed som udvikleren kan bruge til at modellere sin datamodel og senere vælge at persistere strukturen på en faktisk datakilde.

### 3.1.4 Adobe Flash Builder

Adobe Flash Builder er Adobe's kommercielle værktøj til flashudvikling. Værktøjet har sin force i udvikling af applikationer i Flex frameworket, hvilket er det framework som Kalileo Diagrammer (nævnt senere i dette afsnit), benytter sig af. Værktøjet er et typisk IDE bygget på Eclipse platformen og giver mulighed for den sædvanlige rutine, hvor kildekoden kompileres, programmet køres, programmet afprøves og

lukkes, koden ændres, kompileres, programmet afprøves igen og så videre. Intet nyt under solen her. Programmet er desværre kommercielt og benyttes på en licens tilhørende undertegnedes arbejdsgiver *AKAIT*.

### 3.1.5 Kalileo Diagrammer

Kalileo Diagrammer er en komponent baseret på Adobe Flex som er Adobes framework til udvikling af business-line applikationer i Flash. Programmeringssproget er ActionScript4 hvilket er et typisk C-syntax derivat som derfor er forholdsvis hurtigt for Java/C# udviklerne at sætte sig overfladisk ind i. Kalileo Diagrammer tilbyder en simpel måde at visualisere og editere vilkårlig data leveret i GraphML formatet. Komponenten tilbyder at udvikleren kan udvide funktionaliteten via et dependecy injection system som giver mulighed for at indføre egen-implementeret logik istedet for den indbyggede. Dette designpattern kaldes typisk IoC – Inversion of Control. Det er først ved runtime at applikationen får kendskab til den konkrete binding mellem interfaces og konkrete instanser. Denne form for dependency injection i Kalileo Diagrammer giver os altså mulighed for selv at tage kontrol og udvide med ekstra logik på en masse områder hvor firmaet bag komponenten har ønsket at åbne op for udvidelser. Nogle områder kan dog være en udfordring at udvide, da man sætter den originale kode ud af drift og risikerer at skulle genskabe meget avanceret logik, selvom man blot ønsker at ændre på simple funktioner.

### 3.1.6 SignalR

SignalR giver mulighed for at foretage to-vejs live kommunikation mellem server og klient - en funktionalitet der længe har eksisteret et ønske om at kunne benytte i browserbaserede applikationer. Websockets (RFC 6455) er den browser-understøttede mulighed for denne funktionalitet og er blevet standard i webengines som Mozilla og WebKit. Derfor tilbyder SignalR nu en mulighed for at bygge bro mellem native understøttelse af to-vejs kommunikation og simuleret understøttelse i browsere der ikke implementerer *Websockets*. Den store styrke i SignalR, er at det er usynligt hvilken af de to metoder der anvendes på et givent tidspunkt.

Mere specifikt giver SignalR mulighed for at klienten kan abonnere på data fra brugerdefinerede tilslutningspunkter og derefter rejse et interrupt på klienten når nye begivenheder finder sted. I vores tilfælde kører klienten javascript og de konkrete interrupts leveres via callback funktioner. På den måde kan klienten automatisk foretage en opdatering af data i brugergrænsefladen når der opstår en begivenhed på serveren. Hvis browseren understøtter *Websockets* bruges dette til at opretholde en åben forbindelse mellem server og klient, hvis ikke, så benyttes konceptet *Long Polling*. *Long Polling* undgår hyppige periodiske forespørgsler til serveren, ved at anvende en

teknik hvor SignalR åbner en HTTP forbindelse fra klienten til webserveren og lader denne stå åben så længe som forbindelsens timeout tillader det. Hver gang forbindelsen termineres, oprettes automatisk en ny forbindelse således at der altid er en åben forbindelse mellem klient og server. Denne åbne forbindelse kan bruges til at sende data og rejse begivenheder på klienten, uden den forsinkelse der opstår ved traditionel polling. Det giver samtidig et lavt antal forespørgsler til serveren, sammenlignet med polling teknikken. Ulempen kan være at serveren skal håndtere mange åbne forbindelser samtidig, men siden introduktionen af ..NET 4.0, er maksimum antal samtidige forbindelser pr. CPU, som standard sat til 5000. På vores testsystem med 8 kerner, giver det et maksimum på 40.000 samtidig forbindelser, hvilket rigeligt møder vores krav til systemet.

### 3.1.7 ANTLR

ANTLR står for *Another Tool for Language Recognition* og er en parser generator der producerer LL(*) Recursive Descent Parsere. ANTLR forventer en grammatik i EBNF format og understøtter at generere parsere til en række forskellige programmeringssprog, heriblandt C# som vi benytter os af. I projektet gør vi brug af ANTLR 3 samt *ANTLRWorks* 1.4.3 som er et specialudviklet IDE til test og udvikling af EBNF grammatikker i samspil med ANTLR generatoren.

# 4 Design

Her vil vi forklare hvordan systemet skal designes i forhold til teknologier og softwaredesign. Vi vidste fra starten at grafisk visualisering og editering var et emne der i sig selv kunne optage et helt projekt, og har derfor valgt at fokusere teknologieksperimenterne på et gratis færdiglavet visualiseringsframework. Det er derfor valgt at anvende Kalileo Diagrammer til denne del. I forbindelse med parsing af logiske udtryk er der fokuseret på ANTLR da vi allerede havde erfaring med at beskrive grammatikker heri.

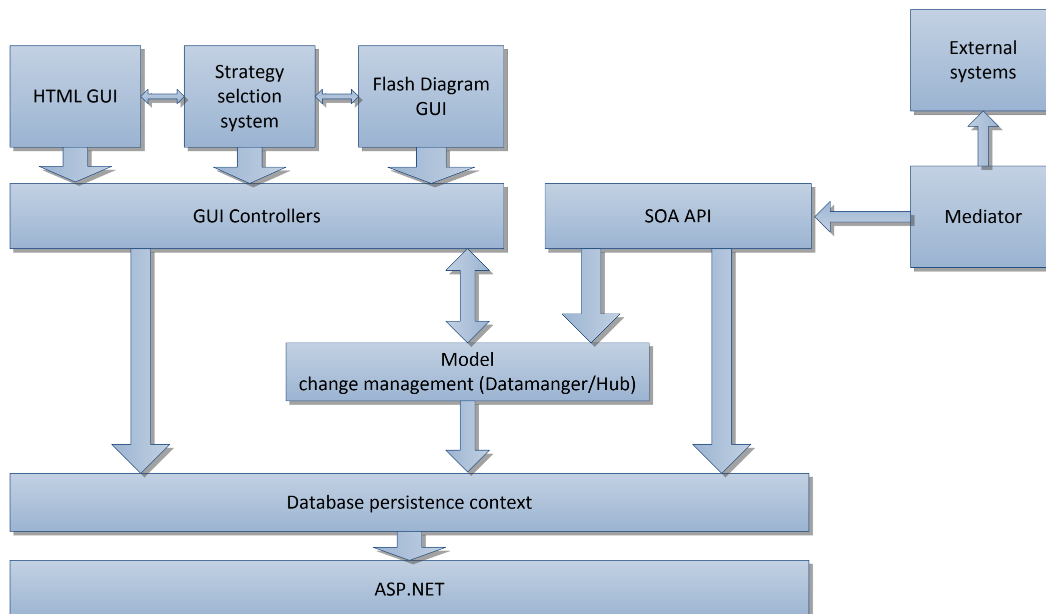
## 4.1 Teknologieksperimenter

Inden vi kunne påbegynde implementeringen af en prototype var vi nødt til at lave nogle eksperimenter med nogle mulige teknologier for at finde ud af hvad der passede bedst i vores design. Da vi allerede var ret overbeviste om at benytte ASP.NET som drivkraft bag systemet, var det mere eller mindre bestemt at webserveren og alt hvad der kørte i dette lag, skulle kodes mod .NET i C#.

Det var dog en udfordring at finde et godt .NET baseret framework til grafisk fremstilling som kunne benyttes gratis. Derfor faldt blikket på Kalileo Diagrammer som tilbyder en communitylicens som må bruges af alle, blot ikke ved videresalg. Kalileo Diagrammer er udviklet i Flex som benytter ActionScript og compiles til en Adobe Flash fil. Vi har ingen tidligere erfaring med udvikling til Flash og det var derfor ikke sikkert at løsningen ville være brugbar for os. Efter at have studeret sproget, konkluderede vi dog at det ville være muligt, inden for rimelig tid, at lære så meget om syntaxen, semantikken og logikken, at det ville være muligt at bruge. ActionScript er objektorienteret og sproget minder på mange områder om Java og C#, bortset fra det deklarative GUI modelleringssprog MXML som er en del af frameworket Flex.

## 4.2 System model

I Figur 16 kan vi se en oversigt over de komponenter som systemet består af, samt en generel indikation af hvordan de enkelte komponenter er associeret med hinanden.




Figur 16 – System model

## 4.3 HTML brugergrænseflade

Brugerfladen til editering (*HTML GUI*, Figur 15) er som nævnt webbaseret og til dette formål er det valgt at benytte *ASP.NET MVC* framework fra Microsoft.

Model og Controller delen af systemet lægger sig tæt op af domænemodellen, hvilket giver en hensigtsmæssig kategorisering af logikken i systemets controllere. I forhold til navigering på den webbaserede del af systemet, giver det også en fordel idet *ASP.NET MVC* implementerer en URL routing mekanisme der gør at controllere kan kaldes via pæne konfigurerbare stier i formatet /{*Controller*}/{*Action*}/{*Id*}, som eksempelvis "/*Components*/*Show*/*2*". For hver controller findes der typisk en tilhørende model som repræsenterer den kategori af data som controllerens navn indikerer. I det nævnte eksempel med *Components*, ville der eksempelvis findes en tilhørende model *Component* indeholdende det data som *Show* metoden ville forvente at præsentere (*View*).

*ASP.NET MVC* giver os mulighed for at implementere HTTP baserede opkaldspunkter som andre dele af systemet, end blot den grafiske brugerflade, kan bruge til at kommunikere med. Dette kan bruges i forbindelse med kommunikationen med grafkomponenten (*Flash Diagram GUI,* Figur 15) som ikke afvikles af ASP.NET, men derimod er en prækompileret komponent som der skal kommunikeres med via. et, til formålet, udviklet interface.

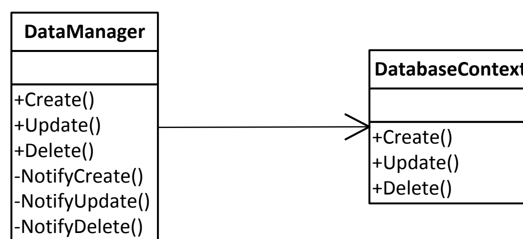
Ændringer i data forventes at opdatere den grafiske brugerflade automatisk og til dette er det valgt at bruge *SignalR* som et opensource framework bygget til ASP.NET stack'en. *SignalR* giver systemet mulighed for at webserveren, vha. asynkron kommunikation med javascript i klientens browser, kan aktivere hændelser på klienten (den enkelte browserinstans) samtidig med at de opstår.

At anvende en metode hvor serven kontakter klienten, er ny i forhold til den traditionelle metode, hvor klienten typisk kontakter serveren i fastlagt intervaller, for at se om der er ændringer i data.

Denne metode anvendes også i forbindelse med live opdatering af dataændringer i diagram komponenten når data ændres og notificeres via change management systemet (se *Model Change Management*, Figur 15). Det kan lade sig gøre fordi diagramkomponenten er flash baseret og understøtter mulighed for at kommunikere med browseren. Læs mere om automatiske notifikationer i kapitel 4.7.

## 4.4 Model Change Management

Datamanager konceptet, i dette projekt, er ideen om at have et fælles sted at rapportere om ændringer og begivenheder (Se *Model Change Management* i Figur 15). En *DataManger* vil typisk wrappe funktionalitet der håndterer den faktiske udførelse af en ændring eller begivenhed, således at den der kalder manageren, aldrig oplever en forskel i forhold til at foretage ændringen direkte i databaselaget. Det vil typisk være *Create, Update, Delete* operationer som *DataManager* wrapper omkring, se Figur 17.




**Figur 17 – Generisk visning af DataManager relation til database context.**

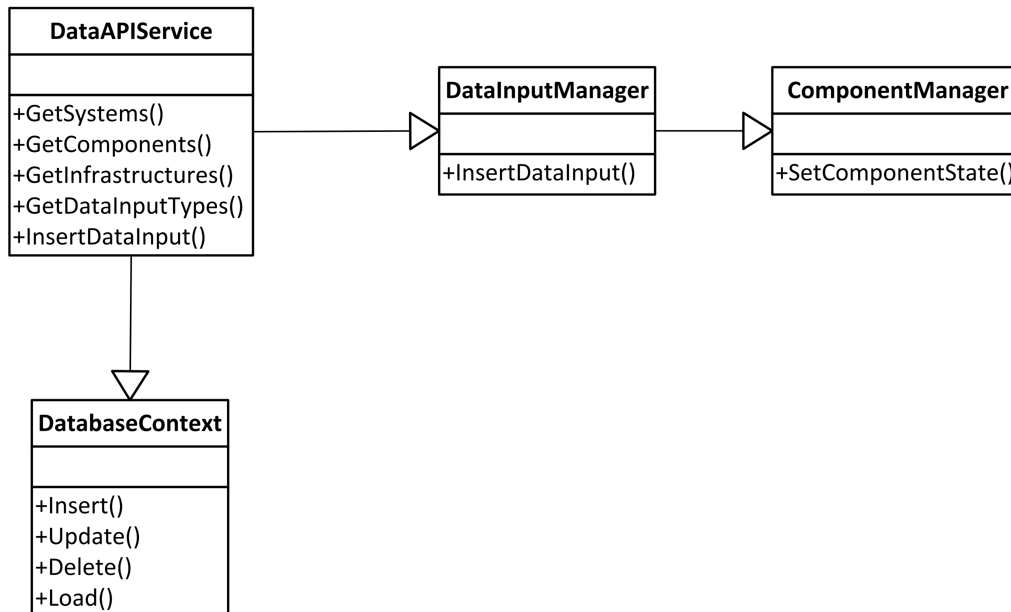
Den åbenlyse fordel med denne abstraktion, er at det giver en mulighed for at registrere og handle på alle ændringer og begivenheder i systemet et centralt sted, samt eventuelt at foretage relevante handlinger på baggrund af disse handlinger. Det fjerner også et ansvar fra udvikleren, som nu kun skal vedligeholde et enkelt sted i systemet, når der foretages ændringer. Det betyder at der implementeres en *DataManager* for hver overordnet type i systemet, som der kunne tænkes at opstå interessante ændringer eller begivenheder på.

Til det nuværende behov, er der udviklet to managers: *ComponentManager* og *DataInputManager*. I forbindelse med live-opdatering af data i den grafiske brugerflade, har vi implementeret logik i begge managers der sørger for at aktivere SignalR, sådan at alle aktive brugere af systemet, automatisk bliver orienteret om at der er opstået en ændring på systemet. Se *Notify**-operationerne *i DataManager*-klassen på Figur 17 (s. 40). For klarhedens skyld, så vil notifikationsmuligheden primært blive brugt til at pushe meddelelser ud via SignalR, omtalt i kapitel 4.7.

## 4.5 SOA API Interface

Denne del af systemet repræsenterer boksen SOA API i Figur 16 (s. 39).

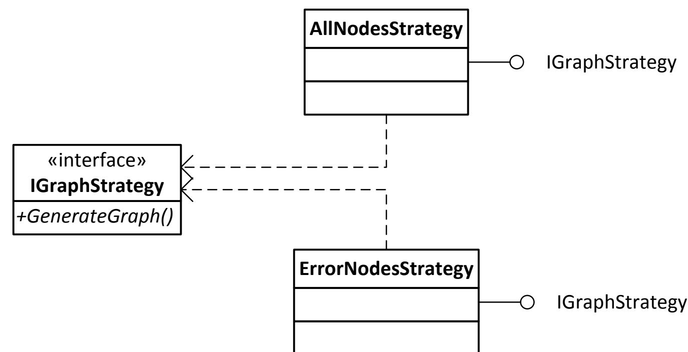
For at give mulighed for at kommunikere med systemet fra eksterne systemer, har vi valgt at anvende *Windows Communication Foundation*, som er et webservice framework udviklet af Microsoft ovenpå ASP.NET. WCF hostes på samme ASP.NET webserver som resten af løsningen, hvilket giver en fordel i forhold til at vi kan dele in-memory information med resten af systemet. Det kan lade sig gøre fordi webservice laget kører i samme proces som resten af ASP.NET og derfor har mulighed for at dele implementering med de øvrige komponenter der kører på ASP.NET, heriblandt webserveren som er ansvarlig for den grafiske brugerflade. Operationerne i webservicen er implementeret i klassen *DataAPIService* som kan ses på Figur 18 (s. 42). *Insert/Update*-operationer foretages igennem *Model Change Management* systemet som er beskrevet i kapitlet *Model Change Management (s. 40)*. Det betyder at data der tilføjes til den dynamiske del af modellen, automatisk opdateres asynkront i HTML GUI grænsefladen. Det kan også ses i Figur 18 (s. 42), hvor der anvendes henholdsvis *DataInputManager* og *DatabaseContext*, hvilket giver servicen mulighed for at manipulere (CRUD) model data. *DataInputManager* anvendes når der indsættes data, for at sikre at ændringer i den dynamiske del af modellen bliver håndteret og videreformidlet til resten af systemet. Til alt øvrig kommunikation med databasen, anvendes *DatabaseContext* klassen.




