



Goodtech Monitoring

Et overvåking og monitoreringsprosjekt for Goodtech



Gruppe 29
Christoffer-Theodor Arntzen
Victor Nascimento Bakke
Tommy Abelsen
OsloMet - storbyuniversitet



Studieprogram: Dataingeniør
Postadresse: Postboks 4 St. Olavs plass, 0130 Oslo
Besøksadresse: Pilestredet 35, Oslo

Prosjektnummer
19-29
Tilgjengelighet
Offentlig

HOVEDPROSJEKTETS TITTEL	DATO
Goodtech Monitoring	23.05.2019
	ANTALL SIDER / BILAG
	73 / 2
PROSJEKTDELTAKERE	INTERN VEILEDER
Victor Nascimento Bakke Tommy Abelsen Christoffer-Theodor Arntzen	Boning Feng

OPPDRAGSGIVER	KONTAKTPERSON
Goodtech AS	Mikkel Sannes Nylend

SAMMENDRAG
Dette bachelorprosjektet er skrevet ved OsloMet - storbyuniversitetet i samarbeid med Goodtech AS. Goodtech Monitoring er et overvåknings- og visualiseringsverktøy for Goodtech sine MES-systemer. Goodtech Monitoring inkluderes i MES-en til Goodtech som en integrert modul, og hjelper dem oppdage og reagere på feil i systemene deres ute hos kundene deres.

3 STIKKORD
REST
SQL
Visualisering

Forord

Dette dokumentet er sluttrapporten til bachelorprosjektet “Goodtech Monitoring” skrevet av studenter ved OsloMet - storbyuniversitetet.

Dokumentet er optimalisert for digitale medier, med lenker til forskjellige deler av dokumentet, samt kildehenvisning. Rapporten kan likevel leses helhetlig i papirformat.

Oppgaven ble utlyst gjennom OsloMet sin “prosjektforslag”-side på hjemmesidene for bacheloroppgaver til informasjonsteknologi.

Høsten 2018 tok vi kontakt med Goodtech og ble kalt inn til intervju hvor de i dypere detalj forklarte ønsker og formålet ved dette prosjektet.

Vi ønsker å takke vår interne veileder ved OsloMet, Boning Feng for veiledning og gode råd underveis i prosjektet, og Torunn Gjester som ga oss hjelp i begynnelsen av prosjektet.

Videre ønsker vi å takke Goodtech som har gitt oss muligheten til å utvikle denne løsningen og har bistått med kontorplasser hele prosjektperioden.

Sist, men ikke minst, vil vi takke våre interne veiledere hos Goodtech, Mikkel Sannes Nylend, for god oppfølging, veiledning og faglig utvikling gjennom hele prosjektet, samt Øystein Myhre for den siste uka med veiledning.

1 Innledning	7
1.1 Gruppen	7
1.2 Oppdragsgiver	7
1.3 Goodtech Monitoring	8
1.4 Bakgrunn til oppgaven	8
2 Kravspesifikasjoner	9
2.1 Data som skal hentes ut	9
2.2 Loggbehandlere	10
2.3 Database	10
2.4 Visualisering	10
2.5 Utviklingsprosessen	11
2.6 Endringer i kravspesifikasjon	11
4 Teknologi	12
4.1 Versjonskontroll	12
4.2 Backend	12
4.2.1 Rammeverk og biblioteker	13
4.3 Database	14
4.3.1 MSSQL	14
4.3.2 Datatyper	15
4.3.3 Normalisering	15
4.3.4 Joins og delspørringer	16
4.3.5 Tilgangskontroll og sikkerhet	17
4.3.6 Ytelse	17
4.4 Visualisering	18
4.4.1 Hvordan lage en modul	18
4.4.2 Hva er en modul?	20
4.4.3 Tidsfilteret	20
4.4.4 Dashbordet	21
4.4.5 Datasource	21
4.4.6 Lokale dashbord variabler	22
4.4.7 Grafanas funksjoner	22
4.5 Verktøy	24
4.5.1 IntelliJ IDEA	24
4.5.2 Trello	25
4.5.3 Microsoft SQL Server Management Studio 2017	26
4.5.4 TortoiseSVN	27
4.5.5 Cron	27

5 Endring av database	28
5.1 Første forsøk med MSSQL	28
5.2 InfluxDB-forsøket	28
5.3 Andre forsøk med MSSQL	28
6 Testdata	29
7 Prosess	30
7.1 Oppgavefordeling	30
7.2 Backend	30
7.2.1 Uthenting av data	30
7.2.2 Normalisering	31
7.2.3 Første omstrukturering	32
7.2.4 Andre omstrukturering	32
7.2.5 Sikkerhet	34
7.3 Database	35
7.4 Visualisering	36
7.4.1 Målene med visualiseringen	36
7.4.2 Startfasen	36
7.4.3 Etter skiftet	37
7.4.4 Tidssoner	37
8 Dataflyt	38
9 Produkt	41
9.1 Server-instans	41
9.2 Backend	41
9.2.1 Mes-monitoring-client	41
9.2.1.1 CPULogger	42
9.2.1.2 DiskLogger	43
9.2.1.3 MemoryLogger	43
9.2.1.4 HeartbeatLogger	43
9.2.1.5 UsersLoggedInLogger	43
9.2.1.6 QueueElementLogger	43
9.2.1.7 StatusLogEntryLogger	43
9.2.1.8 Properties	44
9.2.2 Mes-monitoring-server	44
9.2.2.1 REST API	45
9.2.2.2 Consumer	45
9.2.2.3 DAO	46
9.3 Database	47

9.3.1 Oppsett	47
9.3.2 Tabellforklaring	49
9.3.3 Retention Policy	50
9.3.4 Normalisering	51
9.4 Visualisering	51
9.4.1 Infoskjerm / Hoved-Dashboard	52
9.4.1.1 CPU, disk og minne	52
9.4.1.2 Heartbeat-modul	53
9.4.1.3 QueueElement og StatusLogEntry	53
9.4.1.3.1 Modulær spørring	55
9.4.1.4 QueueElement-spesifikke tabeller	56
9.4.1.4.1 Processed elements in QueueElement	56
9.4.1.4.2 Ongoing elements in QueueElement	56
9.4.1.5 Users logged in	57
9.4.2 Unik kunde dashbord	57
9.4.2.1 Errors per statusindicator	58
9.4.2.2 Error Table - StatusLogEntry / QueueElement	59
9.4.2.3 Errors or warnings in a row	59
10 Sikkerhet og risiko	61
10.1 Backend	61
10.2 Database	64
10.2.1 Intranett	64
10.2.2 Brukere og tilgangskontroll	64
10.2.3 Backup	64
10.3 Visualisering	64
11 Videre utvikling	65
11.1 Backend	65
11.2 Database	66
11.3 Visualisering	67
11.3.1 Varsling	67
12 Alternativer	69
12.1 Backend	69
12.2 Database	69
12.2.1 InfluxDB	69
12.2.2 Elasticsearch	70
12.2.3 PostgreSQL og MySQL	71
12.2.4 Prometheus	71
12.3 Visualisering	72

13 Oppsummering	73
14 Referanser	74
15 Vedlegg	75
15.1 Vedlegg 1 - Ordliste	75

1 Innledning

1.1 Gruppen

Bachelorgruppen i dette prosjektet består av dataingeniørstudentene Tommy Abelsen, Christoffer-Theodor Arntzen og Victor Nascimento Bakke.

Tommy har gjennom studiene jobbet som studentveileder for Orakeltjenesten ved OsloMet, vært studentassistent i fagene “Programmering” og “Programutvikling”, og har holdt introduksjonskurs til førsteårsstudenter ved Institutt for informasjonsteknologi. Han har tidligere erfaring med utvikling av loggsystemer og datavisualisering fra hans sommerjobb hos Hafslund Nett i 2018.

Christoffer-Theodor Arntzen gikk på medier- og kommunikasjon VGS. Han tok opp fysikk og matte i form av forkurs over et år for å starte på et ingeniørstudie der førstevalget var byggingeniør. Etter et år på denne studieretningen skiftet han til dataingeniør ved OsloMet (tidligere HiOA).

Victor Nascimento Bakke har fagbrev som dataelektroniker, og begynte studiene sine på HiOA på lignende måte som Christoffer ved å ta forkurs først. Han har jobbet for Coop Norge Handels IT-avdeling i 4 år med fokus på overvåking av IT-infrastruktur i Coop. Ved siden av hovedoppgavene hans hos Coop har han også programmert flere forskjellige verktøy brukt internt i Coop IT.

1.2 Oppdragsgiver

Oppdragsgiver for bachelorprosjektet “Goodtech Monitoring” er bedriften Goodtech AS¹, avdeling industriell IT.



Goodtech er et teknologikonsern med kjernevirksomhet innen automasjon, kraft og industriell prosjektering. Deres visjon er å være førstevalget for industriell effektivitet.

Goodtech leverer prosjekter, service og produkter til kunder innen prosess- og produksjonsindustrien, havbruk, olje og gass, kraft og vannrensing.

¹ Logo hentet fra https://www.goodtech.no/media/goodtech_svalt.jpg

1.3 Goodtech Monitoring

I dette prosjektet skal vi hente ut og normalisere loggdata fra Goodtechs Manufacturing Execution System (MES)-instanser som sitter ute på forskjellige kunders lokaler. Disse føres så inn i en sentral database som er plassert på intranettet til Goodtech. Dataene brukes deretter av visualiseringsløsningen, som presenterer dem på en enkel og forståelig måte. Dette gjør det mulig å analysere loggdataene hos kunden.

Originalt hadde vi fått inntrykk av at dataene fra hver kunde ville være veldig forskjellige, men dette viste seg å ikke være tilfellet. Vi har som følger utviklet én universell løsning som fungerer for de nyeste versjonene av MES-instansene (kundeinstans) kundene har. Det ble også luftet ønske om at databasen skulle ligge i skyen, men i praksis spiller det ingen rolle hvor databasen er satt opp, og vi har så langt forholdt oss til en database på de lokale testserverene til Goodtech.

Løsningen som ble utviklet er todelt, med én integrert modul i MES, og én server med REST API, database og webapplikasjon for visualiseringen.

1.4 Bakgrunn til oppgaven

Per i dag har Goodtech ingen oversikt over loggene fra kundeinstanser i sitt bedriftslokale og belager seg på at kunden tar kontakt ved en feil.

Dagens situasjon kan dermed føre til at feil i systemet ikke oppdages. En feil kan forekomme i en kundeinstans, og gå ubemerket hen i lang tid før symptomene på feilen oppdages, og de oppdages først av kunden selv. Når feilen oppdages så sent kan det bety store økonomiske konsekvenser for kunden i form av for eksempel store tilbakekallinger av leveranser, skade på fabrikkutstyr, mangelfulle bestillinger eller lignende.

Når en kunde først tar kontakt med Goodtech om en potensiell feil, hender det at kunden unnlater å nevne egen feilsøking. Dette kan føre til at Goodtech bruker ressurser på feilsøking som allerede er gjort, eller at kunden har kommet i skade for å lage nye feil under egen feilsøking.

Hensikten med Goodtech Monitoring er å få tilgang til og visualisere tilstanden til kundeinstansene. Visualiseringen av loggdata gjør det enklere å følge med på når en feil oppstår, eller forvarslar på hva som kan føre til feil i systemene. Med Goodtech Monitoring kan Goodtech enkelt og raskt sette i gang feilsøking når det oppdages feil eller potensielle feil, selv uten at kunden opplever dette eller varsler selv. Goodtech Monitoring vil også enklere identifisere hvor feilen ligger, slik at man ikke må grave seg gjennom unødvendig mye loggdata for å finne kilden til problemet. Produktet var tenkt å være passende for å ha på en infoskjerm i

Goodtechs lokaler, designet slik at korte blikk opp på denne infoskjermen gir en oversikt av tilstanden til alle kundeinstansene hos Goodtech sine kunder.

2 Kravspesifikasjoner

I planleggingsfasen av dette prosjektet har vi hatt flere møter med Goodtech angående eventuelle krav de stiller til prosjektets utvikling og resultat. Goodtech har stilt noen overordnede krav til funksjonalitet og utforming av oppgaven, som vi har tydeliggjort med dem og formulert på følgende måte:

- Skal potensielt gjøre noen beslutninger angående de uthentede dataene.
- Skal sende inn dataene til en sentralisert database.
- En webside skal vise en intuitiv og presentabel visualisering av dataene.
- Det skal være mulig å gi websiden et kjapt blikk, og da enkelt se om det er noe som må tas tak i eller ikke.
- Det skal være mulig å se en oversikt over en gitt kundes MES-instans(er).

Vi har som følger delt kravspesifikasjonen inn i de tre hoveddelene produktet vil bestå av, i tillegg til utviklingsprosessen i seg selv:

- Loggbehandlere ute på kundenes servere som sender loggdata til databasen
- Databasen som holder på all loggdata for alle kundene
- Visualiseringsplattform som leser og tolker data fra databasen

2.1 Data som skal hentes ut

- Data fra MES-ens "StatusLogEntry"-tabell.
 - Denne tabellen inneholder statuslogger, med tilhørende indikatorer som grupperer sammen forskjellige feilmeldinger i kategorier, og meldinger som beskriver feil som har oppstått i systemet.
- Data fra MES-ens "QueueElement"-tabell.
 - Kø-tabellen systemet bruker for videreformidling av oppgaver til andre tilkoblede systemer i MES-en. Denne tabellen fungerer som en kommunikasjonskanal mellom MES-en og andre systemer, hvor MES-en legger til et element på det tilhørende systemets kø i QueueElement-tabellen. Hver kø i QueueElement-tabellen er definert ved kolonnen "queueName", hvor en gitt unik "queueName"-verdi markerer at den gitte raden gjelder for den spesifikke køen og dens systemer. Etter at kø-elementet har blitt lagt til i køen vil det tilhørende systemet utføre oppgaven som kreves av kø-elementet, for så å oppdatere det med resultatet.

- Processor (CPU), disk og minnebruk.
 - Det er ønskelig å se CPU, disk og minnebruk for ytterligere visualisering av maskinens tilstand til enhver tid. En gitt feil ett sted i systemet kan være relatert til et økt bruk av CPU, disk eller minne, og monitorering av disse tre ressursene vil dermed være essensielt for hver av kundeinstansene.
- Heartbeat
 - Selve monitoreringssystemet på hver kundeinstans må også overvåkes, da en mangel på loggdata fra en kundeinstans kan komme av flere årsaker. Hvis CPU-, disk- og minneloggføring er slått av vil ikke monitoreringsløsningen føre inn noe data før det dukker opp nye elementer i StatusLogEntry- eller QueueElement-tabellene. Heartbeat lar oss se at monitoreringssystemet fortsatt kjører som det skal ved at den sier ifra periodisk at den fortsatt er på.

2.2 Loggbehandlere

- Leser data fra databasen til MES-systemet til kunden.
- Vi utvikler våre løsninger mot siste versjon av systemet til Goodtech, så hver kunde har generelt sett samme struktur på loggdataene.
- Leser data fra StatusLogEntry-tabellen.
- Leser data fra QueueElement-tabellen.
- Skal kun være tilgjengelig på Goodtechs nettverk og for hver individuelle kunde.

2.3 Database

- Ingen reelle krav til database vi benytter.
- Skal kun leses fra Goodtechs nettverk, men skal kunne skrives til fra kundenes nettverk.

2.4 Visualisering

- Web-basert grafisk fremvisning av datarelasjoner og tilstanden til kundeinstansene.
- Ingen reelle krav til spesifikt hva slags visualiseringsplattform vi benytter, om det er en ferdig løsning eller en egenutviklet en.
- Benytter regler for tolking av data fra StatusLogEntry-tabellen.
- Benytter regler for tolking av data fra QueueElement-tabellen.
- Skal kun være tilgjengelig på Goodtechs nettverk.

2.5 Utviklingsprosessen

Goodtech har ikke stilt noen krav til utviklingsprosessen, så vi satte noen enkle krav til oss selv for hvordan vi ønsket at utviklingsprosessen skulle foreta seg. Vi ønsket originalt å benytte Scrum-metodikken, men det viste seg veldig tidlig at vår manglende erfaring med slik metodikk, samt Goodtechs til tider varierende tilgjengelighet for møter grunnet deres eget arbeid, gjorde dette vanskelig. Vi endte dermed naturlig opp på en Kanban-lignende metodikk, hvor vi benyttet Trello, en kanban-board-tjeneste, for å organisere arbeid i små doser, som så kan jobbes på individuelt.

2.6 Endringer i kravspesifikasjon

Gruppen ble ferdig med en prototype allerede om lag en tredel ut i semesteret. Den originale kravspesifikasjonen var minimal, og vi regnet med at det i ettertid kom flere forslag til funksjonelle og ikke-funksjonelle krav etter hvert som prosjektet ble mer og mer ferdig. Dette er et velkjent fenomen, og kalles gjerne "Scope Creep" i ekstreme tilfeller.

I backend handlet blant annet disse forslagene om tilleggsfunksjonaliteter, arkitektur og struktur; muligheter for å konfigurere loggere med forskjellige intervaller; muligheter for å aktivere/deaktivere hele loggesystemet eller individuelle loggere; loggføring av CPU, disk og minnebruk; samt sikkerhet og autentisering. Dette førte til at det underveis i prosjektforløpet ble flere endringer av kravspesifikasjonen.

Det ble ikke satt noen konkrete krav til hvordan visualiseringen skulle bli utført. Dette medførte at mye av utviklingen ble gjort i lag med veileder på ukentlige møter, hvor vi presenterte forskjellige data vi hadde utviklet visualisering for og fikk tilbakemelding på hvilken retning vi skulle gå videre. Prosessen fulgte en "prøving og feiling"-fase i starten av prosjektet der det ble utviklet noen moduler som ble vist til veileder der han ga tilbakemelding på hva som var bra eller hva som trengte endringer. Utviklingen foregikk slik fordi Goodtech ønsket å se hva vi kunne utvikle på egenhånd. I denne fasen kom veileder, andre ansatte hos Goodtech og gruppens medlemmer med flere idéer om hva som kunne være nyttig å visualisere. Denne perioden med prøving og feiling varte i ca. 1,5 måneds tid før vi fikk konkretisert ønskene til Goodtech, og kunne iterere videre på den typen datavisualisering de ønsket.

4 Teknologi

4.1 Versjonskontroll

Goodtech har i lang tid benyttet seg av Apache Subversion (SVN)² for distribuert versjonskontroll av kildekoden i deres prosjekter. Et versjonskontrollsystem holder rede på et sett med filer og mappers tilstand over tid, og hvordan de endres etterhvert som en legger til, fjerner eller endrer på innholdet i de nevnte filene eller mappene. SVN er da et versjonskontrollsystem som fokuserer på en enkelt kilde for historikken til et prosjekt, hvor en utvikler som interagerer med prosjektet gjør endringer direkte på denne ene sentraliserte kilden. Siden Goodtech benytter SVN til deres prosjekter valgte vi også SVN for dette prosjektet, og fordelte hele prosjektet inn i 3 forskjellige SVN-mapper, også kalt “repositories” eller “repos”: “loghandler”, “db” og “visualization”.



Vi har tidligere erfaring med Git, et konkurrerende versjonskontrollsystem til SVN med mer fokus på lokal lagring av versjonshistorikk for offline bruk, men valgte å bruke SVN grunnet Goodtechs bekjentskap til SVN i forhold til Git, samt at Goodtech allerede har en intern SVN-server.

En utfordring vi møtte ved å bruke Subversion var at vi ikke kunne lagre progresjon i prosjektet utenfor kontorene til Goodtech, siden vi ikke hadde mulighet til å nå deres interne SVN-servere fra utsiden, og SVN har ikke muligheter for lokal lagring av historikk. Dette gjorde det mindre effektivt å jobbe med prosjektet hjemmefra.

4.2 Backend

Programmeringsspråket vi skulle benytte for å lage programvare for å hente ut, prosessere og videresende loggdata var allerede bestemt veldig tidlig i planleggingsfasen av prosjektet. Siden Goodtech sin nåværende løsning er skrevet i Java³, ønsket de at vi skulle utvikle Goodtech Monitoring i Java. Dette blant annet for å kunne inkludere vår utvidelse i deres produkt som en integrert modul.



² <https://subversion.apache.org/>, logo hentet fra <https://svn.apache.org/repos/asf/subversion/svn-logos/images/tyrus-svn2.png>

³ <https://www.java.com/en/>, logo hentet fra https://en.wikipedia.org/wiki/File:Java_programming_language_logo.svg

Vi benyttet som følger Java versjon 8 for utviklingen av backend-modulene, med forskjellige rammeverk, for å oppfylle kravene til Goodtech.

4.2.1 Rammeverk og biblioteker

Vi har benyttet noen rammeverk i backend-delen av dette prosjektet. Et rammeverk brukes for å kjøre egen kode på en spesifikk måte, hvor egen kode tilpasser rammeverket til den oppgaven det skal utføre. Rammeverk brukes gjerne for å slippe å skrive kode og tester som andre allerede har gjort. I tillegg er store rammeverk som Spring-rammeverkene ansett som tryggere enn å implementere samme funksjonalitet på egenhånd, siden disse rammeverkene er skrevet av mange bidragsytere og er testet grundig før de blir utgitt.

- Spring⁴
 - Spring Framework⁵
 - Spring Boot
 - Spring Security
 - Spring Data JDBC



Vi benyttet rammeverk fra Spring i dette prosjektet for å enklere lage konfigurerbare applikasjoner med REST API og databasebehandling med autentisering og kryptert datatrafikk.

Et Representational State Transfer (REST) API er en type web-API som gjør det mulig å sende eller motta informasjon fra en webtjeneste ved å bruke forhåndsdefinerte regler for kommunikasjon.

Spring Framework er et rammeverk som tilbyr konfigurasjon av Spring-applikasjoner, og fungerer som hoved-infrastruktur i Spring-prosjekter.

Spring Boot tilbyr selvstendige Spring-applikasjoner med blant annet auto-konfigurasjon og inkludert webserver.

Spring Security er et sikkerhets-rammeverk som tilbyr konfigurierbar autentisering og tilgangskontroll, samt integrering mot andre Spring-rammeverk. Spring Security tilbyr også flere sikkerhetsmekanismer enn det vi har brukt i dette prosjektet.

Spring Data JDBC er et rammeverk tilhørende Spring Data-familien av rammeverk. Dette rammeverket støtter bruk av Java Database Connectivity (JDBC), og vi bruker dette i vår kommunikasjon mellom Spring-applikasjonene våre og databasen.

⁴ <https://spring.io>, logo hentet fra https://commons.wikimedia.org/wiki/File:Spring_Framework_Logo_2018.svg

⁵ <https://spring.io/projects/spring-framework>

- Microsoft SQL
 - Microsoft SQL JDBC Driver for SQL Server⁶

Vi bruker Microsoft SQL JDBC Driver for SQL Server for å opprette tilkoblinger til Microsoft SQL Server. Det er denne driveren som gjør det mulig å benytte Spring Data JDBC til å sende til, og motta dataene fra, databasen.

- Apache
 - Maven⁷
 - HttpClientComponents HttpClient⁸



Apache Maven er et rammeverk, og verktøy, som automatisk laster ned deklarererte avhengigheter, tillater automatisk bygging, testing, pakking og distribuering av Java-prosjekter. Vi bruker Maven for å slippe å manuelt laste ned bibliotekene vi bruker i prosjektene, og for å bygge/kompilere koden og generere kjørbare Java Archive (JAR)-filer.

HttpClientComponents er et rammeverk fra Apache som tilbyr forskjellig funksjonalitet og verktøy knyttet til blant annet HTTP-protokollen.

Vi bruker HttpClient for å konfigurere en HTTP-tilkobling fra klient-applikasjonen til server-applikasjonen med støtte for kryptert HTTPS-kommunikasjon.

4.3 Database

4.3.1 MSSQL

Vi benyttet Microsoft SQL Server 2017 (MSSQL)⁹ for langtidslagring av loggført data. MSSQL er et relasjonsdatabasesystem for lagring av relasjonsbaserte data med støtte for store deler av Structured Query Language-standarden (SQL)¹⁰. MSSQL lar oss definere hvordan hver enhet med data skal se ut i form av tabeller, hvor hver enhet er en rad som har et antall attributter lagret i kolonner i tabellen. En gitt rad kan være koblet sammen med en rad i en annen tabell ved hjelp av "fremmednøkler", hvor en gitt rad som peker til en annen rad har den andre radens

⁶

<https://docs.microsoft.com/en-us/sql/connect/jdbc/microsoft-jdbc-driver-for-sql-server?view=sql-server-2017>

⁷ <https://maven.apache.org/>, logo hentet fra

<https://maven.apache.org/images/maven-logo-black-on-white.png>

⁸ <https://hc.apache.org/httpcomponents-client-ga/>

⁹ <https://www.microsoft.com/en-us/sql-server/sql-server-downloads>

¹⁰ http://standards.iso.org/ittf/PubliclyAvailableStandards/c053681_ISO_IEC_9075-1_2011.zip

“primærnøkkel” i en av kolonnene dens. En “primærnøkkel” er en unikt identifiserende kolonne for en gitt rad.

4.3.2 Datatyper

Kolonnene i hver tabell har en bestemt type, som begrenser hva slags type data som kan lagres i den gitte kolonnen. En kolonne kan også markeres som “not null”, som vil si at den gitte kolonnen ikke kan være tom for noen rader. Vi benytter oss av dette veldig mye, da det er enklere å ha med data å gjøre når vi kan garantere at data som burde være der faktisk er der.

4.3.3 Normalisering

Data lagret i en relasjonsdatabase som MSSQL burde “normaliseres” når det lagres. Det vil si at hver tabell representerer kun 1 unik entitet fra de gitte dataene, og flere tabeller kobles sammen ved hjelp av fremmednøkler for å indikere relasjoner mellom dataene. Et eksempel på dette er om en har en “Personer”-tabell, hvor man lagrer navn, adresse, postnummer og poststed. Postnummer og poststed er tett bundet til hverandre, og representerer i bunn og grunn kun et postnummer, da et gitt postnummer impliserer et gitt poststed. Som følger kan postnummer og poststed trekkes ut til en egen tabell, hvor postnummeret kan være en unik primærnøkkel. “Personer”-tabellen vil så ha en fremmednøkkel som peker til en gitt rad i “Postnummer”-tabellen.

Normalisering er kategorisert i forskjellige grader, hvor høyere grader av normalisering betyr som oftest flere tabeller. Disse kategoriene går fra “første normalform” (1NF), helt opp til “sjette normalform” (6NF), i tillegg til noen spesielle normaliseringsformer imellom. Hver normalform krever at den foregående normalformen er oppnådd for en gitt tabell, i tillegg til noen nye krav for hver normalform.

- 1NF: Verdiene i hver kolonne av tabellen er atomære, og kun representerer én ting. Eksempler på dette er å dele opp “navn” i “fornavn” og “etternavn”, eller “adresse” i “gatenavn” og “gatenummer”. Under slik oppdeling kan det også gi mening å opprette ekstra tabeller for å representere unike entiteter i den originale databasen. (Studytonight, u.å.)
- 2NF: Hver rad har en “kandidatnøkkel” i form av en eller flere kolonner som kan brukes for å unikt identifisere den gitte raden. Resten av kolonnene i raden skal være knyttet til hele denne kandidatnøkkel. Om noen kolonne kun har en tilknytning til deler av kandidatnøkkel må tabellen deles opp ytterligere for å gjøre kandidatnøkkel unikt identifiserende for alle de andre kolonnene i raden. (Studytonight, u.å.)
- 3NF: Ingen kolonner har en tilknytning til en annen kolonne i tillegg til kandidatnøkkel. Dette kan skje for eksempel hvis man har et postnummer og et poststed i en “Personer”-tabell. Poststedet er tett tilknyttet postnummeret, og både poststedet og postnummeret er tilknyttet primærnøkkel i raden. Postnummeret og poststedet kan

som følger trekkes ut til deres egen "Poststed"-tabell, hvor postnummeret er en primærnøkkel for poststedet. (Studytonight, u.å.)