Figur 18 – Webservice klassediagram

## 4.6 Diagrambaseret visualisering

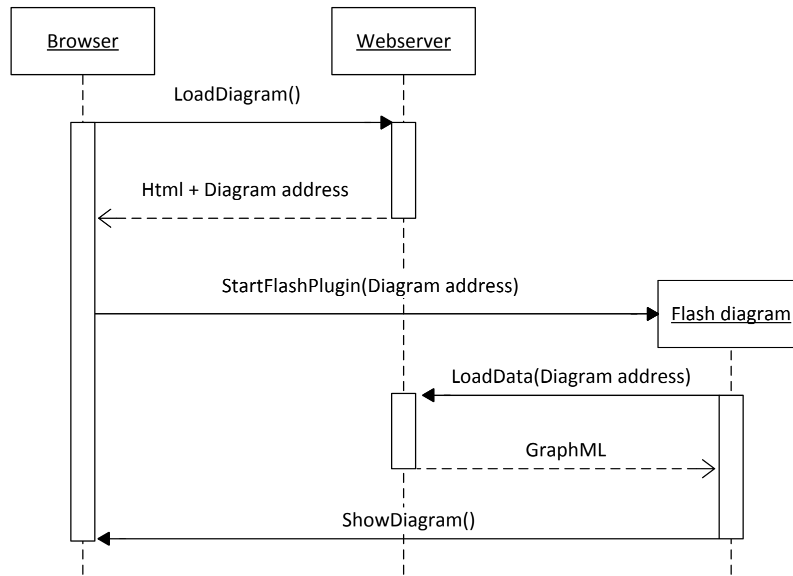
For at illustrere associationerne og tilstanden i systemet, giver systemet mulighed for at vise en grafisk illustration af komponenterne og afhængighederne mellem disse. Fordi at brugerne af systemet kan have forskellige formål med at vise den grafiske oversigt, og derfor ikke nødvendigvis ønsker at se hele systemet altid, så har brugeren mulighed for at vælge forskellige udtræksalgoritmer i forhold til hvilke komponenter der skal inkluderes i diagrammet. For at løse opgaven har vi valgt at implementere forskellige udtræksalgoritmer vha. et S*trategy* P*attern* interface, som lover at en konkret implementering vil tilbyde en specifik udvælgelseslogik. I Figur 19 vises klasseopbygningen af strategisystemet, samt to konkrete implementeringer. En strategi indeholder en algoritme der udvælger hvilke komponenter der skal vises og muligheden for at returnere en liste af disse i et binært format. Den der kalder strategien (Se *GUI Controller* Figur 16) vil herefter serialisere og præsentere listen i GraphML format for diagram komponenten således at der kan dannes en visuel repræsentation af udtrækket. Samtidig kan den grafiske brugerflade benytte samme udtræk, til at danne en tekstbaseret liste over hvilke komponenter algoritmen har inkluderet, som kan vises samtidig med den diagrambaserede visualisering.




Figur 19 – GraphStrategy pattern med konkrete implementationer

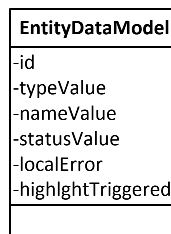
Visualisering af komponenter og associationer foregår vha. en flashbaseret komponent i den webbaserede brugergrænseflade. For at kommunikere med komponenten anvendes formatet GraphML. GraphML er et bredt format der forsøger at sætte en standard for filbaseret repræsentation af diagramdata og er implementeret af mange løsninger. At komponenten afvikles af flash, giver en udfordring i forhold til at komponenten ikke kan initialiseres med data når grænsefladen initialiseres, men derimod først kan begynde at hente data når browserens flash plugin er initialiseret.

Kommunikationen med grafkomponenten sker ved at klientens browser initialiserer diagram komponenten med information om hvilken strategi der benyttes. Herefter foretager diagram komponenten et request til webserveren hvor den forespørger data i GraphML format for den valgte strategi. Webserveren afvikler den ønskede strategi og returnerer data til diagrammet hvorefter diagrammet renderer komponentgrafen og viser resultatet i browseren. Sekvensdiagrammet i Figur 20 (s. 44) viser hvordan flowet af requests mellem browser, webserver og diagram eksekveres når den grafiske brugerflade aktiverer visning af en strategi.




Figur 20 – Request sekvens ved valg af visningsstrategi

I den flashbaserede komponent anvendes MVC pattern til håndtering af data mellem view, model og controller. Her har vi implementeret modellen i Figur 21 som view-delen af kildekoden benytter når hver enkelt komponent skal tegnes i grafen. Når data modtages fra webserveren tildeles dette direkte til de enkelte noder i grafen i det nævnte model format, sådan at noderne får mulighed for at reflektere de modtagne metadata der gælder for hver node. Umiddelbart før, tildeles den modtagne rå GraphML data, direkte til grafkomponenten som derefter udarbejder en grafisk visualisering af noder og associationer. Modellen understøtter mulighed for databinding på alle egenskaber, hvilket giver mulighed for at view'et øjeblikkeligt kan reflektere ændringer i modellen.



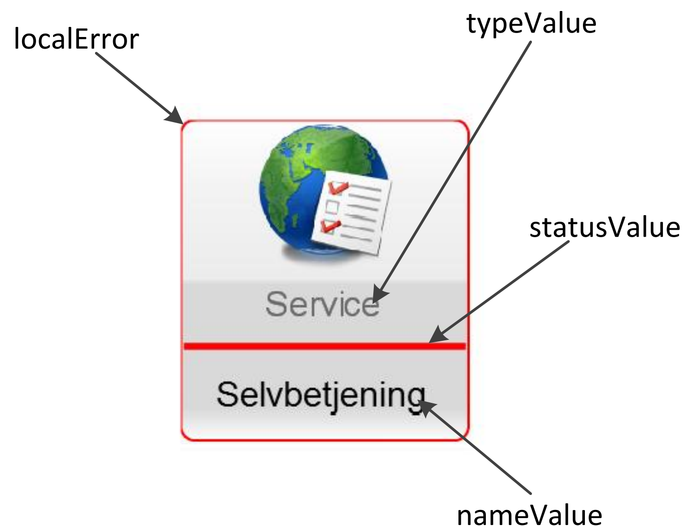

Figur 21 – Diagram data model

Det vil sige at view implementeringen benytter en bindingsmekanisme som gør det muligt at de visuelle komponenter automatisk reflekterer de egenskaber i modellen

som de er bundet til. De bundne og automatisk opdaterende egenskaber i den visuelle model inkluderer disse egenskaber i datamodellen:

- typeValue
- nameValue
- statusValue
- localError
- *highlightTriggered*

I Figur 22 vises det hvilke visuelle områder på den enkelte node-visning som modellens egenskaber hænger sammen med. Vi kan af gode grunde ikke illustrere hvordan *highlightTriggered* hænger sammen med visningen, da *highlightTriggered* blot benyttes for at fortælle modellen at der skal udføres en animation der highlighter noden visuelt. Dette sker ved at controlleren blot tildeler egenskaben en ny værdi, hvilket trigger en callback funktion ude i viewet som igangsætter highlight animationen. Årsagen til at denne egenskab er benyttet til dette formål, er at kommunikationen mellem model og view kun foregår gennem det indbyggede bindingsframework og derfor præsenter simpel metode til at kommunikere mellem disse på. Man kan argumentere for at det ikke er logisk at have egenskaber i modellen som blot fungerer som notifikationskanaler uden at værdien repræsenterer en brugbar værdi, men i vores tilfælde er argumentet at benytte et kommunikations-system der allerede fandtes, fremfor at bygget noget nyt.

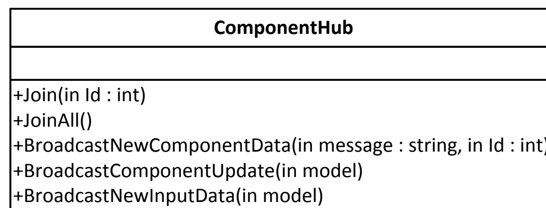



Figur 22 – En visuel nodes tilknytning til datamodellen

## 4.7 Live opdatering af data i HTML grænseflade

I forbindelse med live opdatering af data, understøttet af *DataManager* konceptet, så har vi valgt at implementere et *Hub* koncept, understøttet af *SignalR*.

Implementeringen minder om *Observer Pattern*, idet en *Hub* beskriver en central klasse hvor forskellige dele af applikationen kan abonnere på begivenheder, samt muligheden for at udsende begivenheder. I Figur 23 ses en konkret implementering af en Hub. I tilfældet med *ComponentHub* på Figur 23 har vi henholdsvis *Join** og *Broadcast** metoder. *Join*-metoderne fortæller Hub'en at den der kalder, er interesseret i at modtage begivenheder fremover, enten på alle, eller på en enkelt komponent. *Broadcast*-metoderne bruges af parter der er interesserede i at rejse en specifik begivenhed. Denne begivenhed sendes herefter til alle der har abonneret via *Join*-metoderne.




**Figur 23 – Hub**

Den teknologi som sørger for at vores notifikationssystem i praksis kan flytte beskeder fysisk og ubesværet mellem server og klient, er leveret af frameworket SignalR som der kan læses mere om i kapitel *3 Teknologier*.

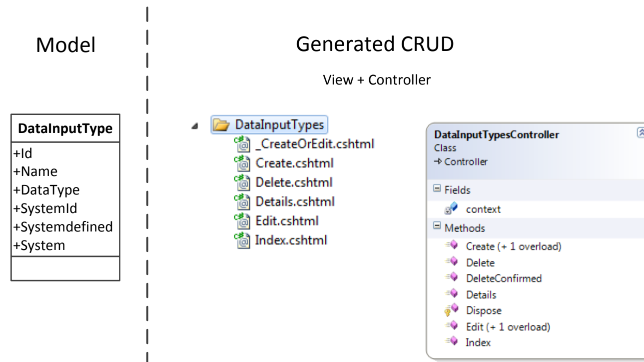
## 4.8 Tekst og HTML baseret editering

Muligheden for at editere data i modellen eksisterer som et essentielt led i at give brugeren mulighed for at konfigurere og tilpasse systemet i henhold til gældende ønsker og krav der hvor systemet er installeret. Editering foregår udelukkende via den tekstbaserede HTML brugerflade og der er gjort brug af autogenererede grænseflader der hvor det er muligt (se kapitel 4.9). Det betyder at vi udelukker editering via diagramkomponenten for at fjerne et lag af kompleksitet i løsningen. Det har fra starten været et mål at editering skulle kunne lade sig gøre via en tekst/html baseret grænseflade. Hvis man kigger på Figur 15 (s. 33), så er editeringslogikken implementeret i *GUI Controller* klasserne som benytter sig direkte af database *Database Persistence Context* til indlæsning af data og *Model Change Management* systemet til persistering af data.

## 4.9 Model CRUD scaffolding

På baggrund af den databasemodel der er udarbejdet i Entity Framework, er det muligt at bruge værktøjer til at generere en række simple CRUD interfaces til brug i webgrænsefladen. Softwaren der anvendes til at generere CRUD interface, er *MvcScaffolding 1.0.7* som er opensource og gratis at anvende. Efter genereringen er det

muligt at ændre i de resulterende filer, sådan at der kan tilføjes eller fjernes funktionalitet der ligger udover det som autogeneratoren har kunne antage ud fra modellen. Generatoren understøtter eksempelvis ikke editering af mange-til-mange relationer, hvilket vi derfor har været nødsaget til at efterimplementere i den genererede version af CRUD grænsefladen. Strukturen på de autogenererede filer, kan ses i Figur 24 nedenfor.




**Figur 24 - Eksempel på autogenereret CRUD**

## 4.10 Evaluering af komponenttilstande

Evalueringen af komponenttilstande foregår i klassen *ComponentDataManager* omtalt i kapitel 4.4. Årsagen til at logikken er placeret der, er at det giver mulighed for at *ComponentDataManager*, sammen med *ComponentHub* kan broadcaste ændringer i komponenternes status, samtidig med at status evalueres. I forhold til en grafisk brugerflade, giver det hurtig respons, eftersom ændringsbegivenheden bliver rejst med det samme og ikke først når evalueringen af alle komponenter er overstået.

## 4.11 Parsning af boolske udtryk

Det set af boolsk algebra vi vil bruge i evalueringen af tilstandskriterier, er defineret her:

$$B = \langle B, \wedge, \vee, \neg, 1, 0 \rangle$$

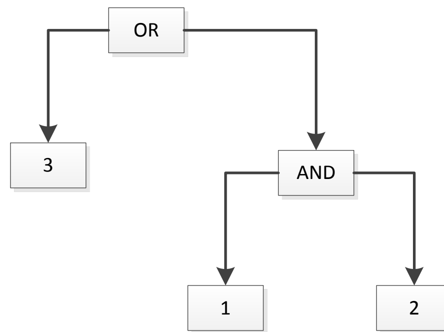
**Equation 1**

For at parse det boolske udtryk, benytter vi parser som overholder operationernes præcedens og understøtter uendelig dybde af indlejrede parenteser. Parseren returnerer et AST (*Abstract Syntax Tree*) som repræsenterer udtrykkets sammensætning, i en træstruktur (Se AST model på Figur 26 (s. 48)).

Komponenterne i de logiske udtryk er i databasen repræsenteret ved komponentens database ID, som det vises her:

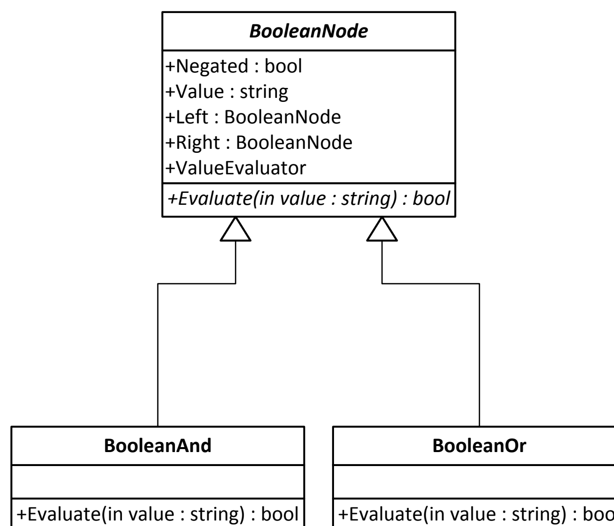
*1 AND 2 OR 3*

Når vi modtager et AST fra parseren, indeholder træet derfor struktur og værdier som det ses i Figur 25.




Figur 25 – Abstract Syntax Tree fra boolsk udtryk parser

Mens træet gennemløbes, oversættes de enkelte noder løbende til konkrete instanser af *BooleanNode* klasser fra Figur 26 som repræsenterer de logiske operationer. Det giver os mulighed for at udføre en specifik logik for hver nodetype. Ved at angive en *ValueEvaluator* (Læs om *IBooleanNodeEvaluator* i kapitel 5.1), som er en reference til en funktion der returnerer true/false ud fra *Value*, kan noderne gøre brug af en ekstern logik der foretager et opslag i databasen, hvor der hentes sandhedsværdi for den givne komponent (reglerne for oversættelse af status til boolsk værdi findes i Tabel 3 (s. 26)). Noder der repræsenterer en logisk operation, vil returnere den logiske konsekvens af nodens logiske operator med værdierne af de tilhørende to under-noder som højre og venstre input.




Figur 26 – BooleanNode AST model

### 4.12 Grafisk visning af komponenttilstande

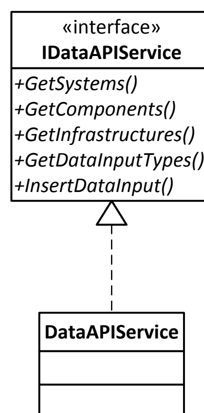
I overblikket vil tilstandene *Up*, *Down* og *Warning* henholdsvis være markeret med farverne grøn, rød og gul. Se Figur 12 (s. 30) som viser et eksempel på en afhængighedsgraf.

Tilstanden på komponenter kan sættes på flere måder:

1) *Den første* måde er at tilstanden sættes direkte på komponenten gennem det grafiske web interface.
2) *Den anden* måde er gennem det eksterne Datainput API, som eksempelvis kunne være når der periodisk importeres eksterne data fra andre systemer.
3) *Den tredje* måde er at komponenten ændrer tilstand som et resultat af en ændring i en af de tilhørende associationer der er sat op på komponenten.

### 4.13 SOA API Grænseflade

SOA API Grænsefladen (Se *SOA API*, Figur 15), eller webservice laget som vi kalder det, definerer et API som eksterne systemer kan benytte til at interagere med systemet, typisk for at rapportere live data. Mulighederne i API'et kan ses i det interface der illustreres i Figur 27. Som det ses, så tilbyder interfacet, udover at hente data, også mulighed for at indsætte data. Denne type operationer, selvom der kun er en, gør også brug af *model change management*, således at indgående data automatisk notificerer interesserede lyttere. Læs mere om webservicen i kapitel 3.1.2.

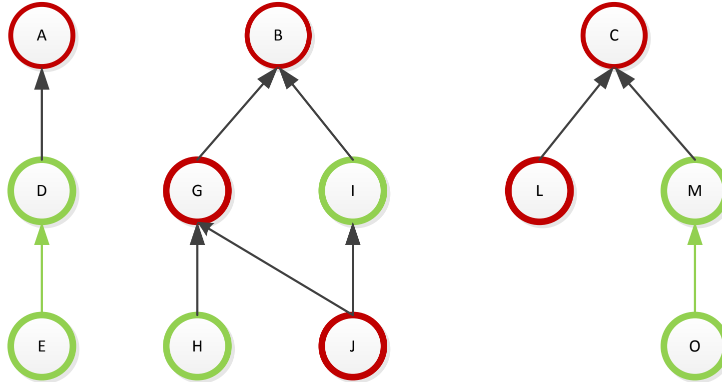



Figur 27 – Webservice klassediagram

### 4.14 Konsekvensanalyse

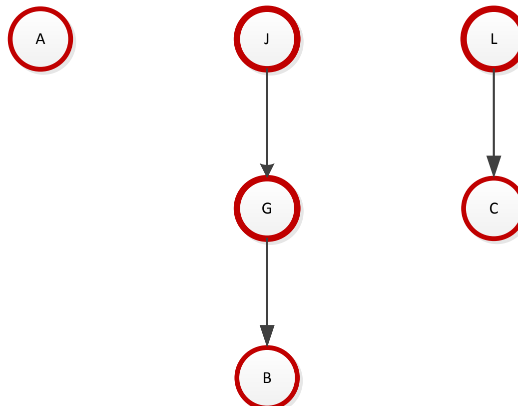
I konsekvensanalysen foretages der en analyse af hvilke konsekvenser det vil have at tage en eller flere komponenter og antage at de er i *Down* tilstand og derefter få vist hvilke konsekvenser det har på resten af systemet. Måden det fungerer på er ved at bygge en datastruktur der indeholder alle de stier som berøres når hver komponent i

*Down* tilstand evalueres rekursivt. For hver node i stien registreres hvilken status noden har på det givne sted i stien. Det giver en struktur, der kan bruges af præsentationslaget til at visualisere hvilke noder der er årsag til at en afledt *Down* tilstand er opnået på en associeret node.




Figur 28 - Resultat af simuleret Down tilstand på A, B og C. Rød = Down, Grøn = Up

I Figur 28 kan vi se at node A ikke resulterer i nogen følgende *Down* tilstande, hvor både B og C, begge har underliggende evalueringsstier som resulterer i *Down* tilstande. Den interessante visning, kommer hvis man fjerner alle de grønne noder og derved kun efterlader de stier der resulterer i tilstand *Down*. Hvis man derefter vender grafen på hovedet, så vil man se et træ som øverst viser hvilke noder der er ramt af den simulerede ændring og derunder har en sti der fortæller hvordan tilstanden er opstået. Den "rensede" figur kan ses i Figur 29.




Figur 29 - Renset og omvendt evalueringstræ

Med det rensede og omvendte træ kan vi nu udlede sigende forklaringer som f.eks. *"J er nede fordi G er nede fordi B er nede"*. Det kan naturligvis udtrykkes som en indrykket liste i en tekstbaseret visning, så det visuelt udtrykker hierarkiet i evaluerings-rækkefølgen.

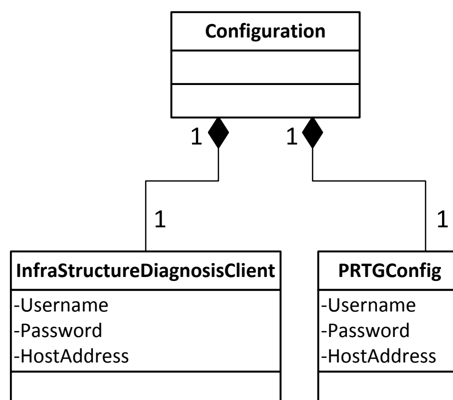
## 4.15 Mediator

Dette kapitel omhandler Mediatorens design og funktionalitet. Som tidligere beskrevet er Mediatoren et program hvis formål er at indhente tilstandsoplysninger fra et overvågningsværktøj og indsende tilstandsinformationer til de, af os på forhånd, udvalgte komponenter til Infrastructure Diagnosis.

Mediatoren skal kunne forbinde til API'erne fra Infrastructure Diagnosis og PRTG. Herefter skal den på baggrund af oprettede associationer, kunne opdatere komponenter i Infrastructure Diagnosis, med tilstandsinformation fra PRTG. Dette skal den så blive ved med så længe vi ønsker det, og til dette har vi brug for en schedulerings funktionalitet, der sørger for at opdatere tilstande i de intervaller vi ønsker det gjort. Vi vil nu beskrive Mediatorens datamodeller, til henholdsvis API og komponent konfiguration og senere logikken og hermed også job schedulerings-funktionaliteten.

### 4.15.1 Datamodel for mediator API konfiguration

Mediatoren har først og fremmest brug for at kunne autentificere op imod de to APIer, som den skal benytte sig af. Til dette har vi brug for brugernavn, password og adresse, til begge API'er. Figur 30 nedenfor illustrere hvordan modellen for konfigurationsfilen vil se ud.




Figur 30 – API konfiguration model

Passwordet gemmes i denne løsning i clear text, men skulle løsningen bruges i produktion, ville det være fordelagtigt at gemme passwordet krypteret.

### 4.15.2 Datamodel for mediator komponent konfiguration

Det skal det være muligt for Mediatoren at indlæse og indsende tilstands-informationer. For at gøre Mediatoren i stand til at kunne det, har den behov for at kende til en række informationer som til sammen udgør en association. Vi har brug

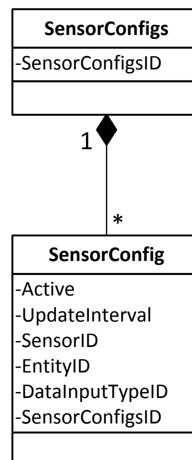
for følgende informationer for at kunne skabe de ønskede associationer mellem Infrastructure Diagnosis og PRTG:

**Aktivitets indikation (*Active*)**, for at kunne slå opdateringen af en association til eller fra.

**Opdateringsinterval (*UpdateInterval*)**, for at kunne bestemme opdaterings-intervallet af en association.

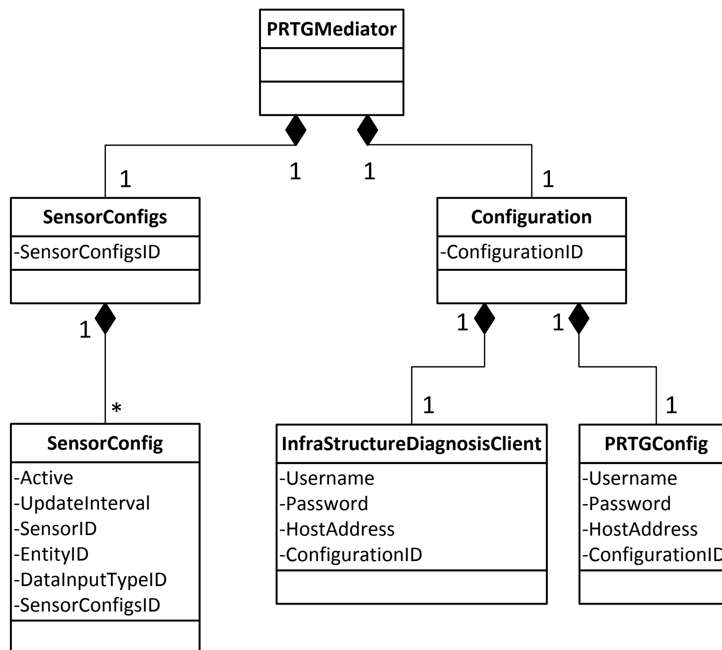
**SensorID** og **EntityID** for at have unikt kendskab til en associations komponent IDer i Infrastructure Diagnosis og PRTG.

**Data input**, for at kende til typen af tilstandsinformation. Figur 31 nedenfor illustrerer hvordan modellen vil se ud for komponent konfigurationen, og den viser også at det er muligt for os at have flere associationer.




Figur 31 – Komponent konfiguration model

Samlet set vil modellen se ud som på nedenfor på Figur 32, hvor der er tilføjet en øvre klasse som ejer begge de tidligere del-modeller.

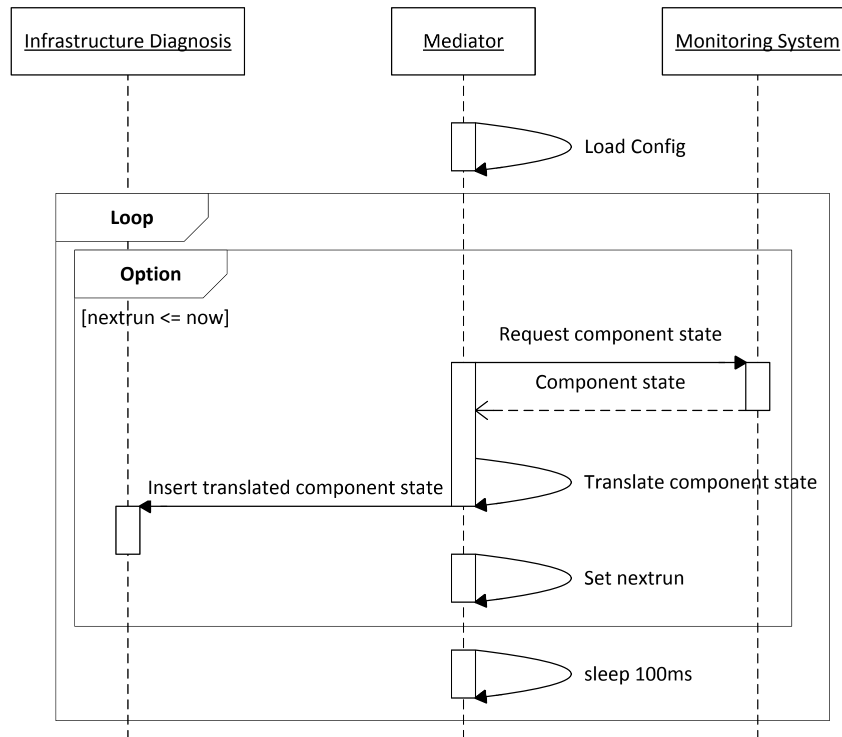



Figur 32 – Mediator data model

Med datamodellen på plads går vi videre til at beskrive logikken der skal drive Mediatoren.

### 4.15.3 Logik

Vi har nu datamodellen på plads, og skal nu fokusere på at give Mediatoren funktionalitet. Dette vil vi gøre med udgangspunkt i sekvensdiagrammet illustreret på Figur 33 (s. 54)




Figur 33 – Sekvensdiagram for Mediatoren

Først indlæses den initiale konfiguration. Det er her Mediatoren kommer i besiddelse af autentificerings- og associationsinformationer. Herefter træder vi ind i et loop, hvorfra den nuværende tid tjekkes op imod en værdi kaldet nextrun, som alle associationer besidder. Nextrun benyttes til at fortælle jobscheduleren hvornår associationen skal opdateres næste gang. Er der nogle komponenter der skal opdateres, altså, at deres nextrun tidspunkt er mindre eller lig med nuværende tid, så forespørger Mediatoren overvågningsværktøjet, i dette tilfælde PRTG, om tilstandsinformation, hvilket den efter at have modtaget det, oversætter til en tilstand som Infrastructure Diagnosis kan modtage. Herefter sendes den oversatte tilstand til Infrastructure Diagnosis og en fuld tilstandsopdatering har fundet sted. Herefter træder vi ind i en sleep kommando, som sørger for at der går 100ms før vi igen starter forfra. Dette er for at lette belastningen af system ressourcerne, på den maskine der driver Mediatoren.

# 5 Implementering

## 5.1 Parsning af boolsk logik med ANTLR

Til at parse og forstå de bolske udtryk der bruges i systemet, har vi beskrevet en grammatik i EBNF, se Figur 34, og brugt ANTLR til generere en R*ecursive Descent Parser* i *C#*.

```
public program
                    :       expression EOF!
                    ;

expression          :       orexpr
                    ;

orexpr              :       andexpr (OR^ andexpr)*
                    ;
```

Figur 34 – EBNF Grammatik eksempel

Den genererede parser består af to dele: En lexer og en parser. Lexeren anvendes til at oversætte input til de tokens der beskrives i grammatikken. Disse tokens bruges efterfølgende af parseren som nu kan fokusere på den syntaktiske og semantiske fortolkning af input.

I Figur 35 ses et eksempel på brug af den genererede parser:

```
//Create lexer from input
var lexer = new ErrorReportingBooleanExpressionLexer(expression);

//Get tokens from lexer
var tokens = new CommonTokenStream(lexer);

//Create parser with tokenized input
var parser = new ErrorReportingBooleanExpressionParser(tokens);

//Get AST by calling program() method from EBNF grammar
var ast = parser.program().Tree;
```

Figur 35 – Eksempel på brug af autogenereret parser

Ved traversering af AST i forbindelse med evaluering af udtrykket, oversættes alle noder til instanser af en mere specifik nedarvning af *BooleanNode* klassen. *BooleanNode* er en baseklasse for alle noder i det abstrakte syntaks træ.

Ved blot at anvende en baseklasse der skal nedarves fra, kan hver nodetype implementere sin egen logik for hvordan den specifikke logiske operation udføres. Se Figur 36 (s. 56) for et eksempel på dette.

```
public class BooleanAnd : BooleanNode
{
    protected override bool Evaluate(bool a, bool b)
    {
        return a && b;
    }
}
```

Figur 36 – Specifik node evaluering

*BooleanNode*, som er base for alle noder, understøtter at der angives en evalueringsstrategi for hvordan indholdet af den enkelte node skal oversættes til en true/false værdi. Interfacet for alle sådanne strategier er givet i *IBooleanNodeEvaluator*. Det er implementeret sådan, for at give plads til at man i fremtiden kan vælge at evaluere indholdet af noderne på anden måde, uden at ændre direkte i base klassen. Lige nu indeholder de boolske udtryk altid databasens unikke ID for komponenterne som nodernes værdi, hvilket vi kan oversætte med den konkrete strategi i Figur 36. Denne strategi overholder samtidig reglerne fra *Tabel 3 – Oversættelse mellem tilstand og logisk værdi (s. 25).* Skulle man derimod senere have et ønske om f.eks. at hente data fra en anden kilde end databasen, så kunne det let implementeres vha. en anden strategi. Se Figur 38 for den faktisk implementering af logikken der bruges i systemet.

***Bemærk:*** Når Figur 38 studeres, så giver det formentlig bedre forståelse hvis man husker på at et udtryk, i databasen, er udtrykt på samme måde som eksemplet i Figur 38.

*238 **AND** ((4782 **OR** 9823) **AND** 2312)*

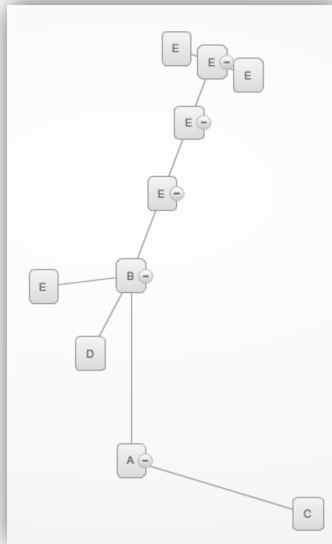
Figur 37 – Eksempel på kriterium som det gemmes i databasen

```
public class InfrastructureDiagnosisNodeEvaluator : IBooleanNodeEvaluator
{
    public bool Evaluate(string value)
    {
        int id = int.Parse(value);

        //Fetch component from database
        var entity = DbContext.entities.Single(x => x.Id == id);

        //Convert status to boolean
        if (entity.Status== EntityStatus.UP || entity.Status == EntityStatus.WARNING)
            return true;
        else
            return false;
    }
}
```

Figur 38 – IBooleanNodeEvaluator med opslag i database

## 5.2 Flashbaseret diagramkomponent

Til grafisk visualisering af associationerne mellem komponenterne i system-overblikket, benytter vi en flash baseret grafkomponent *Kalileo* fra firmaet Kap Lab, udviklet i programmeringssproget ActionScript 4 på frameworket Adobe Flex. Ideen bag Kalileo komponenten er at tilbyde en mulighed for at visualisere og editere en liste af associerede komponenter. Ved blot at give komponenten en GraphML struktur, kan Kalileo rendere en standardvisning som tydeliggør sammenhængen mellem givne noder som det ses i Figur 39.

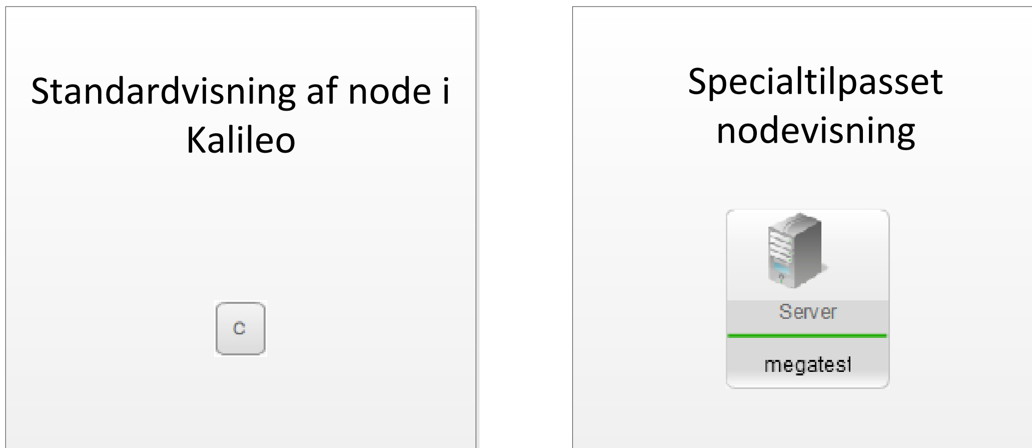



Figur 39 – Standardvisning af data i Kalileo

I vores system har vi dog imidlertid et krav om en mere tilpasset visning af komponenterne, hvilket Kalileo håndterer ved at muliggøre at brugere af komponenten, kan definere specialtilpassede måder at præsentere de enkelte noder på. Det kræver dog et kendskab til ActionScript 4, idet interaktionen med komponenten er baseret på at der implementeres specifikke eksterne logikklasser som komponenten kan benytte hvis den får kendskab til disse. Interaktionen med Kalileo følger et MVC pattern, hvor Kalileo leverer motoren der tager sig af at håndtere brugerinput/interaktion, overordnet visning og tegning og layoutalgoritmer, mens vi selv står for at levere controlleren, modellen og eventuelle specifikke visninger.

I den specialtilpassede visning af hver node, som vi selv kan implementere og derved tilsidesætte den indbyggede, er der adgang til det tilhørende element i modellen. På den måde får vi fuld kontrol over hvordan det specifikke element i modellen skal

vises, rent grafisk. I Figur 40 sammenligner vi resultatet af vores specialtilpassede nodevisning, med den standard visning som Kalileo tilbyder.




Figur 40 – Sammenligning mellem standard- og specialtilpasset nodevisning i Kalileo

I forbindelse med implementeringen af tilpassede visninger, er det relevant at nævne at ActionScript 4 understøtter modelbinding direkte. Det giver mulighed for at have grafiske elementer der observerer ændringer og opdaterer automatisk. Det har vi anvendt i forbindelse med den specialtilpassede visning af noderne, da vi ønsker at visningen automatisk skal reflektere de ændringer der opstår i modellen. Måden det funger på, er vist i Figur 41, som viser hvordan ActionScript, med en meget simpel syntaks, tilbyder at håndtere hele orkestreringen af notifikationer og observeringer der er involveret når vi snakker modelbinding. Måden vi definerer bindingen i GUI, sker via MXML som er det sprog der bruges til at beskrive den grafiske brugerflade i Adobe Flex. De grafiske kontroller i Flex, understøtter ikke direkte modelbinding, men MXML tilbyder en smart deklaration som tager sig af den bagvedliggende logik og giver os mulighed for at koble felter der ikke understøtter modelbinding, sammen med felter som gør. Måden det fungerer på kan ses i Figur 42 hvor *_dataModel* er navnet på den konkrete model instans der beskriver en enkelt node af typen *EntityDataModel* (se Figur 41).

```
public class EntityDataModel
{

            [Bindable]
            public var typeValue:String;

            [Bindable]
            public var nameValue:String;

....
```

Figur 41 – Binding syntax i data model i AS3

```
....
<fx:Binding twoWay="true"  source="_dataModel.nameValue" destination="labelName"/>
....
```

Figur 42 – Binding syntaks i MXML

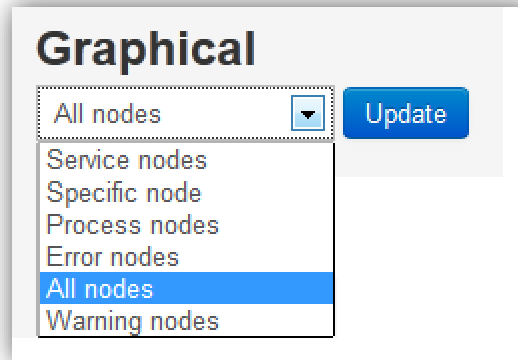
For at hente modeldata som vi ønsker at visualisere grafisk, forbinder vi til en adresse som browsere, der hoster Flash plugin'et, leverer med til os som det er illustreret på Figur 20 (s. 44). Her forventer vi at modtage et XML baseret GraphML dokument som beskriver hvilke noder der findes i grafen og hvordan de er associeret. I vores tilfælde benytter vi muligheden i GraphML for at medsende en række brugerdefinerede felter som beskriver metadata omkring den enkelte node. Denne metadata vil, når den er modtaget, bliver oversat til den model som de enkelte noder forventer. Det kan ses på Figur 43 hvordan et faktisk GraphML XML dokument ser ud.

```
<diagram ratio="1" layout="orthogonal">
  <node id="5">
    <data label="cphmodu02" status="UP" type="Server" localerror="False"/>
  </node>
  <node id="22">
    <data label="ESX VMware" status="UP" type="Server" localerror="False"/>
  </node>
  <link id="5->22" target="22" source="5" linkLine="2">
    <style thickness="1" arrowTargetType="arrow" renderingPolicy="solid"/>
    <data id="22" dependencyType="Physical"/>
  </link>
</diagram>
```

Figur 43 – GraphML over noder og links samt node metadata

## 5.3 Visningsalgoritmer

I systemrapportering delen af brugergrænsefladen, er det muligt at vælge hvilke udtrækskriterier der skal benyttes til at udvælge hvilke komponenter der skal vises i diagrammet. Det sker via en dropdown boks med forskellige valgmuligheder som det ses i Figur 44. Indholdet af denne boks er populeret med data som dynamisk hentes fra mulige *visningsstrategier* via reflection ved runtime.