- BCNF (Boyce-Codd normalform): videre utvidelse av 3NF. hver kolonne beskriver eller er tilknyttet kun primærnøkkelen og ingenting annet. Dette er forskjellig fra 3NF når primærnøkkelen består av en sammensatt kandidatnøkkel med to eller flere kolonner. Hver kolonne som ikke er en nøkkel må ha en tilknytning til absolutt hele primærnøkkelen, kun primærnøkkelen og ingenting annet. (Studytonight, u.å.)
- 4NF: tar for seg flervedistilknytninger. Vi har en tabell "A", med kolonnene "X", "Y" og "Z", som henholdsvis, om vi tar for oss én enkelt rad av gangen, har verdiene "x", "y" og "z". Det er en flerverditilknytning fra X til Y hvis for en bestemt verdi x finnes det en eller flere verdier y som ikke bestemmes av z på noen måte, men hvor Z også er en del av kandidatnøkkelen {X, Y, Z}.
- 5NF: femte normalform tar for seg fjerning av redundant data, samt sikring av invarianter i dataene. La oss si vi har en tabell med tre kolonner; X, Y og Z; hvor hver kolonne er en del av den unike identifiserende sammensatte primærnøkkelen for tabellen. Denne tabellen er i 5NF hvis den allerede er i 4NF i tillegg til at den ikke kan deles opp i ytterligere mindre tabeller uten tap av informasjon. Hvis det eksisterer tilknytninger som kan beskrives som par av hver kolonne, slik som {{X, Y}, {X, Z}, {Y, Z}}, så kan tabellen deles opp videre, og den er ikke i 5NF før den er delt opp. (GeeksForGeeks, u.å.)
- 6NF: en tabell er i 6NF hvis den består av kun primærnøkkelen og opptil 1 annen kolonne, men intet mer. 6NF brukes blant annet i datasentre, og er optimalisert for de kontinuerlig varierende behovene til store systemer (Anchor Modeling, u.å.).

4.3.4 Joins og delspørringer

Siden dataene våre er delt opp i tabeller må vi benytte "joins" eller "delspørringer" om vi ønsker å hente data fra flere enn én tabell av gangen.

En "join" er en måte å hente ut data fra to eller flere tabeller samtidig hvor man kombinerer tabellene til én tabell ved å matche radene sammen gjennom en identifiserende kobling mellom rader i hver tabell, i vårt tilfelle som oftest i form av en fremmednøkkel som peker til en annen rads primærnøkkel. Dette kan være noe slikt som å kombinere en tabell med CPU-målinger med en tabell over kundeinstanser, hvor man bruker CPU-tabellens kolonne over den tilhørende kundeinstansen for en gitt CPU-måling ("customerInstance_pk" i vårt tilfelle) som det bindende leddet. På denne måten kan man hente ut navnet på den gitte kundeinstansen for hver CPU-måling, og videre tilpasse spørringen etter de dataene man vil ha fra hver av tabellene.

En "delspørring" er en spørring i en annen spørring. Siden hver spørring returnerer en tabell kan en for eksempel benytte en delspørring i stedet for en tabell i "FROM"-delen av en spørring. Delspørringer kan også returnere enkeltverdier om selve delspørringen ender opp med kun 1 rad og 1 kolonne. I disse tilfellene kan den returnerte tabellen, som bare har én enkelt verdi i

seg, brukes som om den var en enkelt verdi. Dette er hjelpsomt i "WHERE"-delen av en spørring når man sammenligner kolonneverdier i en rad med resultatet fra en delspørring.

4.3.5 Tilgangskontroll og sikkerhet

Begrensning av tilgang til dataene er også nødvendig, noe MSSQL lar oss gjøre ved hjelp av brukere med tilpassede rettigheter. På denne måten kan vi ha en bruker med de nødvendige rettighetene for innføring av data fra REST APIet, og en bruker for uthenting av data fra visualiseringen.

4.3.6 Ytelse

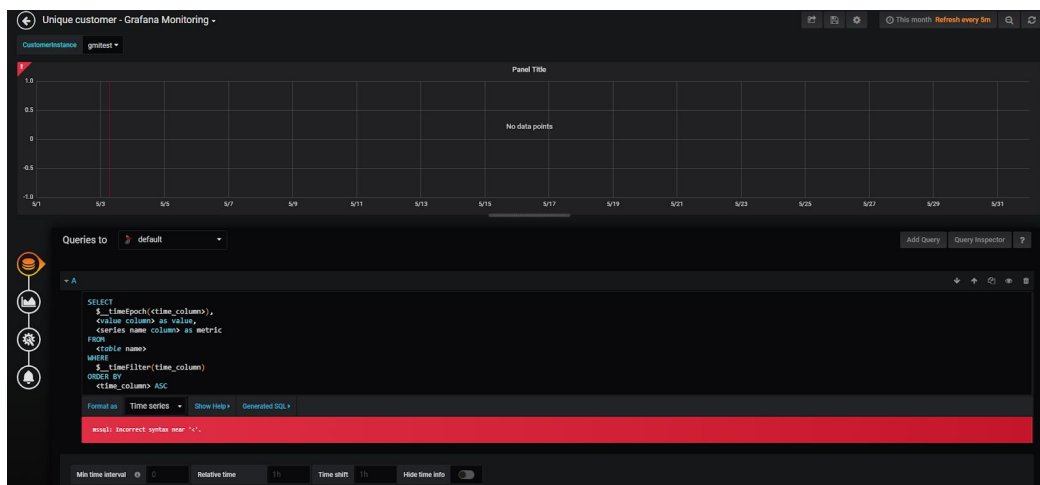
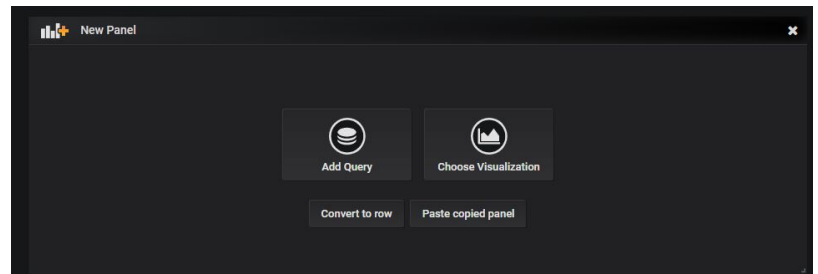
MSSQL, samt andre relasjonsdatabaser, kan benytte indekser for å øke ytelsen ved uthenting, oppdatering og (til en viss grad) sletting av data. En indeks optimaliserer sammenligning av data ved å lagre en kopi av dataene for en gitt kolonne i et format som er mye kjappere å søke gjennom. Indekser reduserer derimot ytelsen ved innføring av nye rader i kolonnen indeksen gjelder for sin tabell, da de nye radene må legges til i hver indeks, i tillegg til selve tabellen (Winand, u. å.). Ved sletting må en gitt rad slettes fra både tabellen og indeksene raden er i, noe som er tregere jo flere indekser en har, men for å finne frem de radene som skal slettes vil indekser gjøre ting kjappere. Som følger er det kjappere med 1 indeks for en tabell ved sletting enn ingen indeks (Winand, u.å.).

4.4 Visualisering

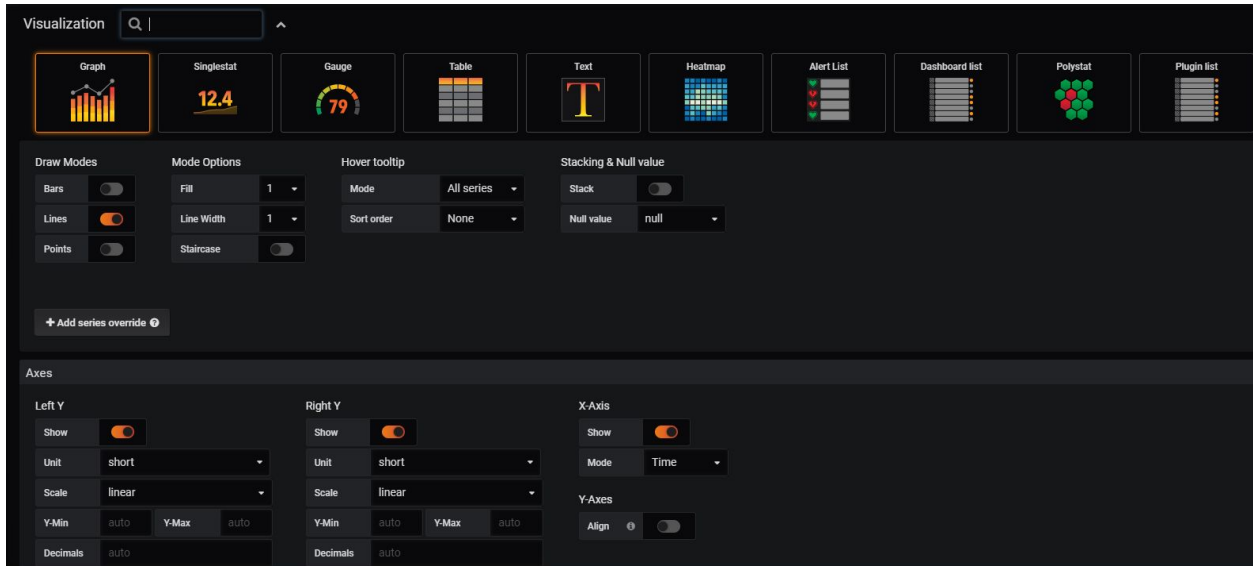
Grafana er et verktøy for å visualisere tidsseriedata. Det tilbyr flere funksjoner som gjør det enklere å representere data over tid, både tilbake i tid og nåværende endringer. Verktøyet har et tidsfilter, som brukes i spørringene og kan endre på grafene i sanntid. Grafana har også flere måter å representere dataene på. Noen av disse er; grafer, tabeller, varmekart. Det har også mulighet for egendesignede moduler.

4.4.1 Hvordan lage en modul

Som sett i bildet til høyre så har man mulighet til å velge om man vil starte med en spørring eller velge hvilken type graf en modul skal være.

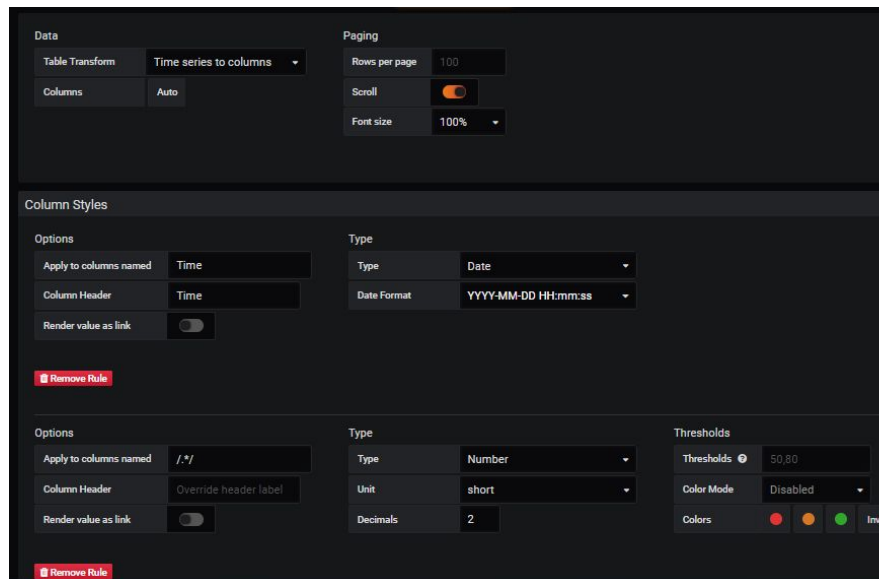


Om man velger ny spørring kommer man til siden vist over. Øverst ser vi hvordan grafen ser ut. I boksen under skriver man inn spørringen; den er fylt med eksempelkode til å begynne med. Det røde rektangelet er en indikator på at spørringen ikke er gyldig. Feilmeldingen i denne boksen forteller oss hva som kan være feil med spørringen. De fire sirkelene på venstre side er menyen for resten av modulen.



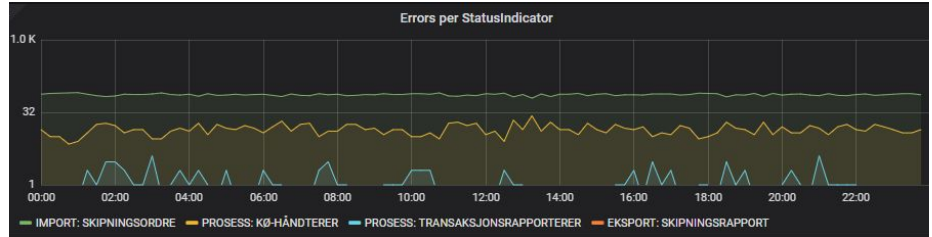
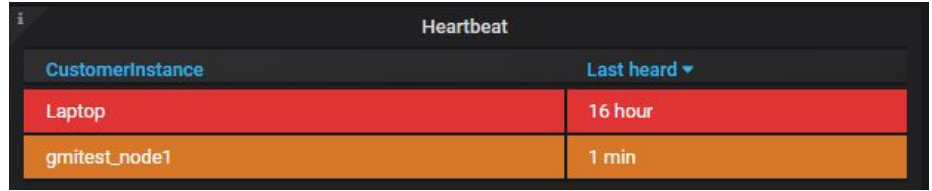
Dette er menyen for visualiseringen (2. sirkel fra toppen). Her kan man endre hvordan man vil visualisere dataene spørringen henter ut, hvordan grafen skal se ut, om det skal være linjediagram, stolpediagram osv. Alle de forskjellige grafene har sine egne former for å manipulere fremvisningen av dataen. Som man ser ovenfor spesifikt for graf, har man innstillinger for Y aksen. Hvilke data som denne aksen representerer og om den skal ha en form for skalar. Under "unit" har man en liste med mange alternativer som f.eks. valuta, tid eller matematiske enheter.

Her er innstillingene for tabellmodulen. I "data" boksen så kan man endre hvordan tabellen skal vise dataene. Om det skal være en enkel tabell, aggregert data eller som vist her tidsseriedata til kolonner. I disse "options"-boksene kan man navngi kolonner i en tabell. I dette eksempelet så sier vi at kolonnen "metric" skal ha navnet "CustomerInstance - queueName". Her har man også mulighet til å legge til en bestemt enhet som tabellen skal representere. I "threshold" kan man legge til fargekoder basert på hvor liten eller stor verdien er i spørringen. Her blir hele raden i tabellen rød om den overskrider 10, gul om den er mellom 10 og 5 og grønn fra 5 og til 0. Man kan legge til så mange av disse "option"-blokkene som det er kolonner i tabellen.



4.4.2 Hva er en modul?

En modul kommer i flere fasonger i Grafana. Det kan være alt fra grafer til histogram til plotter i en graf. Til høyre ser vi tre eksempler på dette. Den øverste er en enkel tabell, modulen i midten er en multi-seriell linjegrav og den tredje er en linjegrav med plotter.



Modulene må ha en spørring i seg så det faktisk er noe å vise i grafen. Det kan være kjekt med regex på modulen for å

fremheve data som er viktig/kritisk. I disse eksemplene så er det bare “Heartbeat”-modulen som har dette, der radene blir fargelagt i forskjellige farger etter hvor stor verdien er i høyre rad.

4.4.3 Tidsfilteret

Bildet til høyre viser tidsfilteret til Grafana. I denne menyen er det mange intervaller å velge mellom. Under “From”- og “To”-feltene kan man velge et egendefinert intervall. “Refreshing every” velger hvor ofte dashbordet skal oppdatere verdiene i modulene. Ved å velge et gitt intervall så vi alle modulene i Grafana endre seg og vise dataene fra databasen i dette intervallet.

Quick ranges

Last 2 days	Yesterday	Today	Last 5 minutes
Last 7 days	Day before yesterday	Today so far	Last 15 minutes
Last 30 days	This day last week	This week	Last 30 minutes
Last 90 days	Previous week	This week so far	Last 1 hour
Last 6 months	Previous month	This month	Last 3 hours
Last 1 year	Previous year	This month so far	Last 6 hours
Last 2 years		This year	Last 12 hours
Last 5 years		This year so far	Last 24 hours

Custom range

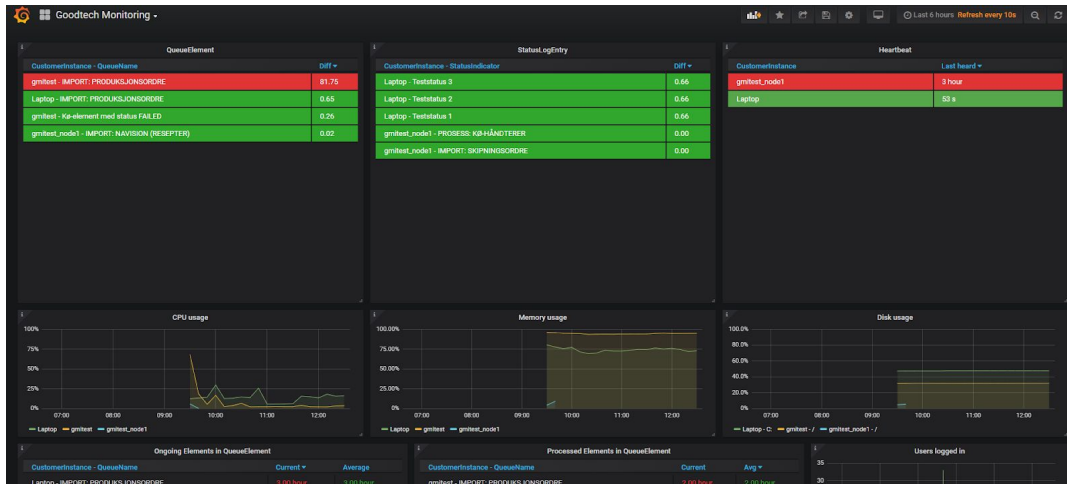
From: now-6h

To: now

Refreshing every: 10s

Apply

4.4.4 Dashbordet

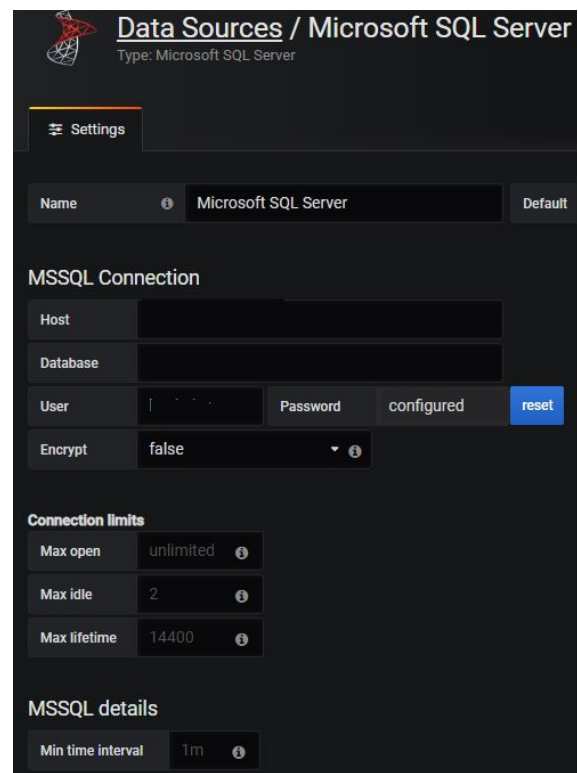


Her er det ene dashbordet vi har lagd (ikke alle modulene er med i bildet). Hver firkant er en modul og har en unik spørring i seg. Det betyr at det her blir 9 spørringer hvert tiende sekund som var definert ovenfor. I dette dashbordet ser man klart hvor og hva som har gått galt. Øverst i venstre hjørne ser vi “QueueElement” modulen. Her er det en rød linje som indikerer at noe har gått galt. Den viser da hvilken kundeinstans som feilet og hvor på denne kundeinstansen problemet ligger. I “Heartbeat”-modulen til høyre ser vi at det også er en rød linje. Her er det en kundeinstans som ikke har meldt noen oppdatering på 1 time. Videre ser vi på “Memory usage” modulen under at kundeinstansen “gmitest” ligger nærmest 99% minnebruk over lengre tid.

Ut fra disse observasjonene kan man gjøre diverse valg for å sjekke nærmere hva som er problemet.

4.4.5 Datasource

Her kan man definere hvilke databaser Grafana skal spørre mot. Den trenger en IP-adresse og en bruker. En kan også definere minimums tidsintervall Grafana kan spørre databasen. Hvis man setter dette til for eksempel 5 minutter, så kan ikke Grafana spørre oftere enn hvert femte minutt.



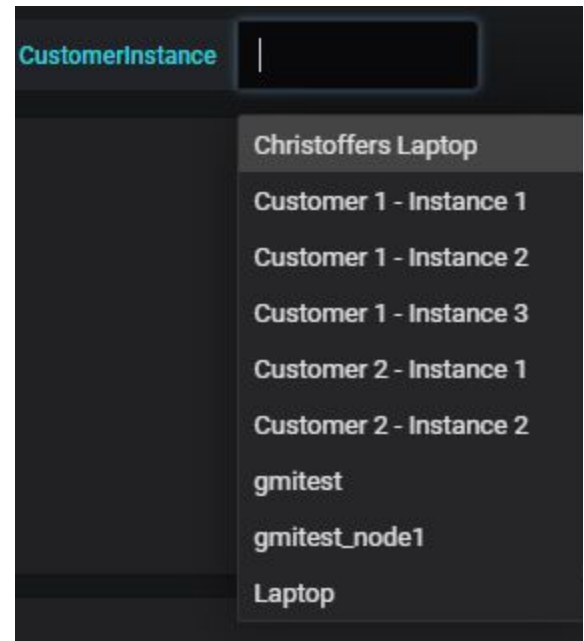
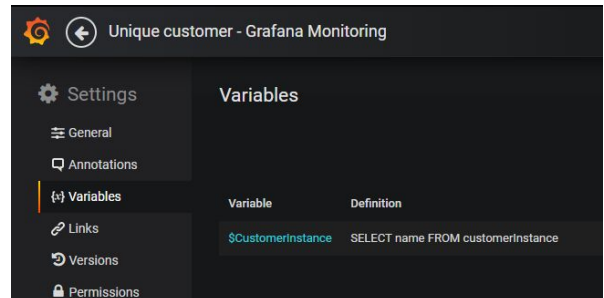
4.4.6 Lokale dashboard variabler

En unik funksjon i Grafana er at man kan ha gitte panelvariabler. Ved å ha en spørring her så vil man kunne velge disse verdiene i en rullegardin på panelet.

Tatt et eksempel fra vår oppgave, vi har en liste med kunder og vil lage et panel som endrer grafene etter hvilken kunde som blir sett på. Dette gjøres ved å sette panelvariabelen til å være en liste over alle kundene. Videre blir det inkludert i WHERE klausulen i hver modul at kunden skal være lik panelkunden. På denne måten så blir dashboardet endret etter hvilken kunde som blir valgt.

4.4.7 Grafanas funksjoner

Grafana tilbyr også sine egne funksjoner som kan settes inn i spørringen. De funksjonene vi brukte er “\$__timeGroup()” og “\$__timeFilter()”. Disse to håndterer tidsintervaller i modulen og er adaptive med tidsfilteret Grafana har.



<code>\$__timeGroup(dateColumn, '5m', fillvalue)</code>	Will be replaced by an expression usable in GROUP BY clause. Providing a fillValue of NULL or floating value will automatically fill empty series in timerange with that value. For example, FLOOR(DATEDIFF(second, '1970-01-01', time)/600)*600 as time.
<code>\$__timeFilter(dateColumn)</code>	Will be replaced by a time range filter using the specified column name. For example, dateColumn BETWEEN '2017-04-21T05:01:17Z' AND '2017-04-21T05:06:17Z'

Hentet fra <https://grafana.com/docs/features/datasources/mssql/#macros>

De aller fleste modulene i Grafana krever at man har en kolonne som definerer tiden. Det vil si at disse to funksjonene går igjen i så å si alle spørringene. TimeGroup-funksjonen blir brukt i SELECT-delen av spørringen og tar to parametere. Den første parameteren er navnet på tidskolonnen, som skal være av typen datetime eller som et antall unix epoch sekunder. Den andre sier hvor store tidsrom som skal grupperes sammen.

TimeFilter brukes i WHERE-klausulen og filtrerer ut rader basert på det globale tidsfilteret i Grafana og tar inn navnet på tidskolonnen som parameter.

Eksempel på funksjon:

```
$__timeGroup(event, 30s) as time
```

Gjøres om av Grafana til følgende kode.

```
FLOOR(DATEDIFF(second, '1970-01-01', event)/30)*30 as time.
```

“TimeFilter” settes inn i “WHERE” klausulen. Den tar inn en kolonne som variabel. Den setter da at tiden skal være fra en viss dato og tid til en annen dato og tid. Dette blir definert av tidsfilteret nevnt ovenfor.

Eksempel på funksjon;

```
$__timeFilter(event)
```

gjøres om til

```
event BETWEEN '2019-04-29T08:12:44Z' AND '2019-04-30T08:12:44Z'.
```


4.5 Verktøy

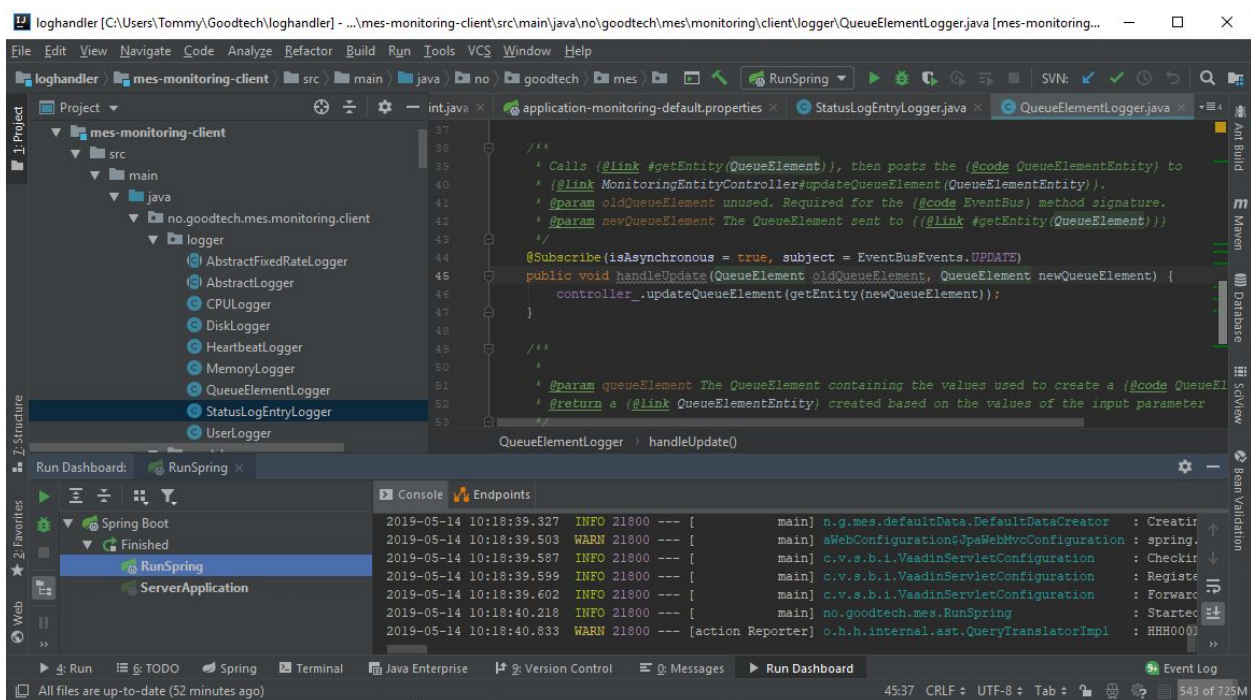
4.5.1 IntelliJ IDEA

Vi har benyttet det integrerte utviklingsmiljøet (IDE) IntelliJ IDEA¹¹ fra JetBrains for å kode i Java. Dette utviklingsmiljøet har støtte for forskjellige andre teknologier og verktøy som brukes i dette prosjektet, som Apache Maven for å håndtere avhengigheter, bygging, pakking og installasjon av Java-prosjekter.