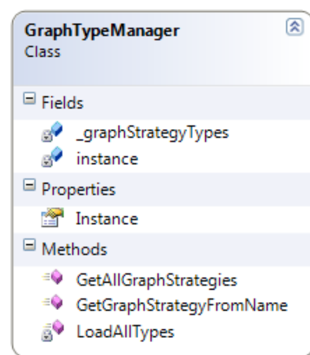

Figur 44 – Valg af visningsstrategi

Det er implementeret ved at de udtræksstrategiklasser (*IGraphStrategy*) som ønskes inkluderet i brugergrænsefladen, dekoreres med attributten *GraphStrategyAttribute* som det ses i Figur 45.

```
[GraphStrategy(FriendlyName = "Error nodes")]
public class ErrorNodesStrategy : IGraphStrategy
{
    ...
}
```

Figur 45 – GraphStrategy attribut dekoration

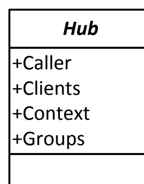
Attributten fortæller at strategien skal vises i brugerfladen, samtidig med at den giver mulighed for at specificere hvilket navn der skal vises i listen. I Figur 46 ses *GraphTypeManager* som indeholder den logik der står for at løbe vores assembly igennem og præsentere den liste af visningsstrategier som systemet p.t. har implementeret. Vi har valgt denne meget konventionsbaserede måde at inkludere algoritmer på, for at gøre det så simpelt som muligt at definere og aktivere nye algoritmer.




Figur 46 – GraphTypeManager

## 5.4 Live data med SignalR

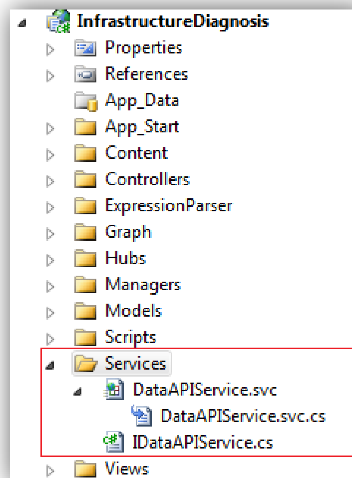
For at håndtere udfordringen med at levere levende datavisning som automatisk opdaterer når nye data tilføjes til modellen, anvender vi som nævnt tidligere i kapitel 4.7, *SignalR* til notifikationsprocessen mellem model og view. Konkret foregår det ved at installere SignalR på serveren for at give IIS processen mulighed for at håndtere og administrere vedvarende forbindelser til forbundne klienter. Kommunikationen med klienterne foregår gennem en SignalR defineret abstrakt klasse kaldet *Hub*, vist i Figur 47. Ideen med *Hub* er at du som udvikler skal implementere en subclass som indeholder alle de funktioner du ønsker at dine klienter skal have mulighed for at tilgå. SignalR vil herefter automatisk opfange kassen og gøre disse metoder tilgængelige for alle forbundne klienter.




Figur 47 – SignalR Hub

## 5.5 Webservice med WCF

Ved at kigge på Figur 48 får man en ide om hvor integreret webservice implementeringen er med resten af projektet. Strukturen spiller en rolle, da ASP.NET som udgangspunkt, benytter strukturen til at bestemme den bagvedliggende URL routing som kommer i spil når der skal bestemmes fra hvilke adresser de enkelte komponenter i projektet kan tilgås.




Figur 48 – Fysisk struktur og placering af webservice i projektet

Webservicen, beskrevet i kapitel 4.5, er implementeret via WCF. Til netop dette projekt har vi valgt at der ikke skal defineres nogen sikkerhedspolitikker omkring webservice laget, hvilket kan ses i attributten *bindingConfiguration* som er tildelt værdien *noSecurityWsHttpBinding*. I det tilfælde at vi skulle ønske at aktivere generelle sikkerhedsforanstaltninger, såsom kommunikation over SSL, ville det blot kræve nogle få deklarationer i *endpoint*-konfigurationen med information om hvilke dele af kommunikationen (message, transport), der skulle omfattes af krypteret dataudveksling. SSL konfigurationen ville foregå udelukkende gennem IIS.

```
<service name="InfrastructureDiagnosis.Services.DataAPIService">
        <endpoint
          binding="wsHttpBinding"
          bindingConfiguration="noSecurityWsHttpBinding"
          name="DataAPIService"
          contract="InfrastructureDiagnosis.Services.IDataAPIService" />
```

Figur 49 – WCF Endpoint konfiguration

Dette starter, på vores udviklingsmaskine, en service der kan kontaktes på adressen:

```
http://localhost/Services/DataAPIService.svc
```

Kommunikationsmetoden og mulige domæneadresser, styres af IIS serveren som i vores tilfælde er sat op til at kommunikere på port 80 over HTTP og acceptere alle domæneadresser.

**Eksponering af metadata**

WCF kan eksponere snitfladen for dataudveksling i form af WSDL. Det giver mulighed for at eksterne forbrugere i mange miljøer kan autogenerere en lokal klient der overholder snitfladen og simplificerer interaktion med servicen. Netop denne tilgang til at interagere med servicen er taget i brug i forbindelse med den implementerede prototype *PrtgMediator* som omtales i detaljer i kapitel 0. Eksponering af metadata bør deaktiveres i produktionssystemer.

## 5.6 Databaseadgang med Entity Framework

Ud fra data modellen som beskriver hvilke data vi ønsker at systemet skal indeholde, har vi mere eller mindre, direkte modeleret dette via Entity Frameworks grafiske model editor, hvilket gør os i stand til at lave opslag i stil med det der vises i Figur 50

```
//Initialiser en databasecontext
var context = new InfrastructureDiagnosisContainer();

//Hent system med Id = {id}
Models.System system = context.Systems.Single(x => x.Id == id);
```

Figur 50 – Eksempel på EF opslag

En rigtig interessant feature i EF som vi gør stor brug af er eager loading af associeret data. Det er værd at vise et kodeeksempel på dette, da det har en enorm indflydelse på systemets performance og ville kræve en stor mængde arbejde at implementere uden EF. Se Figur 51 hvor det tydeligt illustreres hvor simpelt EF gør det at indikere at der ønskes associeret data omkring infrastrukturer, i samme roundtrip som ved hentningen af systemer. Den faktiske SQL som kaldet resulterer i, er vist i *Figur 52 – Faktisk SQL ved eager loading opslag*.

```
//Initialiser en databasecontext
var context = new InfrastructureDiagnosisContainer();

//Hent system med Id = {id}, samt alle relaterede infrastrukturer
Models.System system = context.Systems.Include(x => x.Infrastructures).Single(x => x.Id == id);
```

**Figur 51 – Eager loading af associerede tabeller**

```
SELECT [Project2].[id]          AS [Id],
       [Project2].[name]        AS [Name],
       [Project2].[address]     AS [Address],
       [Project2].[c1]          AS [C1],
       [Project2].[id1]         AS [Id1],
       [Project2].[name1]       AS [Name1],
       [Project2].[description] AS [Description],
       [Project2].[companyid]   AS [CompanyId],
       [Project2].[systemid]    AS [SystemId]
FROM   (SELECT [Limit1].[id]          AS [Id],
               [Limit1].[name]        AS [Name],
               [Limit1].[address]     AS [Address],
               [Extent2].[id]         AS [Id1],
               [Extent2].[name]       AS [Name1],
               [Extent2].[description] AS [Description],
               [Extent2].[companyid]  AS [CompanyId],
               [Extent2].[systemid]   AS [SystemId],
               CASE
                 WHEN ( [Extent2].[id] IS NULL ) THEN Cast(NULL AS INT)
                 ELSE 1
               END                    AS [C1]
        FROM   (SELECT TOP (2) [Extent1].[id]      AS [Id],
                               [Extent1].[name]    AS [Name],
                               [Extent1].[address] AS [Address]
                FROM   [dbo].[systems] AS [Extent1]
                WHERE  [Extent1].[id] = @p__linq__0) AS [Limit1]
               LEFT OUTER JOIN [dbo].[infrastructures] AS [Extent2]
                            ON [Limit1].[id] = [Extent2].[systemid]) AS
       [Project2]
ORDER  BY [Project2].[id] ASC,
          [Project2].[c1] ASC
```

**Figur 52 – Faktisk SQL ved eager loading opslag**

## 5.7 Mediator kommunikation

Mediatoren skal, som tidligere forklaret, kunne kommunikere med de to API'er der er stillet til rådighed af Infrastructure Diagnosis og PRTG. Hvor API'et til vores løsning er en webservice baseret på SOAP, er API'et til PRTG baseret på REST. Begge API'er gør os i stand til at indhente og sende oplysninger, blot på forskellige måder. Vi vil i de næste par kapitler forklare hvordan vi har benyttet os af disse to typer API'er og illustrere det ved kodeeksempler.

### 5.7.1 PRTG

REST API'et er relativt let og intuitivt at benytte sig af, og kan benyttes direkte fra en browser. Vi kommunikerer med PRTG's REST API via HTTP protokollen og et specifikt kald kan være opbygget som illustreret i Figur 53 nedenfor.

```
http://{host address}/api/{method}.xml?id={component}
```

Figur 53 – REST API kald

Som det ses er dette kald simpelt opbygget. I kaldet skal der angives en host adresse, en metode og en komponent. Det er normal praksis at der følger en manual eller vejledning med til et REST API, da det ikke er muligt at gætte sig til funktionaliteten. I en sådan manual vil det fremgå hvilke kald man kan benytte sig af, og hvordan disse kan opbygges efter behov, og her er PRTG's API ingen undtagelse. Vi kunne f.eks. være interesseret i at indhente komponentdetaljer omkring komponenten med ID=1 og for at gøre det, opbygger vi et kald magen til det illustreret nedenfor med Figur 54.

```
http://10.10.50.39/api/getsensordetails.xml?id=1
```

Figur 54 – Udfyld REST API kald

PRTG returnerer, på ovenstående forespørgsel, en XML fil indeholdende komponentdetaljer vedrørende komponenten med ID 1. Vi har ikke nogen indflydelse på hvor mange informationer som der kommer med når vi har valgt metoden *getsensordetails*, så vi bliver i koden nød til at udvælge de data vi ønsker. Derimod kan vi angive ekstra egenskaber til HTTP kaldet, som f.eks. egenskaben count, som gør os i stand til at hente op til 500 komponenters detaljerede oplysninger. Dette benytter vi os af i Mediatoren, når vi skal liste alle PRTG's komponenter i GUI'en.

Det er muligt at sende data til et REST API, men da vi ikke har brug for at sende data til PRTG's API, så benytter vi os ikke af det. Det gør vi derimod i næste kapitel som omhandler kommunikation med Infrastructure Diagnosis.

### 5.7.2 Infrastructure Diagnosis

Infrastructure Diagnosis API er opbygget efter SOAP standarden. Vi har ligesom til PRTG, brug for at kende til adressen på serveren som stiller API'et til rådighed, et brugernavn og et password. Dog har vi i vores løsning ikke benyttet os af brugernavn og password. For at skabe kontakt til API'et sætter vi nedenstående adresse ind i en browser:

http://buffynix.dyndns.org/Services/DataAPIService.svc

Figur 55 – Adresse til Infrastructure Diagnosis API

Vi ser her en side hvor API'et stiller en wsdl fil til rådighed for dem der måtte ønske den. En wsdl fil indeholder oplysninger omkring hvilke funktioner som API'et stiller til rådighed og hvilken type objekt som hver funktion forventer og benytter sig af.

I Visual Studio har vi muligheden for at tilføje en såkaldt Service Reference. Ved at gøre det, genere Visual Studio, på baggrund af wsdl filen, en client til API'et som vi så kan benytte til at kommunikere med API'et.

Vores kommunikation med Infrastructure Diagnosis API'et er begrænset til to funktioner, nemlig GetSystemsFromWebservice og InsertDataInput. I begge funktioner opretter vi et klient objekt af typen DataAPIServiceClient, koden til dette er illustreret i Figur 56 nedenfor. Vi angiver en ny EndpointAddress med den adresse vi har fra vores konfigurationsfil. Dette er strengt taget ikke nødvendigt da Visual Studio som default værdi har indsat den adresse som den generede klienten ud fra. Vi har implementeret manuel adresse angivelse, da vi så ville kunne benytte den samme klient til flere forskellige Mediators ved blot at ændre adressen.

```
EndpointAddress wsAddress = new EndpointAddress(Host);
InfrastructureDiagnosisService.DataAPIServiceClient wsClient = new DataAPIServiceClient();
wsClient.Endpoint.Address = wsAddress;
```

Figur 56 – Oprettelse af klient objekt

Vi kan nu via wsClient objektet benytte API'et som var det en indbygget klasse i vores projekt. I Figur 57 nedenfor er et eksempel på hvor nemt det er at benytte en funktion i API'et. API'et stiller blandt andet funktionen InsertDataInput til rådighed, og alt hvad vi behøver for at benytte os af den er at bidrage med den rigtige inputdatatype. Denne funktion giver os, som navnet indikerer, mulighed for at indsende data til vores API.

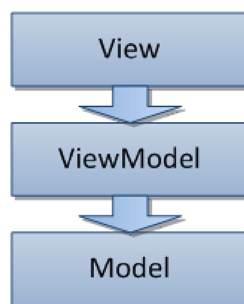
```
public void InsertDataInput(DataInputDTO dataInput)
{
   EndpointAddress wsAddress = new EndpointAddress(Host);
   InfrastructureDiagnosisService.DataAPIServiceClient wsClient = new DataAPIServiceClient();
   wsClient.Endpoint.Address = wsAddress;

   wsClient.InsertDataInput(dataInput);
}
```

Figur 57 – Funktionen InsertDataInput

På samme måde kan vi benytte os af alle de funktioner som API'et stiller til rådighed. Så forskellen mellem et SOAP og REST baseret API, er tilgangen. Et REST baseret API kræver en form for dokumentation for at brugeren kan tilegne sig et kendskab til funktionaliteterne som der stilles til rådighed, men efter at have læst den vil brugeren let kunne tilpasse kald efter præference. Kaldende er lette at foretage, da det både kan gøres i kode, men reelt også manuelt i en browser, hvilket giver stor fleksibilitet. SOAP API'et skal på den anden side tilgås med en klient fra et udviklingsmiljø, så det er ikke ligeså let tilgængeligt som REST API'et er, dog er brugen af det relativt let, da SOAP klienten kan auto generes i de fleste sprog.

## 5.8 Mediator GUI og model binding

Mediatorens GUI er bygget op efter design mønsteret Model-View-ViewModel, forkortet MVVM.




Figur 58 - MVVM

Ved at benytte os af design mønsterets retningslinjer, opnår vi en simpel men effektiv opdeling af koden. MVVM ligger op til at man separerer koden op i tre lag, view laget, model laget og viewmodel laget, illustreret med Figur 58 (s. 66). Model laget repræsenterer data lagret persistent, og har ikke noget kendskab til de to andre lag. Viewmodel har kun kendskab til modellaget, men kan stille datavisningsmodeller, også kaldet viewmodels, til rådighed for de der ønsker at benytte sig af dem. View har kun kendskab til de viewmodels som view laget stiller til rådighed for den. I view knytter vi vores objekter til viewmodels ved hjælp af model bindings. Det er en funktionalitet som WPF stiller til rådighed for os, men som er forberedt til brug i forbindelse med MVVM. En viewmodel kan se ud som illustreret på Figur 59

nedenfor. Vi har her i eksemplet at gøre med en viewmodel der skal benyttes i forbindelse med visning af PRTG komponenter i GUI'en. Vi indhenter her de tre egenskaber ID, Name og Type, og generer med dem egenskaben ListName.

```
class PrtgSensorViewModel
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Type { get; set; }
    public string ListName { get; set; }
    public string Icon { get; set; }

    public PrtgSensorViewModel(PrtgSensor domainModel)
    {
        ID = domainModel.ID;
        Name = domainModel.Items["device"];
        Type = domainModel.Items["sensor"];
        ListName = Name + "." + Type;

        Icon = "node.png";
    }
}
```

Figur 59 – Eksempel på en viewmodel

Hvert PrtgSensorViewModel repræsenterer en komponent i PRTG, hvilke vi ønsker at liste i en listboks i GUI'en. For at gøre det læsbart for brugeren ønsker vi at repræsentere hver viewmodel med navn og type, og til dette har vi lavet egenskaben ListName.

For at få skabt tilknytningen mellem view lag og viewmodel, skal vi først i viewmodellen angive hvilket objekt vi ønsker at skabe en tilknytning til, i dette tilfælde listboksen lstPrtg. Dette gøres ved at tilføje følgende linje kode i viewmodellen.

```
lstPrtg.ItemsSource = prtgViewModels;
```

Figur 60 – Binding af objekt til viewmodel

Vi kan nu i koden til GUI'en, xaml'en, angive hvilken egenskab vi ønsker at repræsentere hver af vores viewmodel i listboksen med. Det gøres ved at angive `Content="{Binding ListName}"` i koden, som også er illustreret i Figur 60 nedenfor.

```
<ListBox Grid.Row="1" Name="lstPrtg">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Image Source="{Binding Icon}" Height="16" Width="16"></Image>
                <Label VerticalAlignment="Center" Content="{Binding ListName}" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

**Figur 61 – Eksempel på model binding**

Vi har nu sørget for at kunne vise data i vores view lag uden at ændre på noget i vores data model. I viewmodel laget sørger vi for at præsenterer den data som vi ønsker, skal være tilgængelig for view laget, og i view laget bestemmer vi selv hvordan data fra viewmodellen skal præsenteres. Det skaber en lav kobling mellem lagene og gør det nemt at vedligeholde.

# 6 Test

Vi vil i test kapitlet først teste om vores løsning opfylder de funktionelle krav stillet i kapitel 0 (s. 15). I kravspecifikationen introducerede vi totalt fem funktionaliter

- *Konsekvensanalyse.*
- *Systemdokumentation.*
- *Statusrapportering.*
- *Konfiguration.*
- *Import af live data.*

Vi vil disse test også undersøge om vi har levet op til vores ikke-funktionelle krav stillet i kapitel 2.7.1 (s. 16). Vi opstillede her følgende fire krav til vores løsning:

- Fokus på ikke fungerende *komponenter*.
- Nem og intuitiv brugergrænseflade.
- Simpel opbygning, kun den vigtigste information vises.
- Løsningen skal fungere optimalt med et it system bestående af op til 250 forskellige *komponenter*.

Da vi har opstillet et performance krav i det sidste punk i de ikke-funktionelle krav, har vi at opstille et test *system*. Denne opstilling består af følgende komponenter:

- 250 servere
- 25 services
- 5 network links

Hver service omfatter 10 servere og imellem servere og netværkslinks er der konfigureret forskellige afhængigheder.

Det er med udgangspunkt i denne opstilling at følgende tests er foretaget.

## 6.1 Konsekvensanalyse

Konsekvensanalysen består af en konfigurationsdel og et beregnet resultat som udgør selve analysen, så det er hvad testen består af.

| Konsekvensanalyse, Konfiguration | OK |
|---|---|
| Konsekvensanalyse, Resultat | OK |
| Ikke-funktionelle krav | OK |
| | |
| Bemærkning: | Med 280 komponenter bliver listen ved konfiguration naturligvis lang, hvorved der skal scrolles en del. Ligeledes kan |

| | resultatet af konsekvensanalysen blive særdeles lang hvis alle *komponenter* er involveret. Hastighedsmæssigt, så er der intet at udsætte, løsningen svarer hurtigt på forespørgsler. |
|---|---|

## 6.2 Systemdokumentation

Systemdokumentationen består af en grafdel og en menu del, hvor man kan tilgå komponenternes detaljer.

| Systemdokumentation, Graf | OK |
|---|---|
| Systemdokumentation, Menu | OK |
| Ikke-funktionelle krav | FEJL |
| | |
| Bemærkning: | Systemdokumentationen virker funktionalitetsmæssigt efter hensigten. Hastighedsmæssigt, så har graffunktionaliteten svært ved at håndtere de 280 komponenter, som resulterer i at grafen opdateres meget langsomt. Visse visningsstrategier resulterer i at browserens flash plugin bryder sammen. |

## 6.3 Statusrapportering

Statusrapporteringen består kun af en del, hvilket er visning af systemets ikke funktionelle services.

| Statusrapportering | OK |
|---|---|
| Ikke-funktionelle krav | OK |
| | |
| Bemærkning: | Statusrapportering virker funktionalitetsmæssigt og selv med alle 25 services nede, er det let at overskue dem. |

## 6.4 Konfiguration

I konfigurationstesten er der foretaget test af oprettelse, ændring og sletning af alle typer komponenter i løsningen. Yderligere er der knyttet bemærkninger til de enkelte typer komponenter.

| Funktionalitet | Test | Resultat |
|---|---|---|
| System | Oprettelse | OK |
| System | Ændring | FEJL |
| System | Sletning | FEJL |
| Bemærkning | Oprettelse: Feltet navn og beskrivelse skal udfyldes.<br>Ændring: Der kan ikke ændres på et systems grundoplysninger.<br>Sletning: Det er ikke muligt at slette et system. | |
| Infrastruktur | Oprettelse | OK |

| Infrastruktur | Ændring | FEJL |
|---|---|---|
| Infrastruktur | Sletning | FEJL |
| Bemærkning | Oprettelse: Feltet navn og beskrivelse skal udfyldes.<br>Ændring: Der kan ikke ændres på en infrastrukturs grundoplysninger.<br>Sletning: Det er ikke muligt at slette en infrastruktur. | |
| Komponent | Oprettelse | OK |
| Komponent | Ændring | OK |
| Komponent | Sletning | FEJL |
| Bemærkning | Oprettelse: Feltet navn og beskrivelse skal udfyldes.<br>Ændring: Feltet navn og beskrivelse skal være udfyldt.<br>Sletning: Det er ikke muligt at slette en komponent. | |
| Afhængighed | Oprettelse | OK |
| Afhængighed | Ændring | OK(!) |
| Afhængighed | Sletning | OK |
| Bemærkning | Oprettelse: Linktype og entitet skal være valgt.<br>Ændring: En afhængighed kan/skal ikke kunne ændres. Den skal derimod slettes og oprettes på ny.<br>Sletning: Intet at bemærke. | |
| Bruger | Oprettelse | OK |
| Bruger | Ændring | OK |
| Bruger | Sletning | OK |
| Bemærkning | Oprettelse: Navn og email skal udfyldes.<br>Ændring: Navn og email skal være udfyldt.<br>Sletning: Intet at bemærke. | |
| Komponent type | Oprettelse | OK |
| Komponent type | Ændring | OK |
| Komponent type | Sletning | OK |
| Bemærkning | Oprettelse: Navn skal udfyldes.<br>Ændring: Navn skal være udfyldt.<br>Sletning. Intet at bemærke. | |
| Datainput type | Oprettelse | OK |
| Datainput type | Ændring | OK |
| Datainput type | Sletning | OK |
| Bemærkning | Oprettelse: Navn, datatype og system skal udfyldes og vælges<br>Ændring: Navn, datatype og system skal være udfyldt og valgt<br>Sletning: Intet at bemærke. | |

## 6.5 Import af live data

Funktionaliteten Import af live data ligger i løsnings API. Funktionen hedder InsertDataInput, og kræver at man indsender tre argumenter, data, datatypeid og enhedsid. Hvis man intet kendskab har til denne information, så tilbyder API'et fire andre funktioner, som gør at man kan indhente oplysninger omkring systemer, infrastrukturer, komponenter og datatyper. API'et er blevet testet ved hjælp af en test

klient der er indbygget i Visual Studio ved navn wcftestclient.exe. GetSystems tager ikke nogle argumenter, og returnere blot de oprettede systemer, inklusiv egenskaber. Med systemID'et er det muligt at indhente infrastruktur oplysninger og med infrastructureID er et muligt at indhente komponenter oplysninger. Alle disse funktioner kan benyttes til at bestemme hvilken komponent, i hvilken infrastruktur, i hvilket system der skal opdateres. Vi har i testen valgt at opdatere komponenten 'Server 250' med skiftesvis *Up* og *Down* tilstande. Resultatet af testen er som følger:

| Funktion | Resultat |
|---|---|
| GetSystems | OK |
| GetInfrastructures | OK |
| GetEntities | OK |
| GetDataInputTypes | OK |
| InsertDataInput | OK |