Vi har også brukt IntelliJ sin integrerte støtte for Spring-rammeverket, med konfigurering av forskjellige Spring Boot applikasjoner og et eget Run Dashboard med mulighet for å kjøre, avslutte og debugge individuelle applikasjoner med få knappetrykk.

I tillegg har IntelliJ støtte for versjonskontrollsystemet Subversion (SVN) for å kontinuerlig lagre oppdaterte versjoner av programvaren og laste ned oppdateringer fra en sentral server.



¹¹ <https://www.jetbrains.com/idea/>, logo hentet fra https://en.wikipedia.org/wiki/File:IntelliJ_IDEA_Logo.svg

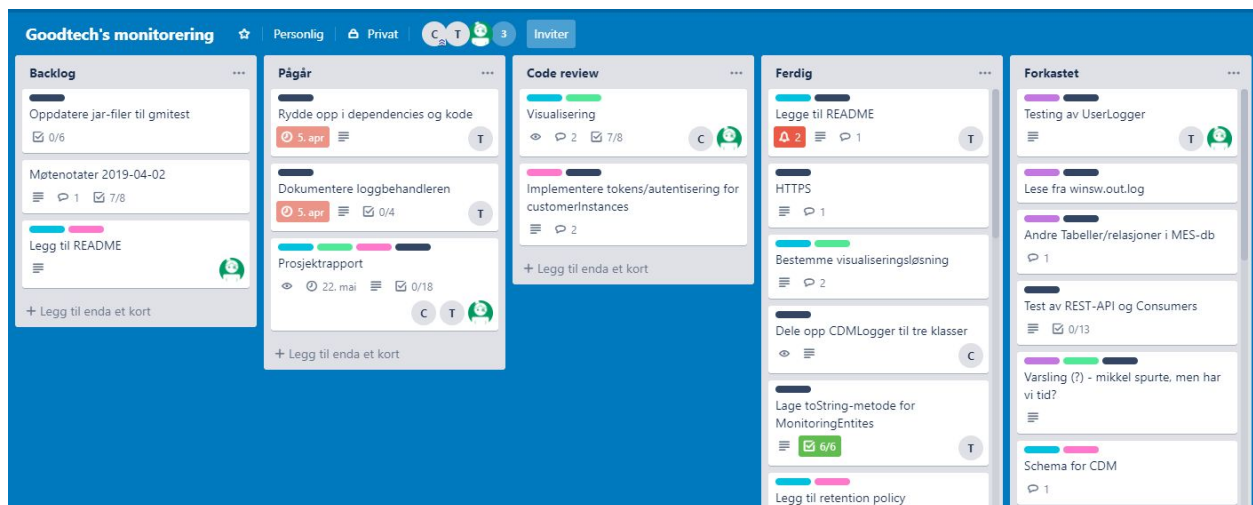
4.5.2 Trello



Vi brukte Trello¹² som planleggingsverktøy.

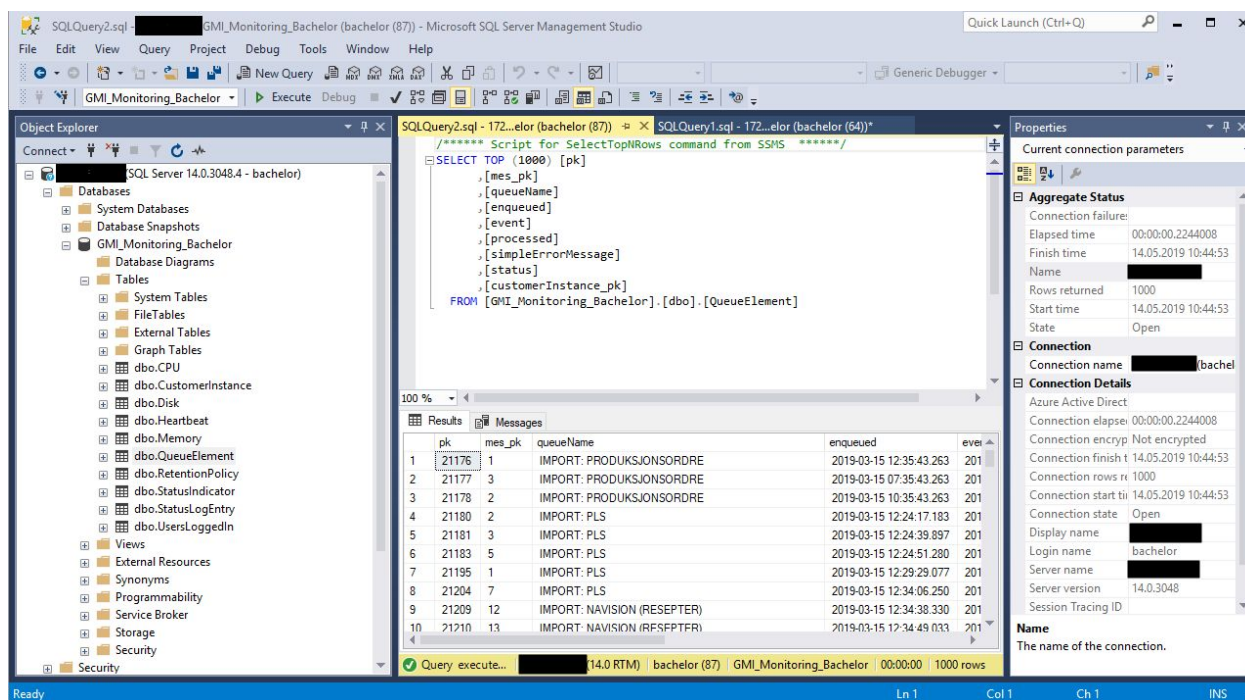
Vi brukte dette som et kanban-board der vi startet med å liste opp alle oppgavene som skulle bli gjort i "Backlog". Oppgavene ble flyttet til "Pågå" da de ble påbegynt av et gruppemedlem, og videre til "Code review" da de var ferdige og klare for gjennomgang av andre i gruppen.

Dette ga oss en måte å holde orden på hva som ble gjort og av hvem, samt sette dato på når en oppgave var antatt ferdig eller eventuelle frister når det skulle vært ferdig.



¹² <https://trello.com/>, logo hentet fra https://en.wikipedia.org/wiki/File:Trello_logo.svg

4.5.3 Microsoft SQL Server Management Studio 2017



For å jobbe med databasen kan man enten bruke databasesystemets konsoll, som lar en skrive SQL-spørringer direkte til databasen lokalt fra maskinen databasen er på, eller så kan en bruke et databaseadministrasjonsverktøy, som Microsoft SQL Server Management Studio 2017 (SSMS)¹³. SSMS lar en fortsatt skrive spørringer direkte til serveren, men som også gir en mer grafisk oversikt over databasens struktur, dens tabeller, og tabellenes innhold. SSMS viste seg derimot å være meget treg på innføring av mange rader på en gang, så vi endte opp med å kun bruke SSMS for å teste spørringer og administrere databasen i seg selv.

13

<https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017>

4.5.4 TortoiseSVN



TortoiseSVN

TortoiseSVN¹⁴ er en grafisk klient til versjonskontrollsystemet

Subversion. I dette programmet kan man bla gjennom forskjellige repositories på en server, laste ned koden, gjøre endringer og oppdatere, samt se forskjellig statistikk. Man har i tillegg mulighet til å se på logger tilhørende endringene i prosjektet, og tilbakestille prosjektet til eldre versjoner.

4.5.5 Cron

Om en ønsker å kjøre et program, et script, en kommando eller lignende periodisk på en Linux-maskin kan en bruke programmet cron. Cron er en oppgaveplanlegger som kan utføre gitte oppgaver med gitte tidsmellomrom. En kan konfigurere tidsmellomrommene helt ned til minuttforskjeller eller helt opp til måneder.

¹⁴ <https://tortoisesvn.net/>, logo hentet fra https://commons.wikimedia.org/wiki/File:TortoiseSVN_logo.svg

5 Endring av database

5.1 Første forsøk med MSSQL

De første to ukene av utviklingen benyttet vi en MSSQL database satt opp på en av Goodtech sine testservere. Vi opplevde MSSQL som ganske tregt da vi forsøkte å legge inn testdata supplert av Goodtech. Vi benyttet Microsoft SQL Server Management Studio 2017 (SSMS) for å føre inn data i databasen, samt teste ut spørringer. SSMS var begrenset til å kunne føre inn på det meste 20000 rader på en gang. Ytterligere rader måtte føres inn i en ny spørring. Det tok i tillegg veldig lang tid, opp mot 2 minutter, for å føre inn 20000 rader med SSMS. Vår originale mistanke rundt disse treghetene var at MSSQL i seg selv var en treg database. Dette var en mistanke vi hadde uten noen reell testing eller investering, og ville vise seg i ettertid å være grovt feil.

5.2 InfluxDB-forsøket

På anbefaling av Goodtech utforsket vi så InfluxDB, en tidsseriedatabase som virket som om var mer egnet for oppgaven vår, da oppgaven i stor grad går ut på å loggføre informasjon over tid. Vi byttet dermed over til InfluxDB, noe som førte til at vi måtte endre på store deler av prosjektet på områder som databaseintegrasjon og spørringer i visualiseringsplattformen, samt det generelle databaseoppsettet. InfluxDB er enklere å håndtere enn MSSQL på mange måter, men det setter en tydelig og begrensende forskjell mellom "tags" og "fields" i den interne datamodell. En "tag" er en del av den identifiserende dataen for en gitt rad (kalt et "punkt" i InfluxDB), og kan ikke behandles på samme måte som et "field", som er en måling gjort i den gitte raden. Dette gjorde at vi ofte måtte sette en verdi både som en tag og som et field. InfluxDB-spørringene var hovedsakelig kortere enn den ekvivalente MSSQL-spørringen for enkle spørringer, men mer kompliserte spørringer, hvor man normalt sett ville brukt "JOIN" eller lignende i MSSQL, ble eksponentielt mer kompliserte i InfluxDB da InfluxDB ikke støtter "JOIN", kun delspørringer. Selv delspørringene var meget begrensede i hva de kunne gjøre, da vi ikke kunne ha delspørringer i "SELECT"-delen av flere spørringer hvor vi hadde behov for det.

5.3 Andre forsøk med MSSQL

Som følger av alle disse vanskene med InfluxDB byttet vi tilbake til MSSQL etter to uker med InfluxDB. Denne overgangen tilbake til MSSQL gikk betraktelig bedre enn overgangen til InfluxDB, da vi kunne gjenbruke mesteparten av det vi hadde fra før vi byttet til InfluxDB i første omgang. Det var på dette tidspunktet, hvor vi gjenbrakte script for å føre inn testdata, at vi oppdaget at treghetene med MSSQL lå i SSMS i stedet for databasen. 98000 rader med data kunne føres inn på under ett sekund med scripting i stedet for gjennom SSMS. MSSQL lot oss

også skrive meget avanserte spørringer i forhold til InfluxDB, som blant annet “antall feil siste timen relativt til gjennomsnittlig antall feil per time de siste tolv ukene for en gitt kø”.

6 Testdata

Gjennom hele utviklingsprosessen har vi hatt behov for testdata for å utvikle selve visualiseringsløsningen, og for å utforske nøyaktig hva Goodtech ønsker å visualisere. Goodtech supplerte oss først med 1000 rader med testdata fra StatusLogEntry- og QueueElement-tabeller til å utvikle visualiseringsløsningen rundt. Disse testdataene ga derimot et urealistisk bilde av hva slags feil som forekommer i ekte systemer, så vi fikk ytterligere 95000 rader med StatusLogEntry og 50000 rader med QueueElement for å ha mer å jobbe med.

Vi byttet så over til InfluxDB kort tid etter dette etter å ha undersøkt og testet InfluxDB på mindre datasett. Dataene vi hadde mottatt fra Goodtech var i form av SQL-spørringer, og egnet seg ikke for InfluxDB da InfluxDB ikke støtter alminnelige SQL-spørringer. Vi måtte dermed omforme disse spørringene til InfluxDB-spørringer, noe vi gjorde med regulære uttrykk direkte i spørringene mottatt fra Goodtech. Vi brukte så NodeJS¹⁵ med Javascript for å føre inn dataene. Javascript er et programmeringsspråk opprinnelig tiltenkt å bruke på nettsider på Internett, men med NodeJS kan en kjøre Javascript på lokale datamaskiner i stedet. Vi valgte å bruke NodeJS da det er enkelt å sette opp, krever ingen kompilering slik som Java gjør, og Javascript er et dynamisk typet språk som lar oss sette opp et program for å føre inn dataene veldig kjapt og enkelt, uten å måtte bevise for en kompilator at programmet er korrekt på noen måte.