Vi var i stand til at indhente de informationer vi skulle bruge for at kunne opdatere en *komponent*. Vi valgte at opdatere komponenten med navnet 'Server 250' og ID 58060 med skiftesvis *Up* og *Down* tilstande, hvilket virkede efter hensigten.

## 6.6 Test konklusion

Løsning opfylder funktionalitetskravene selvom der eksistere områder der kunne have været optimeret yderligere. Konfigurationstesten afslører at der er noget simpel basis funktionalitet der ikke er blevet implementeret. Dette skyldes at vi prioriterede at få en brugbar prototype op og stå, sådan at vi havde noget at bygge ud fra. Vi har under udviklingen naturligvis testet løbende, og har i den forbindelse haft et reset script, som tømte databasen helt og lagde et nyt system på igen, og vi har derfor ikke haft behov for at slette f.eks. et system via GUI.

Løsning opfylder de ikke-funktionelle krav, med en enkelt undtagelse som omhandler systemdokumentationsgrafen. Grafen virker, men dog relativt langsomt og som tidligere nævnt, så kan enkelte visningsstrategier få flash plugin'et til gå ned.

Vil har igennem hele vores udvikling proces, ikke implementeret input validering i større omfang. Vi har dog ved funktioner hvor input data ikke er indlysende, valgt at implementere input validering, da risikoen for at indtaste forkert input var stor, kriterium opsætning er et eksempel herpå.

Undervejs i udviklingen har vi funktionalitetstestet vores delkomponenter. Vi har typisk testet vores delkomponenter op imod den række af input som den kunne forvente at modtage. Vi har ikke testet med input der ikke var hensigtsmæssige at benytte sig af. Dette valg er truffet da dette projekt, primært handler om at vise konceptuelle muligheder og i mindre grad om at levere et produktionsmodent produkt.

# 7 Konklusion

Er system til synliggørelse og håndtering af associationer mellem komponenter i firmaets it-system er blevet udviklet. Systemet indeholder mulighed for at analysere på systemets tilstand både dynamisk i forhold til løbende dynamisk statusinformation, samt statisk analyse på baggrund af tænkte scenarier i konsekvensanalysen. I forbindelse med grafisk fremstilling af associationerne mellem komponenterne i it-systemet har vi stiftet bekendtskab med Kalileo Diagrammer som har givet os nogle udfordringer som vi havde håbet på at overkomme. Det indebærer muligheden for at foretage editering af afhængighedsgrafen via en grafisk editor, hvilket vi ikke nåede at implementere fordi det viste sig at blive for stor en opgave inden for projektets rammer. Det har vist sig at det ville have været ønskværdigt at den grafiske visualisering havde haft en indbygget hukommelse, sådan at den enkelte bruger kunne have valgt at gemme sin egen opstilling af grafen.

Til den webbaserede visning gjorde vi brug af SignalR hvilket har vist sig at blive en stor succes. Den automatisk opdaterende grafiske brugerflade viste sig at have en langt større værdi end vi først havde antaget og vi endte med at bruge en stor mængde tid på at udforske og designe systemet til at understøtte brugen af dette.

Til fortolkning af boolske udtryk gjorde vi med fordel brug af ANTLR parser generator, da vi her havde mulighed for at beskrive grammatikken via EBNF. Det har også fungereret efter hensigten og produceret rigtig gode resultater. Det havde dog sikkert været muligt at implementere en simplere parser af de relativt simple udtryk, på kortere tid, hvilket vi kunne have draget fordel af. På den anden side har vi måske sparet en mængde test af en hjemmelavet løsning og vi er derfor ikke utilfredse med at have valgt den komplicerede løsning via ANTLR.

Implementeringen af den webbaserede brugerflade bærer præg af at være en prototype på hvordan et faktisk system kunne se ud. Det viser sig ved at der ikke nødvendigvis er gennemført konsistent navngivning og inputvalidring på alle systemets funktioner. I forhold til det problem som det var vores oprindelige målsætning at systemet løse, er vi dog selv at den overbevisning at den nuværende implementering sagtens kan anskueliggøre gevinsten ved systemets hovedfunktioner. En oplagt videreudvikling, som vi selv er kommet til at savne, er muligheden for, i konsekvensanalysen, at simulere en komplet omstrukturering af systemets sammensætning, fremfor kun at kunne simulere en antaget *Down* tilstand på en række komponenter.

# Appendiks

# A. Database model

# B. Håndbog

Dette er håndbogen til brug af Infrastructure Diagnosis applikationen. I denne håndbog vil du finde en udførlig vejledning i hvordan du benytter dig af programmets muligheder.

For at benytte løsningen skal du åbne en browser og skrive http://buffynix.dyndns.org/ i adressefeltet. Vi anbefaler at siden åbnes i browseren Mozilla Firefox, som kan hentes her.

## Opsætning

Dette kapitel vil omhandle førstegangs opsætning af programmet Infrastructure Diagnosis.

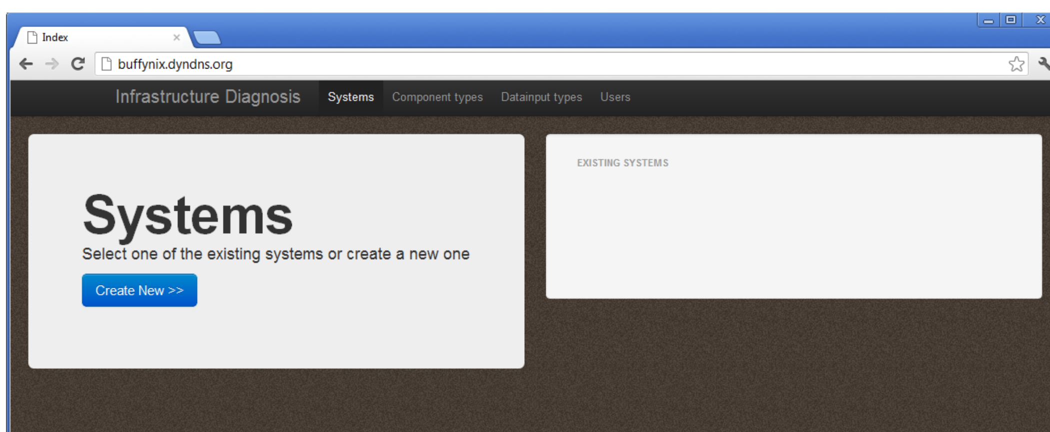
Kapitlet vil fører brugeren igennem en standard opsætning af programmet, hvorefter brugeren vil være rustet til selv at foretage en opsætning.

Opsætningen vil omfatte følgende opgaver:

- Oprettelse af system
- Oprettelse af infrastruktur
- Oprettelse af komponent typer
- Oprettelse af komponenter
- Oprettelse af afhængigheder

## Oprettelse af system

Brugeren tilgår systemet og befinder sig nu på forsiden, som vist nedenfor med Billede 5. På forsiden er det muligt at oprette nye systemer, eller kigge nærmere på eksisterende systemer. Da vi ønsker at oprette et nyt system, trykker vi på knappen *Create new*.

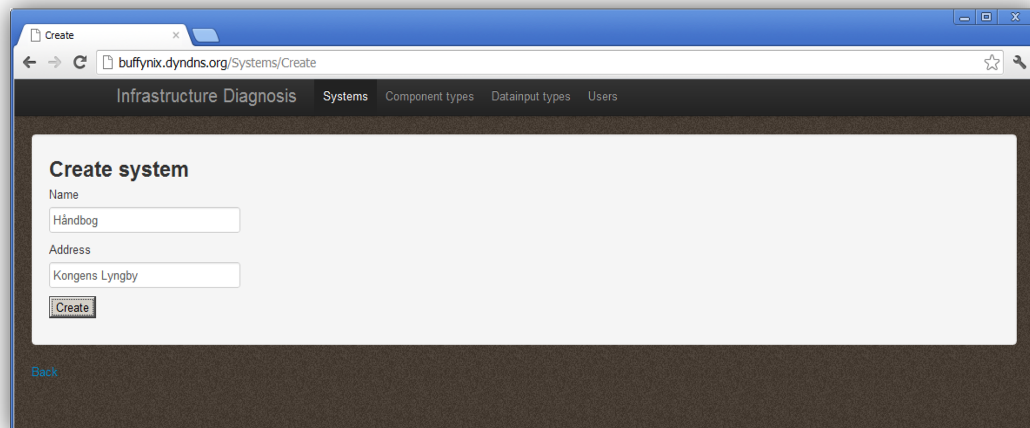



Billede 5 - Forside

Brugeren mødes nu af en side, hvor der skal indtastes oplysninger til brug i forbindelse med oprettelse af et nyt system, som vist på Billede 6 nedenfor. Oplysningerne der ønskes er:

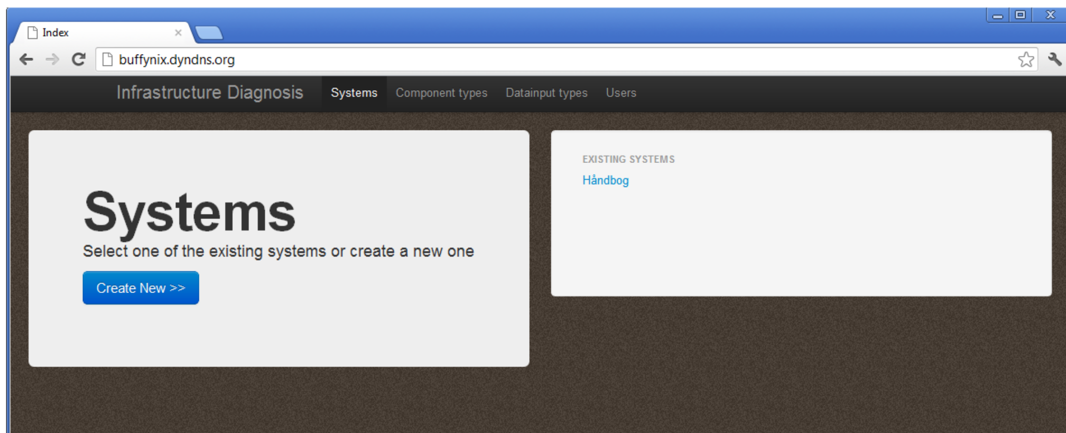
- Navn
- Adresse

For eksemplets skyld udfylder vi "Håndbog" i feltet *Name* og "Kongens Lyngby" i feltet *Address*, hvorefter vi trykker på knappen *Create* for at oprette systemet




Billede 6 – Oprettelse af nyt System

Efter oprettelse af systemet føres brugeren tilbage til forsiden. Af forsiden fremgår nu det system der lige er blevet oprettet, som vist på Billede 7 nedenfor.

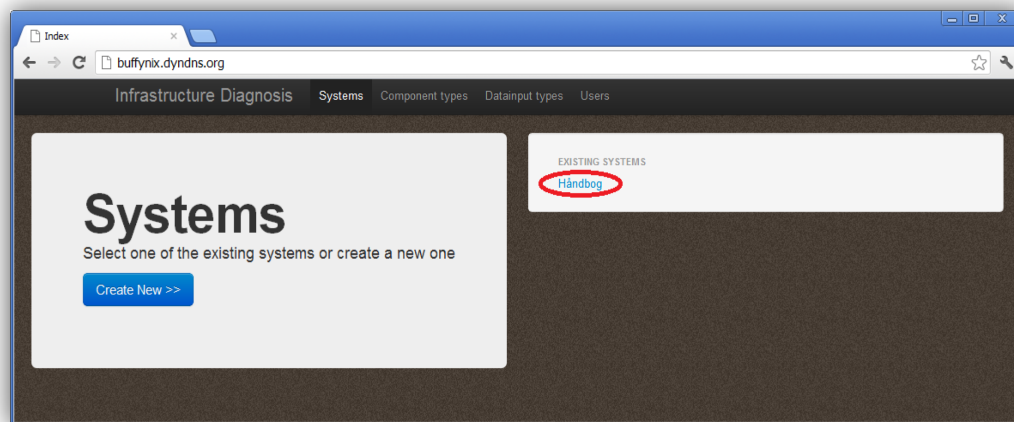



Billede 7 – System oprettet

## Oprettelse af infrastruktur

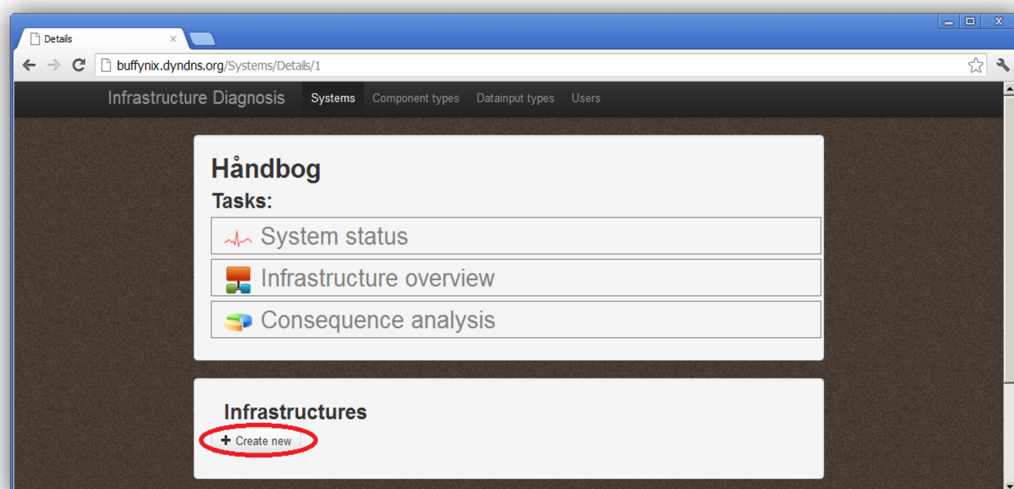
*Det er en forudsætning at der er i programmet, Infrastructure Diagnosis, er oprettet mindst et system, for hvilken man ønsker at oprette en infrastruktur.*

Brugeren tilgår programmet og bliver mødt af forsiden. Vi har tidligere oprettet systemet Håndbog, hvilken vi tager udgangspunkt i dette eksempel. For at oprette en infrastruktur trykker vi på systemet *Håndbog*, som er markeret med en rød ring som vist på Billede 8 nedenfor.




Billede 8 – Forside med oprettet system

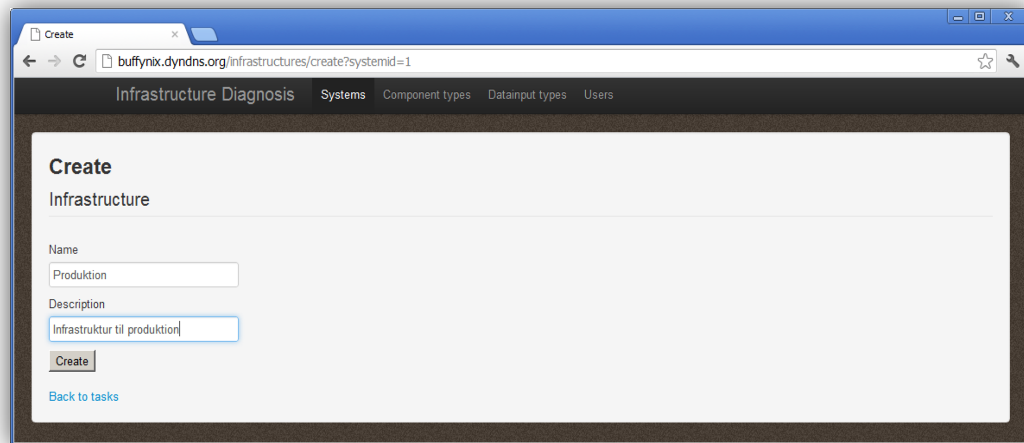
Brugeren tilgår nu oversigtsbillede for systemet *Håndbog*. For at oprette en infrastruktur trykker vi på knappen *Create new* i sektionen *Infrastructures*, som vist med en rød ring nedenfor på Billede 9.




Billede 9 – Oversigtsbillede af systemet Håndbog

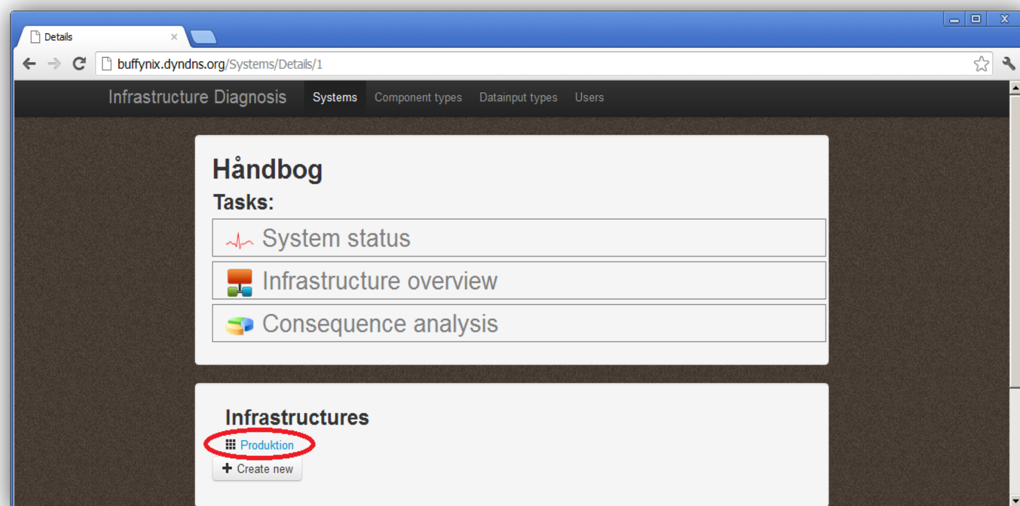
Brugeren mødes nu af en side, hvor der skal indtastes oplysninger til brug i forbindelse med oprettelse af en ny infrastruktur, som vist på Billede 10 nedenfor. Oplysningerne der ønskes er: Navn, Beskrivelse.

For eksemplets skyld udfylder vi "Produktion" i feltet *Name* og "Infrastruktur til produktion" i feltet *Description*, hvorefter vi trykker på knappen *Create* for at oprette infrastrukturen.




Billede 10 – Oprettelse af infrastruktur

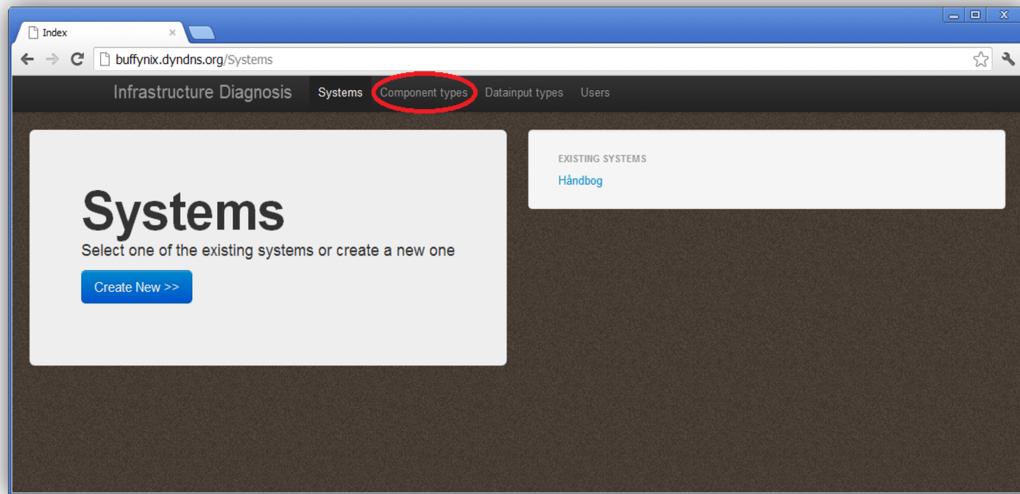
Efter oprettelse af ny infrastruktur sendes brugeren tilbage til oversigtsbilledet for systemet *Håndbog*. Her fremgår den nye infrastruktur *Produktion*, nu af listen over infrastrukturer, som det er forsøgt vist med en rød ring på Billede 11 nedenfor.




Billede 11 – Oversigtsbillede over systemet Håndbog

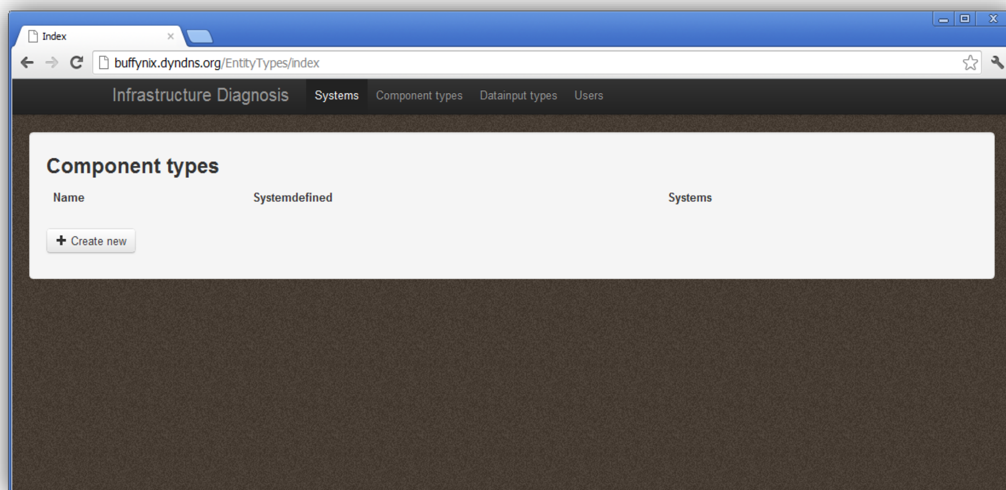
## Oprettelse af komponent type

*Det er en forudsætning at der er i programmet, Infrastructure Diagnosis, er oprettet mindst et system, til hvilken man ønsker at oprette komponent typer.*



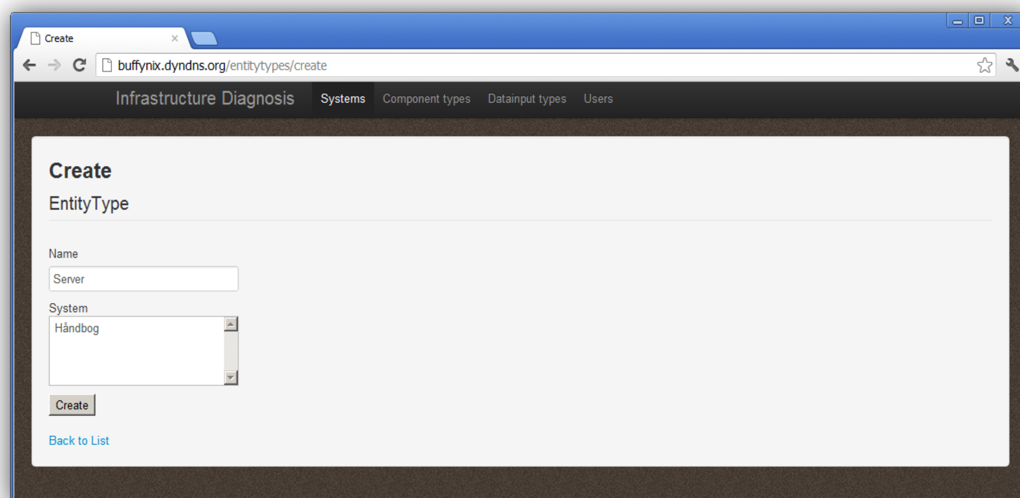

Billede 12 - Forside

Brugeren tilgår programmet og bliver mødt af forsiden. Vi har tidligere oprettet systemet Håndbog, hvilken vi tager udgangspunkt i dette eksempel. For at oprette komponent typer trykker vi på linket *Component types*, som er placeret i top menuen. Vi har på Billede 12 ovenfor markeret linket med en rød ring.




Billede 13 – Oversigt over komponent typer (tom)

Vi befinder os nu på oversigtsbilledet over komponent typer, som vist ovenfor på Billede 13. I oversigten kan vi se at der ikke er oprettet nogen komponent typer. For at oprette en ny komponent type, trykker vi på knappen *Create new*.




Billede 14 – Oprettelse af komponent type

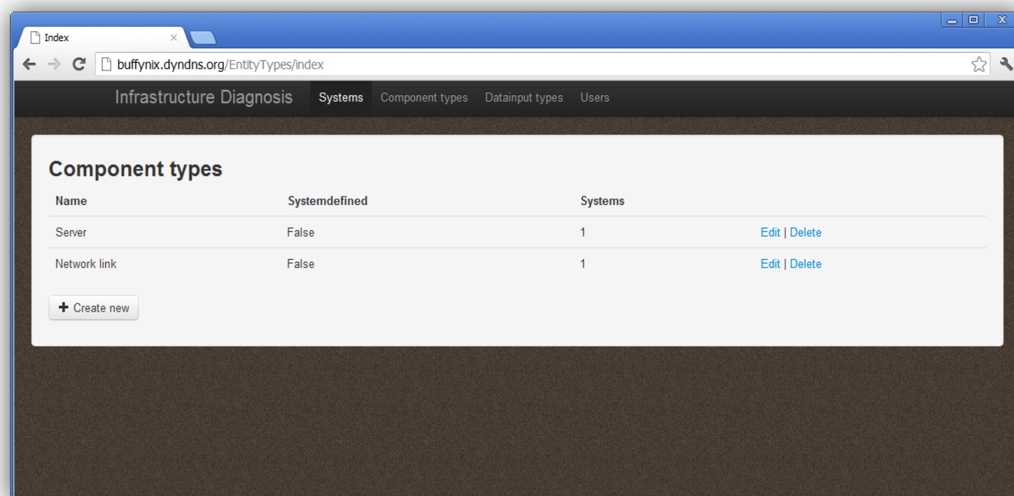
Brugeren mødes nu af en side, hvor der skal indtastes oplysninger til brug i forbindelse med oprettelse af en ny komponent type, som vist på Billede 14 ovenfor. Oplysningerne der ønskes er:

- Navn
- System

For eksemplets skyld udfylder vi "Server" i feltet *Name* og vælger systemet "Håndbog" i listen af systemer, hvorefter vi trykker på knappen *Create* for at oprette komponent typen.

Efterfølgende opretter vi endnu en komponent type med navnet "Network link" og knytter den til systemet "Håndbog.

Af oversigten, vist på Billede 15 nedenfor, kan vi nu se at de to komponent typer vi netop har oprettet.

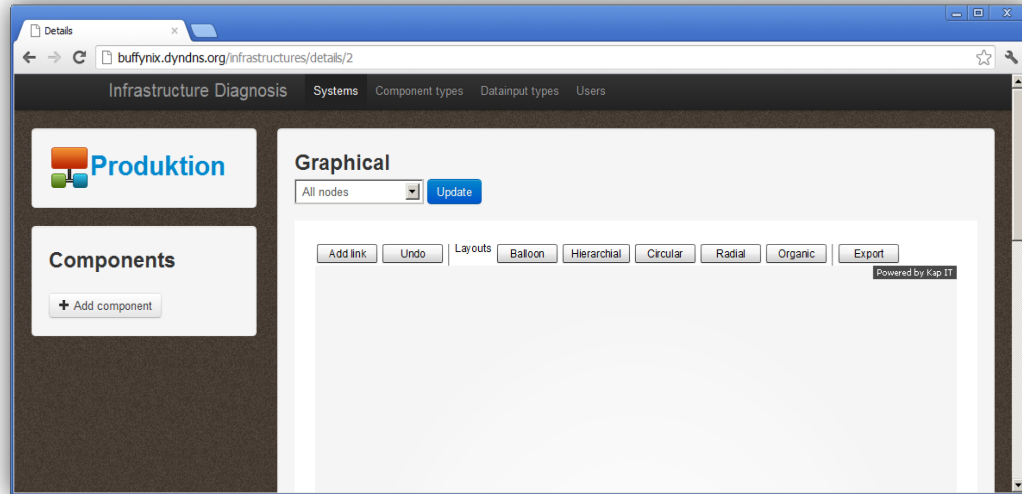



Billede 15 – Oversigt over komponent typer

## Oprettelse af komponent

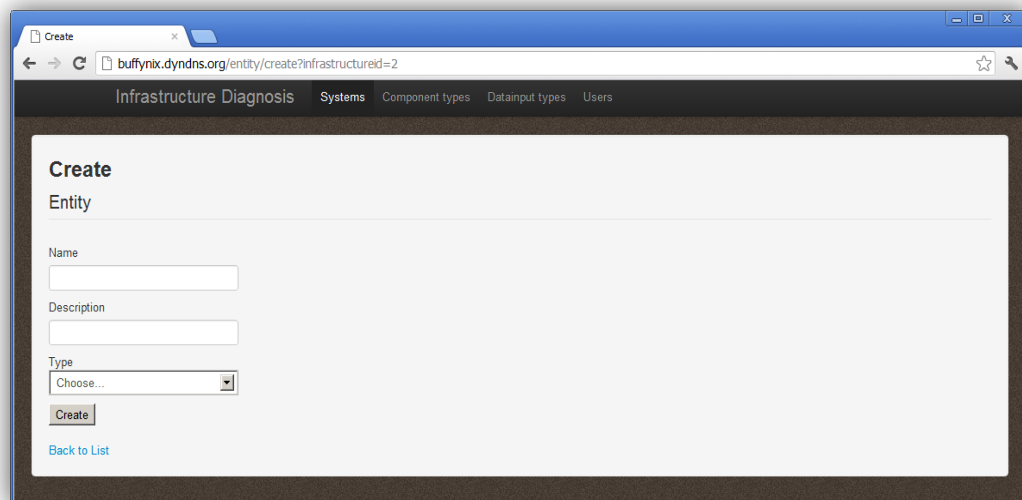
*Det er en forudsætning at der er i programmet, Infrastructure Diagnosis, er oprettet mindst et system og en infrastruktur, for hvilken man ønsker at oprette komponenter.*

Brugeren skal indfinde sig på oversigten for infrastrukturen Produktion, vist nedenfor med Billede 16, hvilket fra forsiden findes ved at benytte linket til systemet *Håndbog* og derfra benytte linket til infrastrukturen *Produktion*.




Billede 16 – Oversigt over Infrastruktur

I oversigten vises til venstre i billedet, en liste over oprettede komponenter i listen *Components*. Vi trykker, for at oprette en ny komponent, på knappen *Add component*, hvorefter vi bliver ført til siden "Opret komponent", vist på Billede 17 nedenfor.




Billede 17 – Opret komponent

Brugeren befinder sig nu på siden "Opret komponent", hvor der skal indtastes oplysninger til brug i forbindelse med oprettelse af en ny komponent, som vist på Billede 17 ovenfor. Oplysningerne der ønskes er:

- Navn
- Beskrivelse
- Type

Vi opretter for eksemplets skyld fem komponenter, hvoraf den første oprettes med følgende informationer. I feltet *Name* udfylder vi "AD Server", i feltet *Description* udfylder vi "Active Directory" og *Type* sætter vi til "Server". Herefter trykker vi på knappen *Create*, for at oprette komponenten. Vi opretter yderligere fire komponenter med følgende oplysninger:

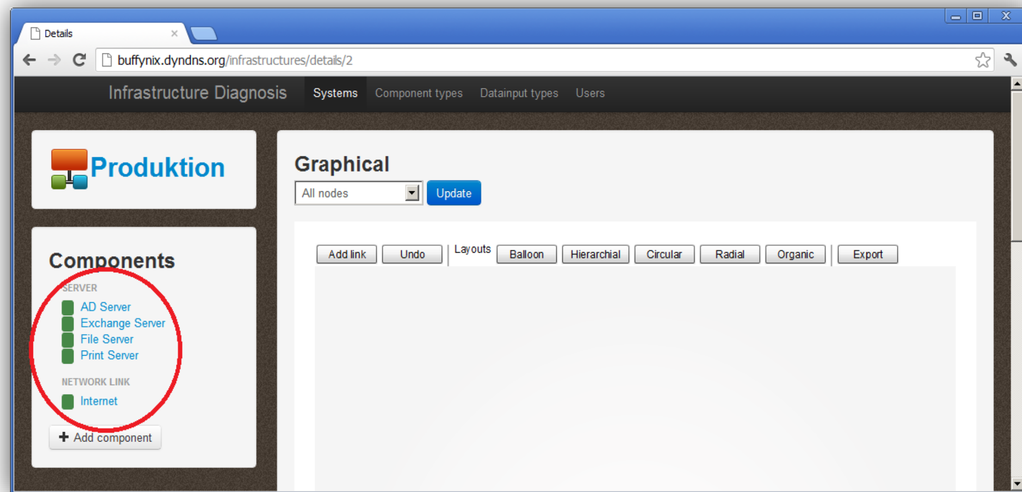
Komponent 2:
Name: Exchange Server
Description: Microsoft Exchange.
Type: Server

Komponent 3:
Name: File Server
Description: File Server.
Type: Server

Komponent 4
Name: Print Server
Description: Print Server.
Type: Server

Komponent 5:
Name: Internet
Description: Network connection with Internet
Type: Network link

Efter oprettelse af den sidste component befinder vi os ved oversigten af infrastrukturen *Produktion*. Vi kan nu af listen i vestre side, se at de fem oprettede komponenter fremgår, som det ses på Billede 18 nedenfor.

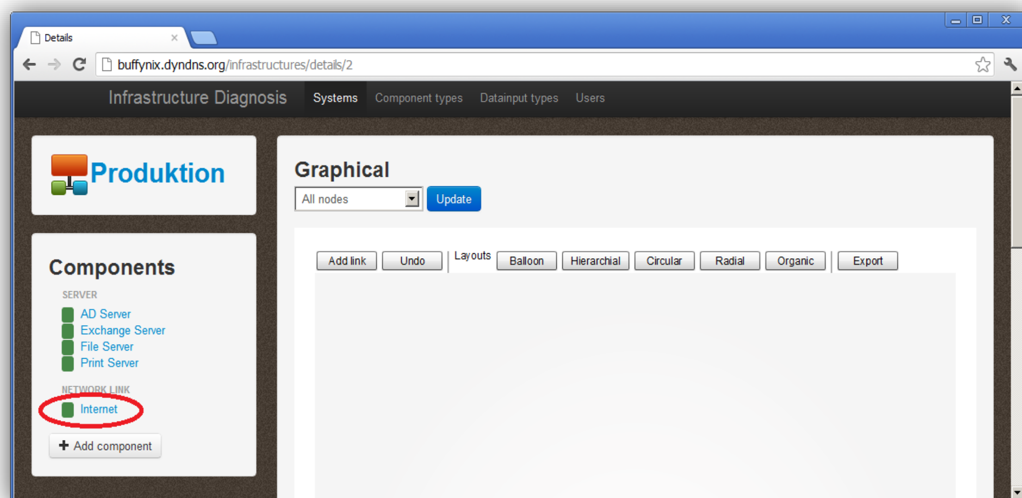



Billede 18 – Oversigt over infrastrukturen produktion (fem komponenter oprettet)

## Oprettelse af afhængighed

*Det er en forudsætning at der er i programmet, Infrastructure Diagnosis, er oprettet mindst et system, en infrastruktur og mindst to komponenter, for mellem hvilke man ønsker at oprette en afhængighed.*

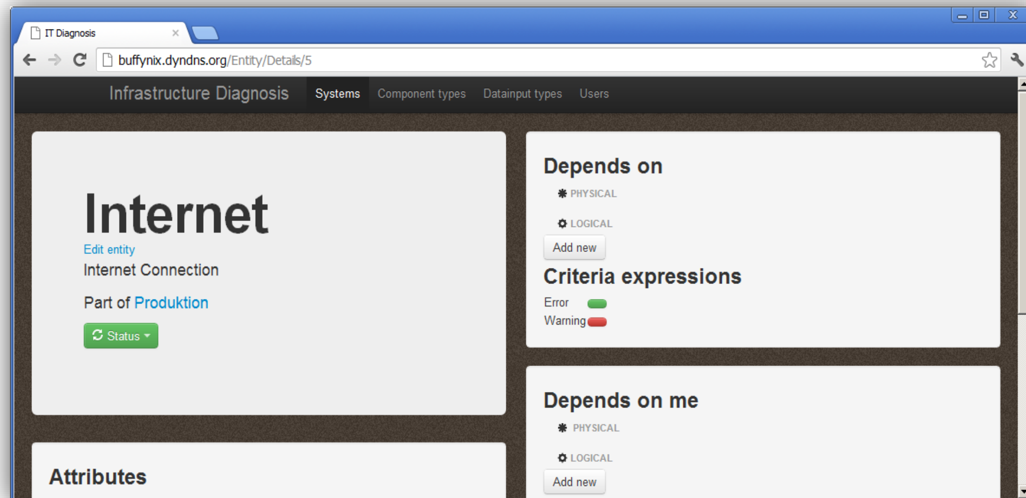
Brugeren skal for at kunne oprette afhængigheder, indfinde sig på oversigten for infrastrukturen Produktion, vist nedenfor på Billede 19, hvilket fra forsiden findes ved at benytte linket til systemet *Håndbog* og derfra benytte linket til infrastrukturen *Produktion*.




Billede 19 – Oversigt over infrastrukturen produktion, med fokus på komponent Internet

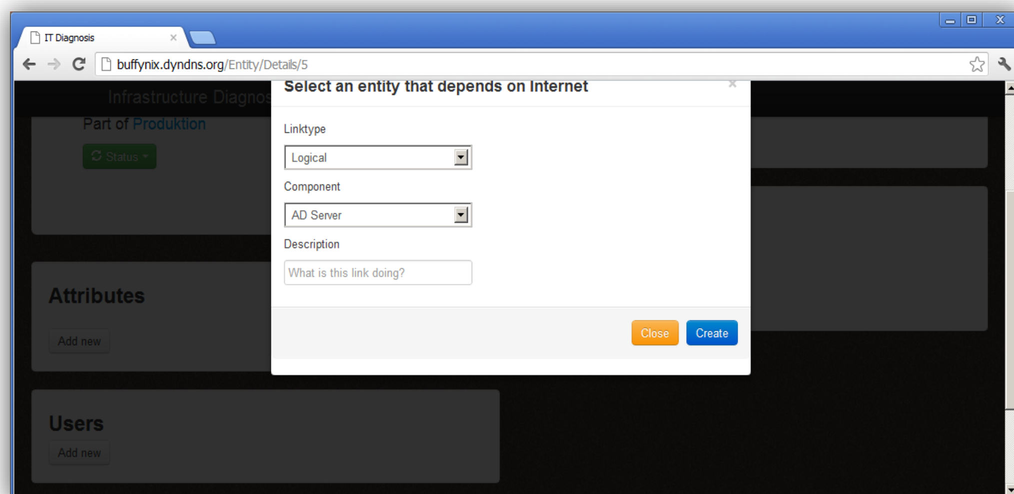
Fra Billede 19, som ses ovenfor, klikker vi for at oprette en afhængighed, på en af de to komponenter som afhængigheden skal oprettes i mellem. For eksemplets skyld

klikker vi på komponenten *Internet* og vi bliver ført videre til oversigten for komponenten Internet, som kan ses nedenfor på Billede 20.




Billede 20 – Oversigt over komponenten *Internet*.

Brugeren befinder sig nu på oversigtsbilledet for komponenten *Internet*. Herinde er det muligt i højre side af billedet at se hvilke komponenter der er afhængig af denne komponent og hvilke komponenter som denne komponent af afhængig af. For at oprette en ny afhængighed trykker vi på knappen *Add new* i sektionen *Depends on me*.




Billede 21 – Opret afhængighed

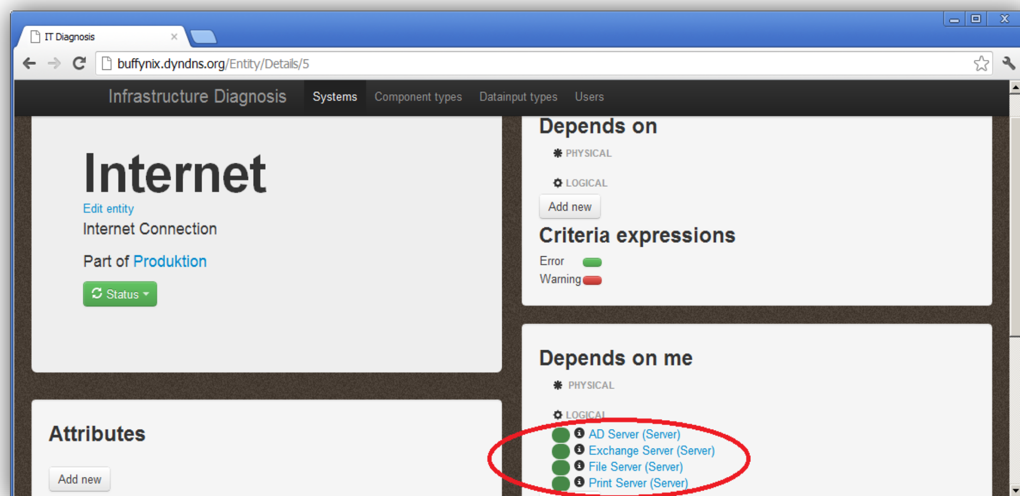
Brugeren befinder sig nu på siden "Opret afhængighed", hvor der skal indtastes oplysninger til brug i forbindelse med oprettelse af en ny afhængighed, som vist på Billede 21 ovenfor. Oplysningerne der ønskes er:

- Link type
- Komponent
- Beskrivelse

Vi opretter for eksemplets en afhængighed med følgende informationer. I feltet *Linktype* vælger vi vi "Logical", i feltet *Component* vælger vi "AD Server" og feltet *Description* efterlader vi blankt. Herefter trykker vi på knappen *Create*, for at oprette komponenten. Vi opretter yderligere tre komponenter, med samme fremgangsmåde, med følgende oplysninger:

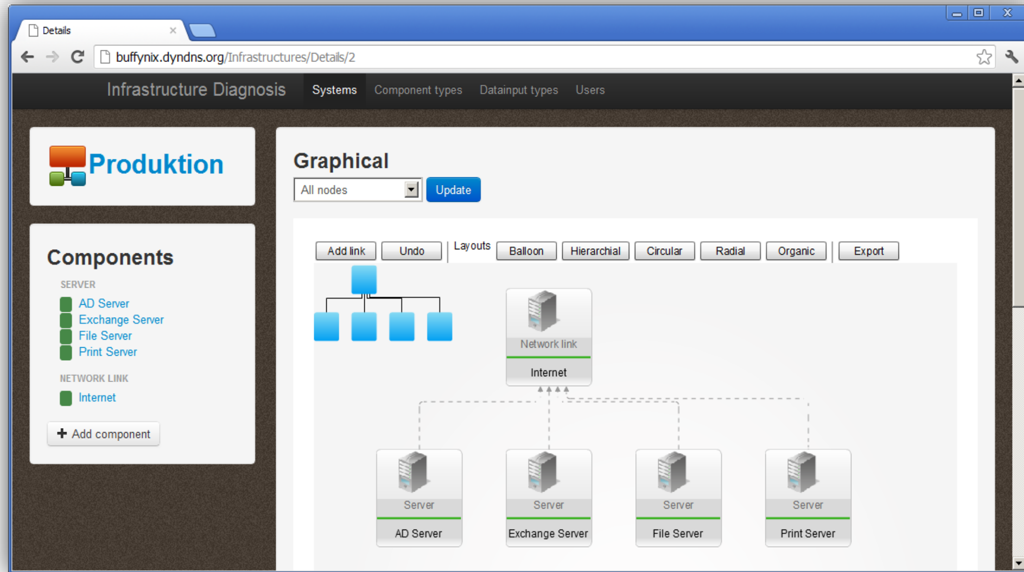
File Server *er afhængig af* Internet
Exchange Server *er afhængig af* Internet
Print Server *er afhængig af* Internet

Efter oprettelse af den sidste afhængighed bliver vi sendt tilbage til komponenten *Internets* oversigtsbillede, som fremgår af Billede 22 nedenfor. Her kan vi i sektionen *Depends on me* se at de fire oprettede afhængigheder nu fremgår af listen.




Billede 22 – Oversigt over komponenten Internet, med fire oprettede afhængigheder.

Yderligere fremgår komponenterne med afhængigheder nu af grafen i oversigtsbilledet af infrastrukturen *Produktion*, som det kan ses på Billede 23 nedenfor.

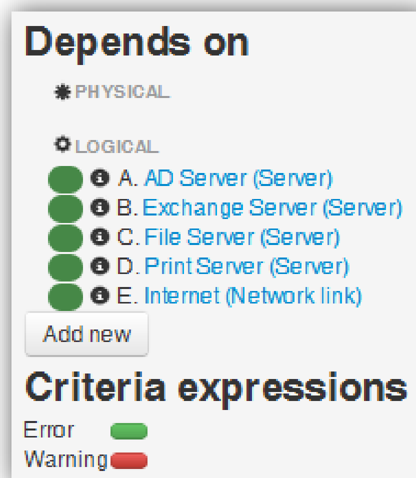



Billede 23 – Oversigt over infrastrukturen *Produktion*.

## Oprettelse af kriterie

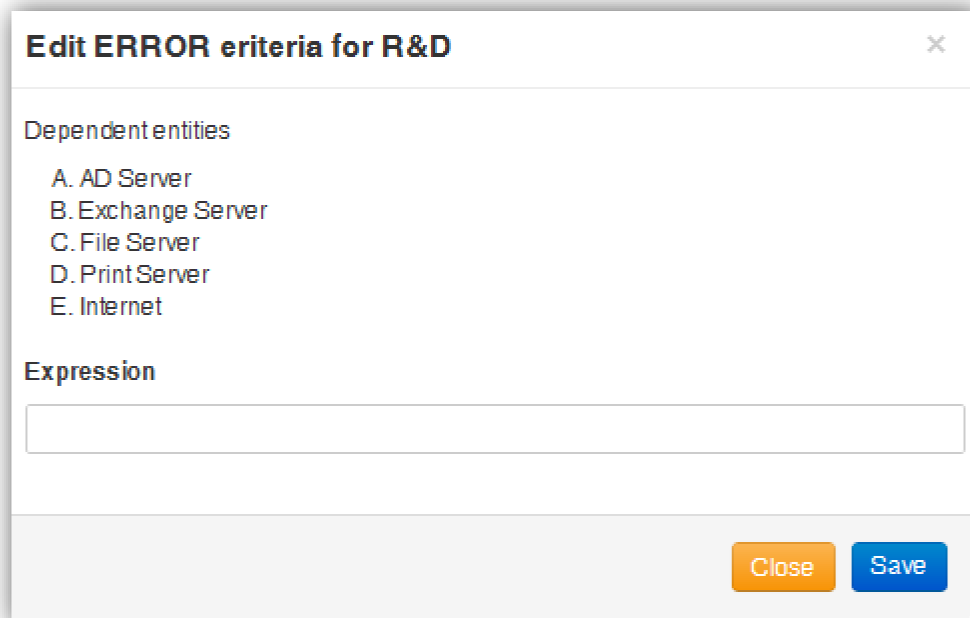