Ca. en måned senere byttet vi tilbake til MSSQL. Vi hadde fortsatt testdataene fra Goodtech i den gamle MSSQL-databasen, men ønskene våre for tabellstruktur hadde endret seg veldig mye siden vi sist brukte MSSQL. Vi eksporterte derfor all testdata fra StatusLogEntry og QueueElement til JavaScript Object Notation (JSON)¹⁶-filer. JSON er et format for tekstfiler for lesing og overføring av data mellom systemer. Som navnet tilsier, passer JSON veldig godt med Javascript, så vi benyttet NodeJS og Javascript igjen for å lese JSON-filene, endre på dataene slik vi ønsket, for så å føre inn testdataene i de nye tabellene våre. Vi oppdaget derimot at SSMS, som vi brukte for å eksportere tabellene som JSON-filer, brøt opp kolonner med mye tekst i flere objekt-nøkler i JSON-filene. Dette korrigerste vi med regulære uttrykk i selve JSON-filene, og så kunne de leses inn som forventet.

Siden vi da endelig hadde testdataene i databasen kunne vi bruke testdataene til å lage mer testdata. Vi gjorde dette ved å lage en test-monitoreringsinstans som meldte realistiske feilmeldinger inn i databasen tatt fra disse testdataene fra Goodtech. Vi kunne også føre inn genererte CPU-, minne-, disk- og påloggede brukere-målinger med en random-walk-algoritme vi utviklet, hvor en gitt måling var en endring fra den forrige målingen, i stedet for å være helt

¹⁵ <https://nodejs.org/en/>

¹⁶ <https://www.json.org/>

tilfeldig data. Med denne test-instansen kunne vi også teste Heartbeat-målingen, i tillegg til å se faktiske data for StatusLogEntry og QueueElement uten å endre på tidsstemplene på de eksisterende testdataene i databasen.

7 Prosess

7.1 Oppgavefordeling

I begynnelsen holdt vi oss til enkel parprogrammering for backend-delen av oppgaven. Etter at vi fikk tilgang til Goodtech sin MSSQL-server fordelte vi oss mer naturlig ut over de tre hovedområdene vi har beskrevet gjennom rapporten her: backend, database og visualisering. Vi ble hver av oss ansvarlige for hver vår del av oppgaven på denne måten. Videre fordeling av arbeid rundt hvert område falt mer naturlig for oss da vi i tillegg brukte Trello for å administrere oppgaver som skulle utføres.

Vi opplevde veldig få perioder der vi ikke hadde mye å gjøre innenfor hvert vårt felt. De få gangene vi opplevde at vi hadde lite arbeid innen vårt eget felt hjalp vi til i de andre feltene.

7.2 Backend

7.2.1 Uthenting av data

De første ukene ble brukt på å sette seg inn i Goodtech sitt MES-prosjekt, ettersom vi måtte ha kjennskap til hvordan vi skulle hente ut dataene vi trengte. Under disse ukene fikk vi en kopi av MES-applikasjonen, som vi brukte for å lese dokumentasjon og teste hvilke funksjonaliteter vi kunne benytte for å hente ut data. Vi fikk et tips fra veileder om å se på deres “finder”-API for å hente ut kø-elementer som under kjøretid blir lagt til i en liste i MES.

Finder APIet tilbyr søk i databasen som inneholder elementer av en spesifikk datatype.

Kø-element findereren tilbyr derfor søk i alle kø-elementer som er opprettet av den gitte MES instansen.

Etter å ha implementert uthenting av data med finder APIet begynte vi å lage funksjonalitet for å hente ut informasjon om disk-, minne-, og prosessorforbruk. Disse dataene ble hentet ut for å gi et inntrykk av systemets tilstand over tid. Samtidig kunne vi selv styre hvor ofte vi skulle sende disse dataene til databasen. Dette gjorde at det ble enklere å lage grafer, siden vi genererte en konstant og jevn strøm med logger.

Uthenting av loggene gjennom finder-APIet krevde at vi filtrerte vekk alle elementene i listen som vi allerede hadde sendt videre, for å unngå dobbeltlagring i databasen.

Vi fikk et tips fra veileder om at det var en enda bedre måte å hente data på, hvor vi ikke trengte å håndtere dobbeltlagring på samme måte.

Goodtech har implementert et "Observer"-designmønster (Gamma, Helm, Johnson & Vlissides, 1995, s. 326) med deres EventBus, noe som gjør det mulig å få tilsendt dataene når de blir laget, endret på eller slettet. Vi vil da automatisk motta de endrede eller nye dataene i stedet for å manuelt hente dataene ut. Dette betyr at alle dataene som blir publisert er enten nye eller er endringer av eksisterende data.

7.2.2 Normalisering

Dataene vi hentet fra MES inneholdt mye ekstra informasjon vi ikke hadde behov for å visualisere. Derfor begynte vi å normalisere verdiene. Dette innebar å lage metoder for å sende de dataene vi hadde bruk for, i riktig format, til databasen. Vi laget funksjonalitet for å opprette tilkobling til databasen og sende en SQL spørring med "prepared statements". En "prepared statement", som navnet tilsier, er en spørring som er forberedt på forhånd før potensielt sensitive data eller data som kan tolkes som kjørbare kode, legges inn i spørringen. På denne måten kontrolleres spørringen som blir sendt til serveren på en fornuftig måte hvor vilkårlige strenger med tekst ikke kan tolkes som kjørbare kode av databasen. Vi brukte prepared statements for å verifisere at dataene sendes i riktig format, og at det skal være vanskelig å sabotere eller sende ugyldige spørringer mot databasen.

På dette tidspunktet bestemte vi oss for å benytte InfluxDB istedenfor Microsoft SQL server. Vi måtte derfor lage funksjonalitet for å sende alle dataene vi tidligere sendte til SQL-databasen, til InfluxDB.

InfluxDB har ikke støtte for SQL, siden det ikke er en relasjonsdatabase, men har sitt eget språk "Influx Query Language" (InfluxQL). InfluxQL er et SQL-lignende språk som brukes for å jobbe med InfluxDB.¹⁷

Vi måtte derfor benytte oss av Influx sitt HTTP API for å skrive spørringer til databasen. Det ble laget en klasse for å bygge spørringene som sendes til InfluxDB. Dette ble gjort for å generere spørringer så effektivt som mulig. Dette inkluderte blant annet tilpasning av tekstdata som inneholdt mellomrom eller var på flere linjer.

Cpu-, disk- og minne-data var frem til nå lagret i én tabell med kolonner for hver av verdiene. Vi fant ut at dette ikke var ideelt for visualiseringen, så vi endret lagringen av disse dataene til å settes inn i tre forskjellige measurements.

Etter å ha brukt InfluxDB en kort periode, fant vi ut at denne databasen ikke var best egnet til vårt formål (Les om dette i [5.2 InfluxDB-forsøket](#)). Derfor ble vi enige om å gå tilbake til Microsoft SQL database.

¹⁷ https://docs.influxdata.com/influxdb/v1.7/query_language/spec/

7.2.3 Første omstrukturering

Byttene frem og tilbake mellom databasene gjorde at utviklingen på backend-delen tok litt lengre tid, ettersom vi måtte skrive funksjonalitet for å sende til to forskjellige databaser om hverandre. Likevel kunne vi benytte mesteparten av koden vi skrev tidligere for MSSQL databasen, så byttet tilbake til MSSQL tok ikke så lang tid som til InfluxDB. I tillegg tar vi med oss erfaringer om hvordan det er å bytte database midt i et prosjekt, og styrker og svakheter hos disse to databasetypene.

Vår veileder foreslo at vi skulle se på en generalisert måte for å legge til loggere på, basert på klasse/objekttype. Dette ville gjort det mye enklere å legge til nye loggere i fremtiden.

Vi brukte omtrent en uke på å lese dokumentasjon og prøve å implementere dette. Det viste seg at Event Bussen til MES ikke støttet denne typen generalisering, så vi bestemte oss for ikke å bruke mer tid på dette.

Vi begynte derfor på å lage entitetsklasser som skulle inneholde dataene vi hentet ut fra loggerne, en egen klasse for håndtering av spørringer til Microsoft SQL server, og et interface. Entitetsklassene er klasser som inneholder loggataene som sendes til databasen, hvor et objekt av denne klassen tilsvarer en rad, og attributtene til objektet tilsvarer kolonner i databasen. Interfacet inneholdt en metode som tok imot en connection og skulle returnere en prepared statement med SQL-spørring. Entitetsklassene implementerte så dette interfacet, med egne SQL-spørringer for hver entitet, og tilhørende verdier.

Samtidig som vi endret på loggerne og entitetsklassene, begynte vi å bruke mer og mer av Spring-rammeverket sin funksjonalitet. Vi benyttet oss av beans, auto-wiring og konfigurasjoner, samt properties-filer for å laste inn verdier som applikasjonen brukte. Disse verdiene kunne ikke hardkodes inn i prosjektet, ettersom forskjellige brukere av applikasjonen skulle ha forskjellige verdier, som brukernavn, passord og hvilke disker/partisjoner som skulle loggføres. Videre flyttet vi deklarasjoner av Spring-beans ut fra XML-filer, som vi startet med, og inn i Java-konfigurasjon, dvs. Javakode. Dette er mer fremtidsrettet, gjør det lettere å feilsøke og videreutvikle, og er i tillegg det Goodtech ønsket at vi skulle gjøre.

7.2.4 Andre omstrukturering

Vi hadde til nå hatt én Java-modul, mes-monitoring-client, som hadde en tilkobling direkte mot databasen som lå på serveren hos Goodtech. Dette er ikke optimalt, siden det krever at databasen kan nås utenfra. Det gjør at databasen blir mer utsatt for potensielle angrep og feil. Vi bestemte oss derfor å lage et REST API.



Første versjon av programmet hentet data med loggere, som deretter sendte dataene direkte til databasen

REST APIet ble laget i en ny modul, mes-monitoring-server, som er et nytt Spring-prosjekt, utenfor loggerne, og har derfor ingen direkte relasjon til MES, utenom en avhengighet mot entitetene som ligger i logger-prosjektet. Vi prøvde å flytte entitetene ut til server-siden, ettersom det er det mest naturlige, men det resulterte i en del problemer med Maven og avhengigheter der. Disse problemene vil løse seg når prosjektet blir integrert av Goodtech inn i MES, og man vil kunne flytte entitetene.

REST APIet tar imot data fra klientene gjennom HTTPS, som er mye tryggere enn direkte tilkobling til databasen. Blant annet fordi vi kan kryptere trafikken og validere dataene som er sendt inn til APIet, og at man ikke får mulighet til å skrive egen SQL.

For å få sendt dataene fra REST APIet til databasen benyttet vi oss av Java Database Connectivity (JDBC) -APIet og Data Access Object (DAO), i tillegg til consumers med køer for entitetene som blir sendt gjennom REST APIet.

Data Access Object er et type objekt som har funksjonalitet knyttet til databasehåndtering, som innsetting, oppdatering, uthenting og sletting av data.

Consumer-klassen er generisk, og kan derfor benyttes av alle entitetene, og brukes for å legge entitetene i kø for å unngå samtidige databasekall mot tabellene som tilhører den enkelte entiteten. På den måten sikrer vi at én og én entitet blir lagt til i databasen om gangen, og at serielle spørringer ikke har påvirkning på hverandre. Køene er av typen `LinkedBlockingQueue`¹⁸, og er derfor trådsikre.

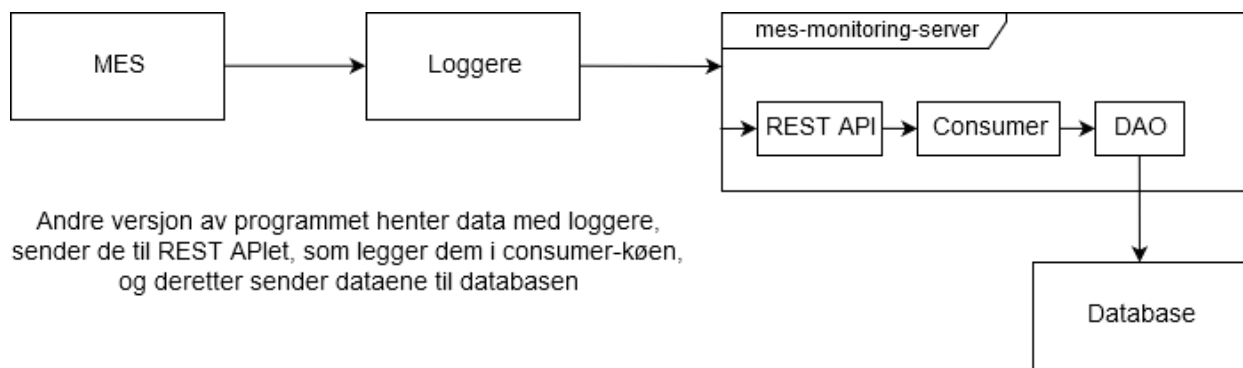
Det er kritisk at et system er trådsikkert i et tilfelle som dette, da det opprettes en ny tråd for hver av forespørslene mot APIet, og hvert punkt i APIet som kan motta data har egne tråder. Disse punktene legger dataene inn i en `LinkedBlockingQueue` etter å ha mottatt og validert dataene. Om vi hadde brukt en alminnelig `LinkedQueue`, uten trådsikkerhet, kan en tråd overskrive det en annen tråd nettopp har skrevet. Men med en trådsikker `LinkedBlockingQueue` vil integriteten til køen opprettholdes ved å få trådene til å vente i tur og orden med å skrive til og hente ut fra køen. Entitetene som ligger i køen tas ut og det blir utført et databasekall gjennom entitetens tilhørende DAO. Det blir instansert én consumer for hver type entitet, ved bruk av Spring-beans og en generisk consumer-klasse.