Kriterier bruges til at definere *Error* og *Warning* tilstandene. Disse tilstande benyttes i vid udstrækning igennem hele løsningen, og det er derfor vigtigt at de bliver opsat korrekt. Da kriterier kun kan oprettes på komponenter, skal brugeren for at oprette et kriterium først tilgå den ønskede komponent. I højre side af skærmbillede befinder sig en sektion ved navn *Depends on*, vist med

Billede 24 nedenfor.



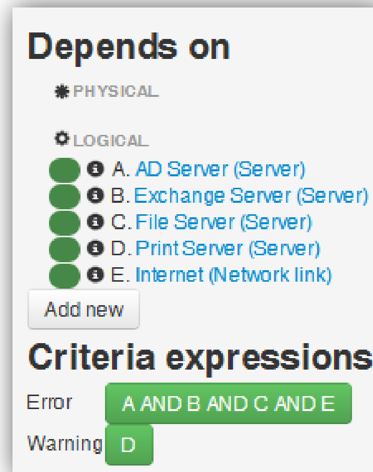

Billede 24 – Afhænger af og kriterier.

Her står listet alle de komponenter som af afhængig af den valgte komponent, og umiddelbart under er det muligt at se og sætte kriterier for den valgte komponent. På billedet fremgår kriterierne ikke umiddelbart, og det skyldes at der ikke manuelt at sat nogle kriterier. I det tilfælde af at der ikke er sat noget kriterium, overtager standard opsætningen. Det betyder for *Error* tilstanden at alle nævnte afhængigheder, om det er logiske eller fysiske, bliver AND'et sammen i et boolsk udtryk. For *Warning* tilstanden er standard værdien NULL. Der gælder altså ikke noget udtryk for *Warning*, og det kan altså ikke forekomme med mindre det er sat manuelt. For at indsætte et kriterium forskelligt fra standard kriterier, trykkes der blot på det røde eller grønne felt, ud for *Error* og *Warning*. Dette leder brugeren videre til en side, hvor kriteriet kan opbygges, vist med Billede 25 nedenfor.




Billede 25 – Opbygning af *Error* kriterium

Lad os tage udgangspunkt i afdelingen R&D. På nuværende tidspunkt er de afhængige af fem komponenter som det fremgår ovenfor. AD, Exchange, Filserver og Internet er alle komponenter som de er afhængig af for at kunne udfører deres arbejde overhovedet. Printserveren derimod er ikke drift kritisk, men det er dog en ulempe hvis den ikke fungerer. *Warning* kriteriet kan derfor sættes til A AND B AND C AND E. Hvor *Warning* så kan udtrykkes NOT D. Hvis eksemplets sættes ind, vil kriterierne se ud som vist nedenfor på Billede 26.




Billede 26 - Afhænger af og definerede kriterier.

## Funktionalitet

I dette kapitel vi vil gennemgå brugen af hovedfunktionaliteten i Infrastructure Diagnosis, hvilket er de tre funktioner konsekvensanalyse, systemdokumentation og statusrapportering.

### Brug af konsekvensanalyse

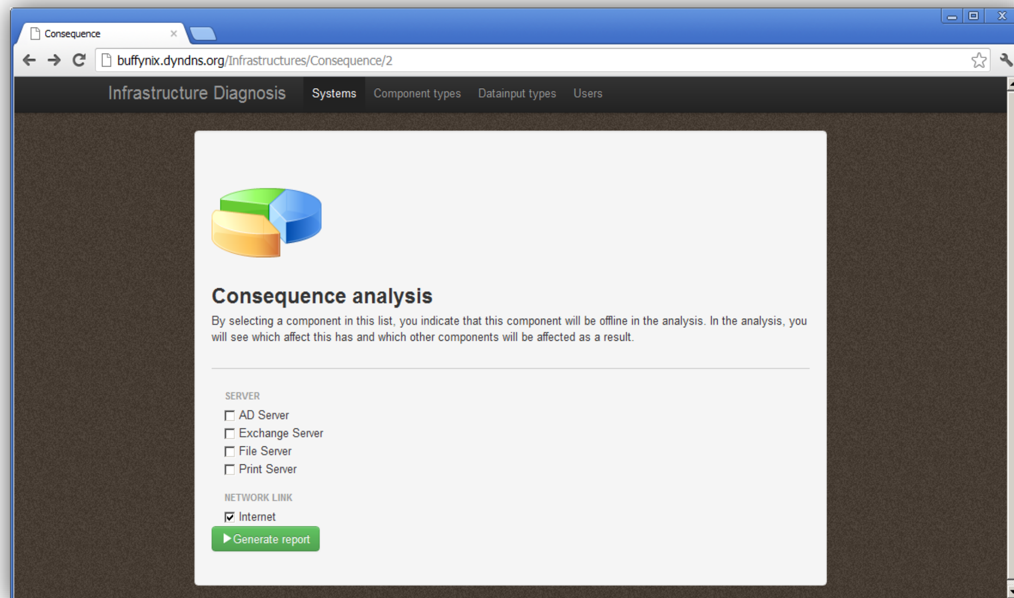
*Det er en forudsætning, at der er oprettet mindst et system med mindst en infrastruktur og mindst to komponenter med afhængigheder imellem sig, før man kan benytte sig af konsekvensanalysen.*

Som navnet på funktionaliteten afslører, så kan vi ved hjælp af konsekvensanalyse analysere hvilken konsekvens et givent scenarie vil have for et it-system. Funktionalitetens formål er at afklarer alle konsekvenser, for at gøre brugeren i stand til at træffe de korrekte valg i forbindelse med ændringer på et it-system.

Konsekvensanalysen kan kun udregnes per infrastruktur. Men funktionaliteten startes fra et link via system oversigten, hvor man umiddelbart efter bliver bedt om at vælge mellem systemets infrastrukturer. Fremgangsmåden er illustreret nedenfor:

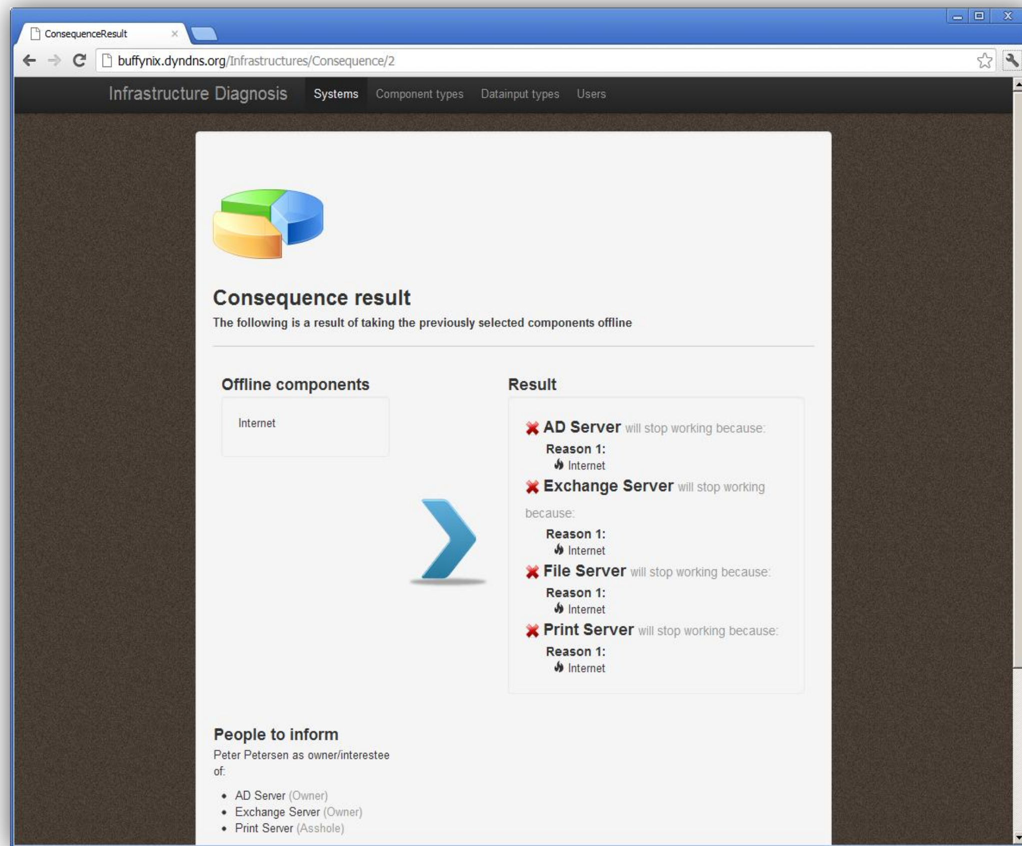
| Forside -> System -> Konsekvensanalyse -> Infrastruktur |
|---|

Konsekvensanalysen kan nu konfigureres ved at vi klikker på de *komponenter* der i analysen skal simuleres som havende tilstanden DOWN. På Billede 27 nedenfor har vi sat hak i *komponenten* Internet.




Billede 27 – Konfiguration af konsekvensanalyse

På Billede 28 nedenfor er vist resultatet af konsekvens analysen. Komponenten Internet har fået tilstanden DOWN, og det har bevirket at fire andre servere også har fået tilstanden DOWN, da de alle afhænger af *komponenten* Internet. I bunden at billedet bliver alle de personer som har tilknytning til de involverede komponenter listet, sådan at de kan informeres i tilfælde af at sceneriet realiseres.




Billede 28 – Resultat af konsekvensanalyse

## Brug af systemdokumentation

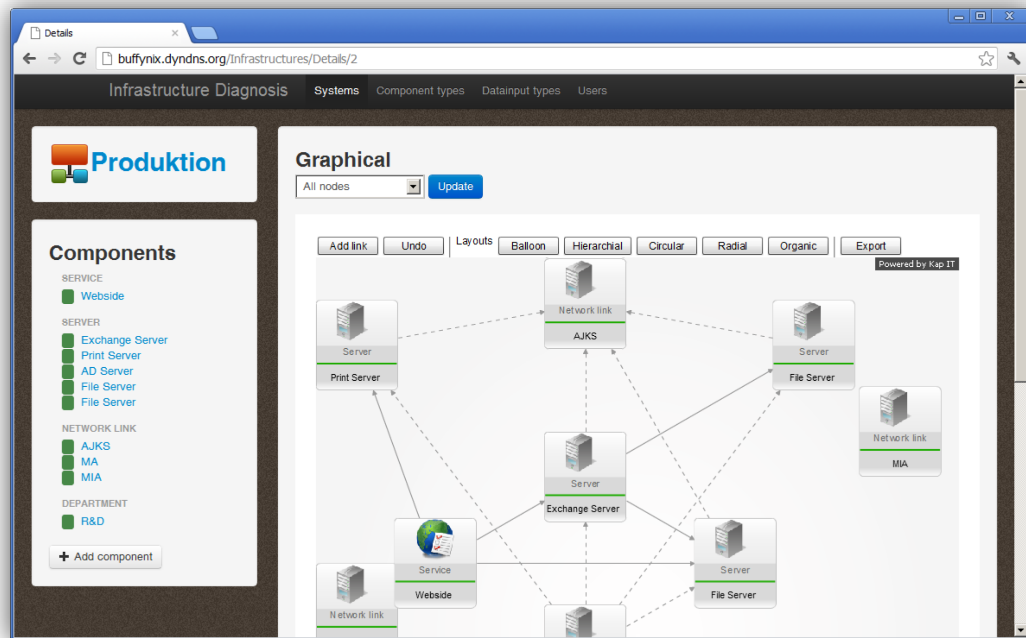
*Det er en forudsætning, at der er oprettet mindst et system med mindst en infrastruktur og mindst to komponenter med afhængigheder imellem sig, før man kan benytte sig af systemdokumentationen, da grafen ellers ikke renderes.*

Systemdokumentation er tiltænkt til at gøre dokumentation og overblik lettere for brugeren, ved hjælp af komponent informationer og en grafisk repræsentation af infrastrukturen. Systemdokumentationen fungerer på infrastrukturniveau, men funktionaliteten startes fra et link via system oversigten, hvor man umiddelbart efter bliver bedt om at vælge mellem systems infrastrukturer. Fremgangsmåden er illustreret nedenfor:

Forside -> System -> Systemdokumentation -> Infrastruktur

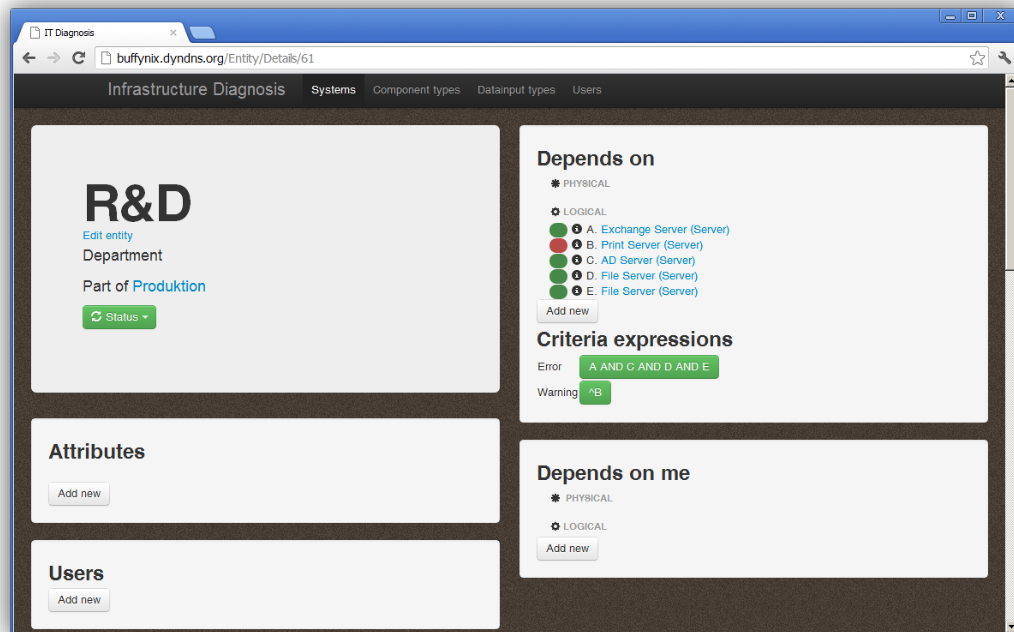
Vi bliver nu mødt af systemdokumentationens oversigt, som illustreret med Billede 29 nedenfor. I venstre side er infrastrukturens komponenter listet og i højre side er infrastrukturen afbildet. Komponent informationer tilgås ved at klikke på komponenter, hvorfra afhængigheder og andre egenskaber også kan aflæses og sættes.

I grafen er det muligt at se hvordan infrastrukturens komponenters indbyrdes forhold er, og via dropdown boksen kan man vælge at fokusere på specifikke typer *komponenter* eller komponenter med tilstand *Error* eller *Warning*.




Billede 29 – Systemdokumentations oversigt

Ved at klikke på en komponent bliver man ført videre til komponentens oversigtsbillede. Her er det vil brugeren kunne se detaljerede informationer omkring komponenten. Disse er illustreret med Billede 30 nedenfor.




Billede 30 – Komponent oversigt

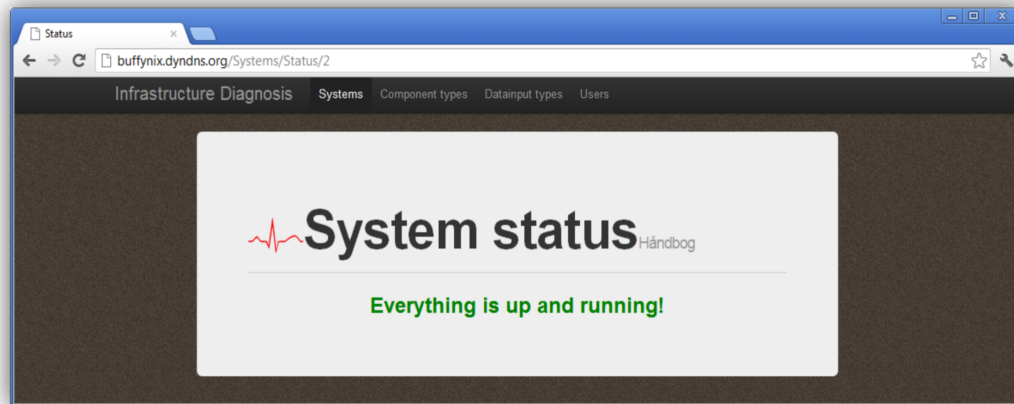
## Brug af statusrapportering

Statusrapportering er en funktionalitet som skal kunne give en bruger et hurtigt overblik over funktionalisten af et it system, fra et forretningsmæssigt synspunkt.

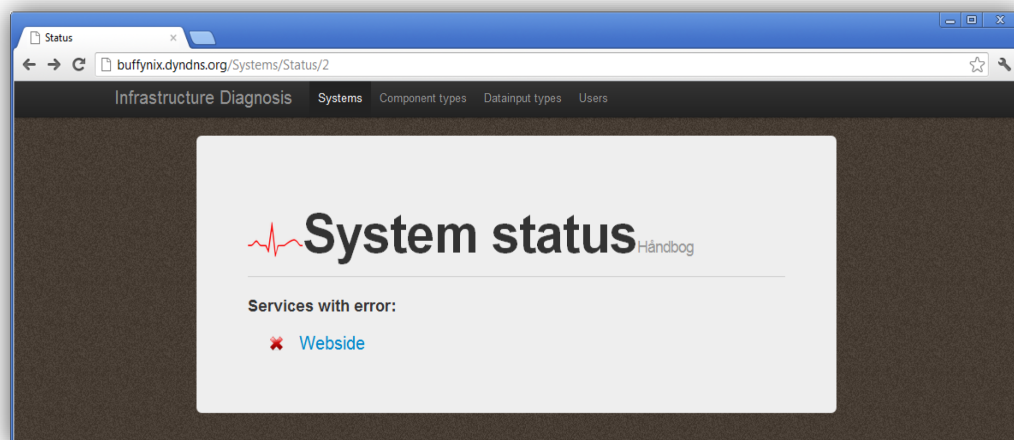
Statusrapportering kan give brugeren en hurtig status melding vedrørende systemets tilstand. Funktionaliteten startes fra et link i system oversigten.. Fremgangsmåden er illustreret nedenfor:

| Forside -> System -> Statusrapportering |
|---|

Det er servicesene der beskriver funktionaliteten der stilles til rådighed af et it-system, og det er derfor services der benyttes til statusrapporteringen. Af oversigten fremgår de services som har tilstanden *Error* eller *Warning*, og hvis de ikke eksisterer, så viser løsningen i stedet det. Dette er illustreret med Billede 31 og Billede 32 nedenfor.



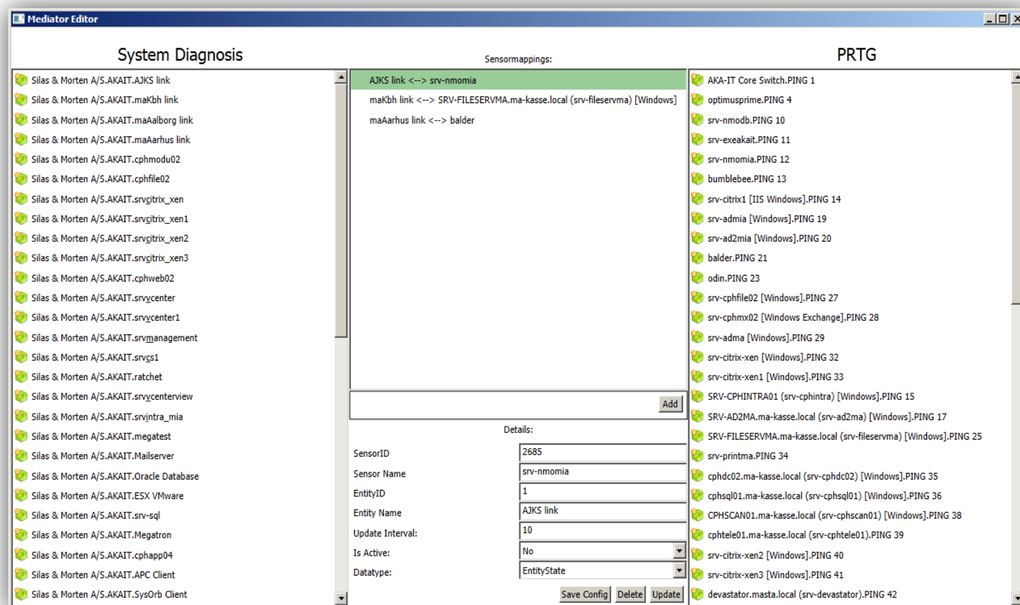

Billede 31 – Statusrapportering, normal drift




Billede 32 – Statusrapportering, fejl på service

### Mediator

Mediatoren benyttes til at opsætte associationer mellem PRTG og System Diagnosis. Det fungerer ved at man finder f.eks. en webserver komponent i System Diagnosis listen ude til højre og i venstre side finder man den samme komponent, bare repræsenteret i PRTG. Herefter trykker man på knappen 'ADD', hvilket så opretter associationen og lister den i det midterste vindue. Her kan man markere den for at se associationens detaljer, og hvis brugeren ønsker at ændre på dem kan det gøres i bunden. Der kan konfigureres opdaterings interval, om associationen er aktiv eller ej og hvilken datatype som associationen skal sende videre til System Diagnosis. På Billede 33 nedenfor er Mediatorens GUI illustreret.




**Billede 33 – Mediator oversigtsbillede.**