Hver enkel entitet har en egen tilhørende DAO-klasse som implementerer et generisk grensesnitt. Dette grensesnittet beskriver funksjonalitet for å legge til og oppdatere rader i databasen basert på hvilken type entitet som de implementerende klassene skal benytte.

¹⁸ <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html>

Denne funksjonaliteten benytter JDBC-APIet gjennom en JDBC-mal (NamedParameterJdbcTemplate¹⁹).

Denne malen inneholder en del funksjonalitet knyttet til oppbyggingen og utførelsen av SQL-spøringer med prepared statement, med parametere basert på entiteten som spørringen angår.



Bruken av REST APIet gjør at all trafikk mot databasen går gjennom REST APIet først. Så blir det sendt til en kø i consumeren, som deretter henter ut entiteten fra køen og kaller på databasekall-metoden i DAOen.

Vi hadde nå flyttet ut all databasehåndtering fra mes-monitoring-client og inn i server-prosjektet. Med våre to moduler, klient og server, følte vi at vi hadde et mer eller mindre ferdig produkt. På dette tidspunktet, som fortsatt var noen uker før innlevering, kom Goodtech med et forslag, og ønske, om autentisering av brukere.

7.2.5 Sikkerhet

Ettersom sikkerhet, med autentisering og kryptering av datatrafikk ikke hadde blitt eksplisitt nevnt i kravspesifikasjonen, hadde vi heller ikke hatt et særlig fokus på akkurat dette tidligere. Men etter et forslag fra Goodtech ble vi enige om å implementere disse tiltakene. Både for løsningens totale sikkerhet, og ikke minst for egen læring.

I både autentiseringsløsningen og kryptering av trafikk benyttet vi oss av Spring Security-rammeverket. Vi brukte mye tid på å sette oss inn i dette rammeverket, men fikk veldig god hjelp av dokumentasjon og eksempelkode²⁰ fra Spring.

Vi startet først med å utvikle autentiseringsløsningen, ettersom vi fikk vite at det å implementere HTTPS-protokollen ikke skulle være så komplisert som vi først hadde fryktet.

¹⁹

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/namedparameter/NamedParameterJdbcTemplate.html>

²⁰ <https://spring.io/guides/topicals/spring-security-architecture/>

Hovedproblemet med autentiseringen var at det meste av referanselitteratur var basert rundt menneskelig håndtering av innlogging, noe som ikke var tilfelle i vårt prosjekt. Vi måtte derfor lage noen ekstra klasser og metoder for å håndtere autentisering gjennom HTTP-forespørslene til REST APIet.

Vi endte opp med å benytte “basic access authentication”, hvor brukernavn og passord blir slått sammen og sendt som en kodet tekst i pakken. Etter mye prøving og feiling med forskjellige variasjoner av konfigurasjoner, klarte vi til slutt å få til en fungerende autentiseringsløsning med brukernavn og passord sendt fra klient-applikasjonen til serveren.

Med den fungerende autentiseringsløsningen på plass, hadde vi fortsatt tid til å se på krypteringen av trafikken. Kryptering var helt essensielt for å hindre at brukernavn og passord lekkes, ettersom basic access authentication inkluderer brukernavn og passord, dog i base64-encoding, i headeren til HTTP-forespørslene.

Det største hinderet ved å implementere kryptering av trafikken var at vi brukte et “self-signed certificate”, dvs. et sertifikat vi selv utstedte og godkjente, uten en troverdig tredjepart til å verifisere sertifikatet. Vi måtte derfor lage en egen klasse for å tillate denne typen sertifikater i mes-monitoring-client-prosjektet.

Testing av autentisering og kryptert trafikk gikk tilnærmet smertefritt, og vi hadde nå en fungerende løsning hvor vi hentet ut dataene fra MES og kunde-prosjektet, sendte disse til serveren, ble autentisert og videresendt til REST APIet, som i tur behandlet og satte inn dataene i databasen.

Den gjenværende tiden ble brukt til å kvalitetssikre og dokumentere modulene, for at det skulle være lettere å kunne utvide disse videre ved en senere anledninger.

7.3 Database

Utviklingen av databaseskjema var veldig kort overstått ved at vi rett og slett tok mye av oppsettet fra MES-ens egne StatusLogEntry- og QueueElement-tabeller, la til en CustomerInstance-tabell og en CDM-tabell for lagring av CPU, disk og minne. Da vi byttet over til InfluxDB delte vi CDM-tabellen opp i CPU-, Disk- og Memory-tabellene (eller “measurements”, som InfluxDB kaller dem).

Etter vi byttet tilbake til MSSQL igjen beholdt vi oppdelingen av CPU-, Disk- og Memory-tabellene, og la til RetentionPolicy-tabellen for å holde rede på hvor lenge data skal lagres og Heartbeat-tabellen for å monitorere selve Goodtech Monitoring-systemet. Vi la også til “hash”-kolonnen til CustomerInstance for å lagre passordhashene til hver kundeinstans.

Resten av tiden brukt rundt databasen gikk til å føre inn testdata.

7.4 Visualisering

Vi har hatt relative frie tøyler til å jobbe med og gjøre det vi ville i denne oppgaven. Det ble ikke stilt noen krav til visualiseringen til å begynne med annet enn at informasjon fra StatusLogEntry og QueueElement-tabellene i MES-enes database og generell helse på kundenes systemer skulle bli visualisert. Vi stilte selv ingen andre krav i forprosjektet annet enn at vi skulle bruke Grafana. Goodtech anbefalte oss Grafana tidlig i planleggingsstadiet grunnet deres personlige erfaring med visualiseringsplattformen.

7.4.1 Målene med visualiseringen

- En infotavle/skjerm som visualiserer den viktigste informasjonen om kundens instanser.
- Klare og enkle visualiserings-momenter som viser når og hvor noe har gått alt.
- Et detalj panel som hvor, når og hva som har feilet.

Uten noen konkret kravspesifikasjon å følge gikk mye av startfasen til å prøve og feile med forskjellige visualiseringsmoduler som kunne itereres på videre. Vi lagde flere moduler som vi presenterte til hverandre innad i gruppen. Vi presenterte også modulene underveis til Goodtech, som hjalp oss med tilbakemelding om det var brukbart eller om det trengte justeringer. Gjennom denne iterative utviklingsprosessen med hyppig tilbakemelding oppsto det både idéer til nye grafer fra oss i gruppen samt veileder. Dette fungerte i dette prosjektet noe på grunn av størrelsen på oppgaven og at vi alltid var tilstede på Goodtechs lokaler.

Et av de store målene med visualiseringen var å ha en presentabel og enkel fremvisning av kritiske feil på alle kundeinstansene. Vi gjorde dette ved å ta hensyn til hvor ofte hver kø i QueueElement og hver statusindikator i StatusLogEntry feiler, og markere køer og indikatorer som feiler oftere enn vanlig som kritiske.

7.4.2 Startfasen

Vi begynte med å skrive spørringer mot MSSQL-databasen. Vi satte opp en lokal Grafana-instans på en av våre maskiner da vi ennå ikke hadde fått tilgang på en test server. Rundt en måned inn i prosjektet ble spørringer en del mer komplisert og det tok mer tid å utføre spørringer enn ønskelig. Dette viste seg i ettertid å skyldes dårlige spørringer. Treghetene, samlet med problemet nevnt ovenfor angående "Microsoft SQL Server Management Studio 2017"²¹ så kom vi til konklusjonen at å skifte til InfluxDB potensielt kunne fikse flere av problemene som vi støtte på. De fleste spørringer var enklere og kjappere i influxDB som virket lovende da. Det ble da brukt ca. en uke på å endre de modulene og endre oppsettet i Grafana fra MSSQL til InfluxDB.

²¹ [Microsoft SQL Server Management Studio 2017](#)

7.4.3 Etter skiftet

Etter en måned med InfluxDB skiftet vi tilbake til MSSQL. Vi prøvde oss også på rekursive spørringer, noe InfluxDB også ikke har støtte for. Det å lære seg InfluxQL, InfluxDBs språk for spørringer mot en InfluxDB-database, og i tillegg et nytt databaseoppsett ble en større utfordring enn antatt. Vi hadde allerede de fleste modulene i MSSQL fra skiftet så det tok ikke lang tid å bytte tilbake til MSSQL. Vi har mer erfaring med SQL fra studiene våre, hvor vi brukte MySQL, slik at å skrive spørringer for MSSQL er betraktelig enklere for oss. Vi så ikke at vi tjente noe på å prøve videre med InfluxDB med tanke på å lære seg en ny syntaks og en annen form å tenke ut spørringer på, selv om vi i ettertid ser at det som vi oppnådde i MSSQL er mulig å oppnå i InfluxDB.

Mye av den resterende tiden ble brukt på “QueueElement”, “StatusLogEntry” og “Errors or warnings in a row”(disse blir forklart i detalj i “Produkt - Visualisering”)²² modulene da dette var de vanskeligste modulene av visualiseringen og de spørringene som var mest utfordrende. Mye av utfordringen vår å få spørringene til å stemme overens med hva som var tanken bak hver av disse modulene. En annen utfordring var å få dataen som feilet ofte til å se lik ut som dataen som feilet sjeldent. Dette var viktig fordi det er mange prosesser som har til vane å feile ofte og andre ikke. Dette blir taklet ved å bruke gjennomsnitt av antall feil og logaritmiske grafer der vi så at dette kunne bli brukt. Dette blir nærmere forklart i “Produkt - Visualisering”.

7.4.4 Tidssoner

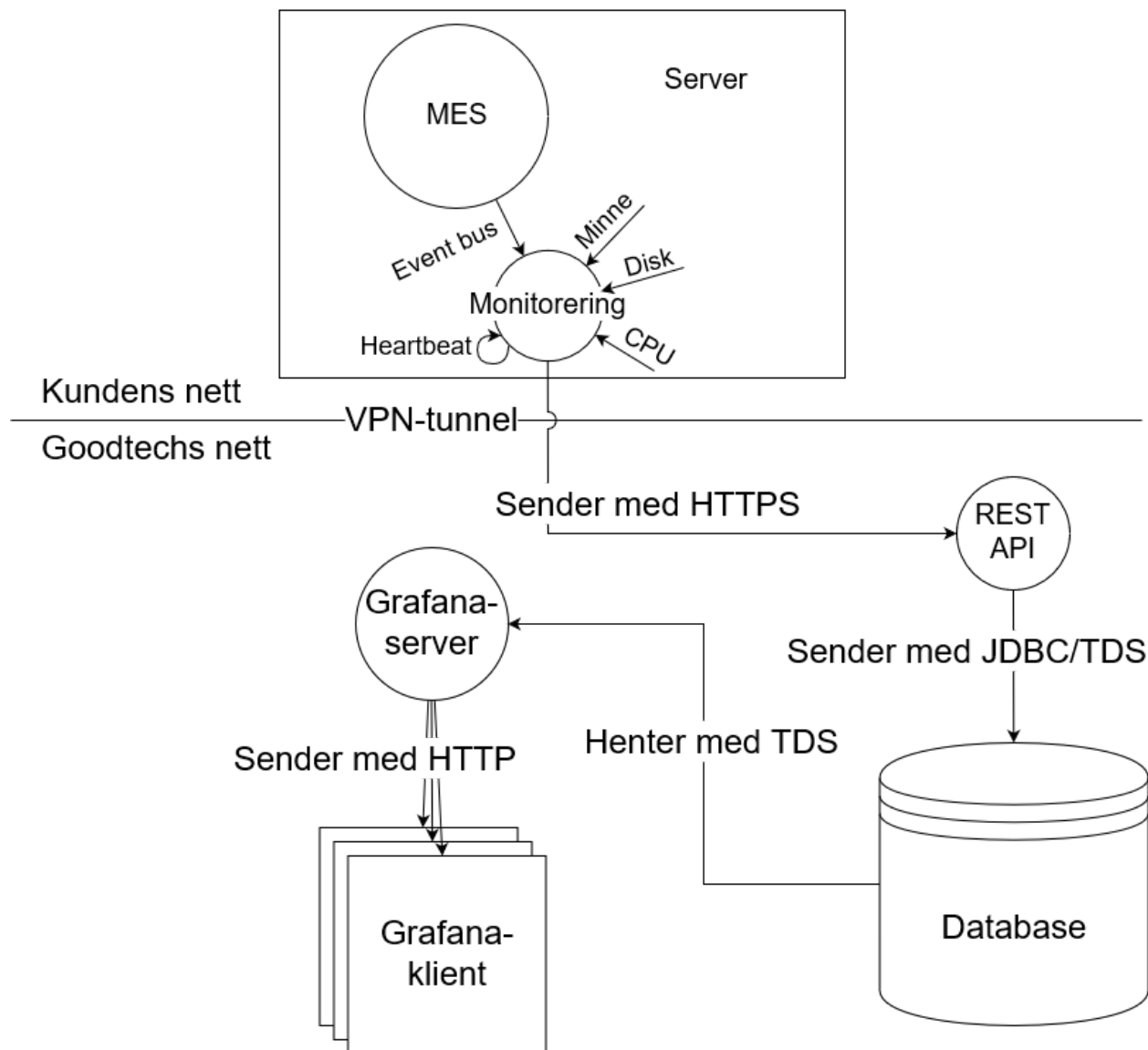
En problemstilling som kom gjentagende ganger var behandling av tidssoner. Grafana antar at alle tidspunkter gitt fra spørringer er i “Unix Epoch seconds”, enten i form av faktiske antall sekunder eller som SQL DateTime-typen²³. Unix Epoch er en måte å representere tid ved å si at et tidspunkt forekomm et antall sekunder etter kl. 00:00 den 1. januar 1970 UTC. Etter å ha jobbet med prosjektet en stund merket vi at tiden som var i grafene alltid var forskjøvet en time fremover. Vi endret så alle tidspunkter i databasen til deres korresponderende tid i UTC tidssonen og satte Grafana til “local browser time”. Dette fikset problemet i første omgang.

Da tiden i Norge ble skiftet til sommertid oppdaget vi det samme problemet igjen. Det var her vi fant funksjonen “GETUTCDATE” som MSSQL tilbyr. Grunnen til at denne funksjonen ikke ble funnet i første omgang var at vi akkurat hadde skiftet til InfluxDB som da ikke hadde noen løsning på dette. Ved å sette inn “GETUTCDATE” de stedene vi hadde brukt “GETDATE” og “CURRENT_TIMESTAMP” så hentet det ut riktig tid for “nå”.

²² [Produkt - Visualisering](#)

²³ <https://grafana.com/docs/features/datasources/mssql/#time-series-queries>

8 Dataflyt



Dataene vi håndterer stammer i all hovedsak fra MES-ene ute hos hver av Goodtech sine kunder, hvor en kunde kan ha flere "kundeinstanser". Vi henter i tillegg ut systeminformasjon, som blant annet CPU-, disk- og minnebruk.

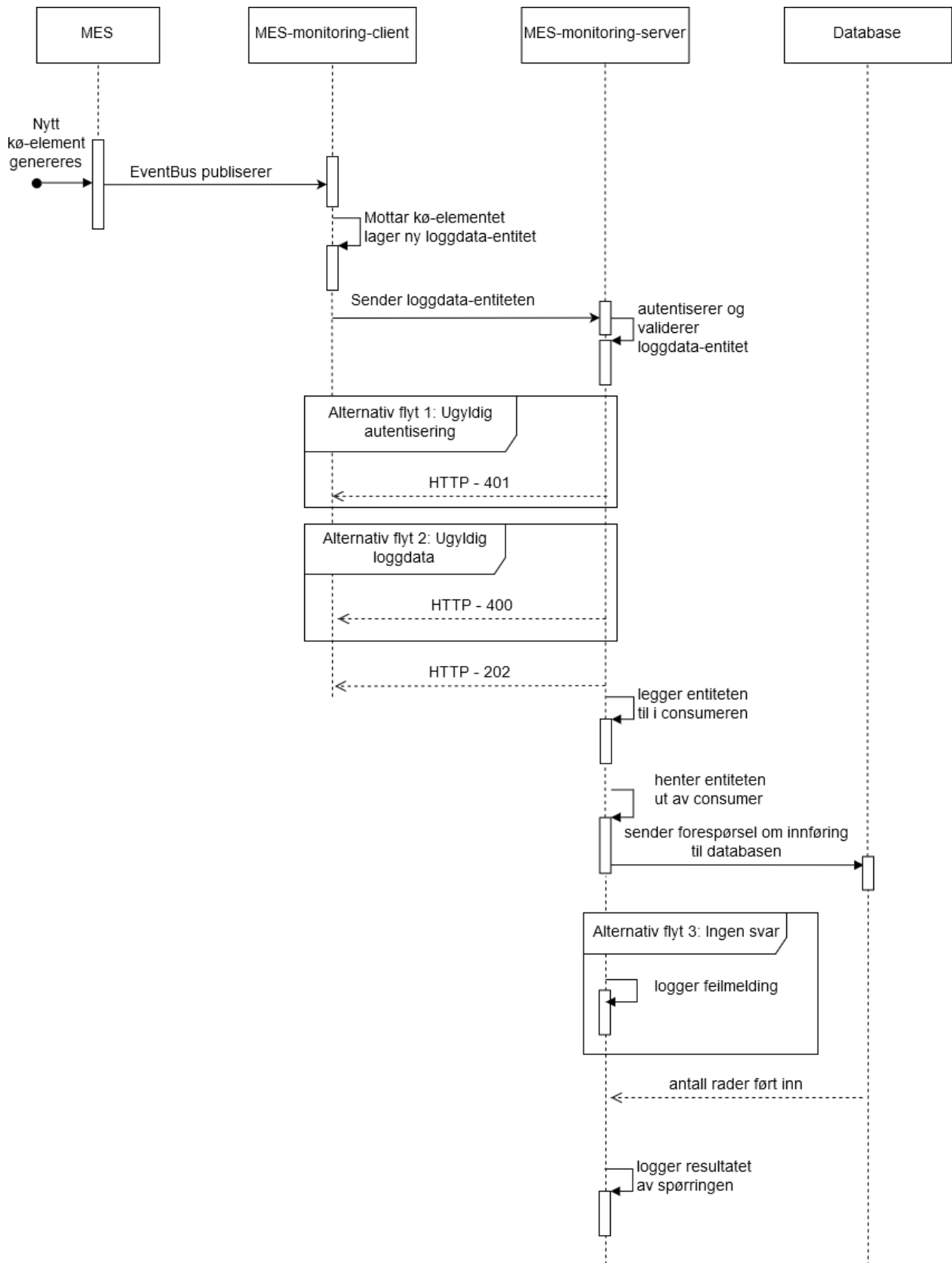
Dataene fra selve MES-en hentes ut fra en hendelsesbuss (event bus) satt opp av Goodtech i MES fra før av. På denne bussen kan vi lytte etter opprettelse og endring av StatusLogEntry- og QueueElement-objekter.

Hver rad med data blir sendt til mes-monitoring-server REST APIet. En gitt kundeinstans fungerer som et brukernavn, og har et tilknyttet passord som APIet verifiserer før den godtar de innkommende dataene. APIet sender så dataene videre internt, før det til slutt sendes til databasen gjennom Java Database Connectivity (JDBC) driveren til MSSQL²⁴, som bruker Tabular Data Stream (TDS)-protokollen (Microsoft, 2019) for å kommunisere med databasen.

Når en klientmaskin navigerer til Grafana-websiden, gjør websiden et kall til Grafana-serveren i form av en HTTP-forespørsel på dataene den forsøker å vise. Grafana-serveren gjør så et kall til selve databasen gjennom TDS, henter ut de relevante dataene, og sender dem så tilbake til klienten som en HTTP-respons.

Illustrasjonen på neste side er et sekvensdiagram som forklarer den typiske dataflyten til et gitt QueueElement som genereres av en MES-instans som kjører hos en kunde av Goodtech.

²⁴ <https://github.com/microsoft/mssql-jdbc>



9 Produkt

9.1 Server-instans

Serveren vi har REST APIet og databasen kjørende på er en Ubuntu Server 18.04.2 LTS²⁵ virtuell maskin (VM) i et VM-miljø Goodtech har. Ubuntu er et operativsystem bygget på Linux-kjernen, og krever lite konfigurasjon etter installasjon for å fungere slik vi ønsker det. Vi installerte først Grafana, og deretter InfluxDB da vi byttet database. Vi byttet som tidligere nevnt tilbake til MSSQL, og denne gangen installerte vi MSSQL på selve server-instansen i stedet for å bruke MSSQL-serveren supplert av Goodtech. Dette lot oss ha full kontroll på serveren, i tillegg til å samle ting på ett sted med både REST APIet og databasen på en og samme server. Ytterligere installerte vi NodeJS for kjøring av Javascript-kode, og Yarn for administrering og nedlasting av biblioteker og lignende for Javascript.

9.2 Backend

Mes-monitoring består av to Java-applikasjoner; mes-monitoring-client og mes-monitoring-server, hvor mes-monitoring-client er en modul som skal integreres inn i Goodtech sitt MES-prosjekt, mens den andre, mes-monitoring-server, er en selvstendig Spring Boot applikasjon.

Begge disse applikasjonene har avhengigheter mot kode som Goodtech har skrevet, og applikasjonene vil derfor ikke være kjørbare uten denne koden.

9.2.1 Mes-monitoring-client

Mes-monitoring-client er modulen som skal integreres inn i Goodtech sitt MES-prosjekt, og inneholder de klassene som brukes for å hente ut loggdata. Klient-modulen er derfor helt avhengig av at MES kjører for å få sendt data til serveren.

Vi bruker to forskjellige typer loggere i mes-monitoring-client; Fixed rate loggere, og EventBus-loggere.

Fixed rate loggerne er generelle loggere som logger tidsseriedata i faste intervaller, dvs. fixed rates. Disse intervallene konfigureres i properties-filen, og intervallene er angitt i antall millisekunder mellom hver gang loggeren sender dataene til serveren.

EventBus-loggere er de loggerne som bruker Subscriber-APIet, dvs. EventBussen, til Goodtech.

²⁵ <https://www.ubuntu.com/download/server>

Vi har implementert følgende loggere:

- Fixed rate loggere:
 - CPULogger
 - DiskLogger
 - MemoryLogger
 - HeartbeatLogger
 - UsersLoggedInLogger
- EventBus-loggere:
 - QueueElementLogger
 - StatusLogEntryLogger

CPULogger og MemoryLogger henter ut data fra operativsystemet ved bruk av Java sin `OperatingSystemMXBean`²⁶ og `Reflection` for å få tak i totalt og ledig minne, samt brukt prosessorkraft.

DiskLogger henter data fra filsystemet ved å lage et nytt `File`²⁷-objekt med en gitt filsti, og henter ut ledig og total plass på disken.

UsersLoggedInLogger har en teller med antallet brukere logget inn i MES-programmet, og funksjonalitet for å øke og senke dette antallet.

HeartBeatLogger logger bare kundeinstans/brukernavn og tidspunkt på heartbeat-signalet. Loggene fra disse fixed-rate-loggerne inneholder tidsstempelen fra når dataene er hentet ut, og genererer dermed tidsseriedata.

EventBus-loggerne mottar `QueueElement`- og `StatusLogEntry`-objekter og henter ut de relevante dataene disse objektene inneholder. Loggene fra EventBus-loggerne inneholder tidsstempler fra når disse loggene er opprettet, eller behandlet, men ikke når de blir sendt. Dette gjøres fordi disse loggene genereres asynkront, og det er viktigere å ha et tidsstempel som samsvarer med Goodtech sitt system fremfor når disse loggene ble sendt.

Alle loggerne oppretter nye instanser av loggdata-entiteter for hver logg som skal sendes til serveren. Hvilken entitet som blir opprettet er basert på hvilke data som skal loggføres. Loggerne sender disse entitetene videre til en REST-kontroller. Kontrolleren sender deretter entitetene til REST APIet som kjører på serveren i skyen.

9.2.1.1 CPULogger

CPU-loggeren loggfører CPU-ytelsen. Dvs. hvor mye prosessorkraft som brukes på det tidspunktet dataene hentes ut. CPU-ytelsen gjengis prosentvis.

²⁶ <https://docs.oracle.com/javase/8/docs/api/java/lang/management/OperatingSystemMXBean.html>

²⁷ <https://docs.oracle.com/javase/8/docs/api/java/io/File.html>

9.2.1.2 DiskLogger

Disk-loggeren loggfører total diskplass og brukt diskplass på en gitt disk/partisjon. Disken, eller partisjonen, angis som en filsti i properties-filen tilhørende mes-monitoring-client-prosjektet. Vi har tilrettelagt for logging av inntil tre disker/partisjoner på hver instans som kjører mes-monitoring-client-programmet. Dette har vi gjort etter ønske fra Goodtech, for å muliggjøre logging av flere partisjoner samtidig.

9.2.1.3 MemoryLogger

Minne-loggeren loggfører totalt minne på systemet, og hvor mye minne som er brukt på det tidspunktet dataene hentes ut.

9.2.1.4 HeartbeatLogger

Heartbeat-loggeren sender en melding til databasen som inneholder tidsstempelen og hvilken bruker denne meldingen kommer fra. Dette gjøres periodisk, for å sikre at man har en kontinuerlig tilkobling mellom klienten og serveren. Dersom det mangler logger av denne typen kan det bety at tilkoblingen er ustabil, eller at det har skjedd et problem med klienten eller serveren som gjør at loggene ikke lenger sendes eller mottas korrekt. Da kan man reagere og feilsøke umiddelbart, uten å måtte vente på at kunden tar kontakt om eventuelle feil på systemet.

9.2.1.5 UsersLoggedInLogger

Etter et forslag fra Goodtech, så implementerte vi en logger som periodisk henter ut antallet brukere som er logget inn i et gitt MES.

Denne loggeren vil sende logger om hvor mange brukere som er logget inn på systemet på det tidspunktet dataene hentes ut. Loggeren er foreløpig ikke implementert fra Goodtech sin side, altså der de håndterer inn- og utlogginger, og gir per nå ingen nyttige data.

Formålet med denne loggeren er å se om det er en sammenheng mellom minne- og CPU-forbruk og antall brukere som er logget inn på samme system, og samtidig holde en oversikt over hvor mange brukere som bruker systemet til enhver tid.

9.2.1.6 QueueElementLogger

Kø-element-loggeren mottar kø-elementer som publiseres av Goodtech sitt MES. Disse kø-elementene kan inneholde informasjon om oppskrifter til produkter, tidspunkt for opprettelse og behandling av kø-elementet, status og annen data. Kø-elementene publiseres av EventBusen og mottas i en metode i loggeren. Der hentes de relevante dataene ut, før de blir brukt til å opprette en ny entitet som sendes til REST APIet.

9.2.1.7 StatusLogEntryLogger

Status-log-entry-loggeren mottar loggoppføringene og behandler dem på samme måte som Kø-element-loggeren. Disse entitetene inneholder informasjon om en statuslogg, og en statusindikator, samt en melding som beskriver statusloggen.

9.2.1.8 Properties

For å unngå å hardkode inn verdier i koden, så bruker vi Spring-rammeverket sine “application.properties²⁸”-filer for å konfigurere applikasjonene.

På denne måten kan vi unngå at sensitiv informasjon som brukernavn og passord blir sendt med programmet. Det er også lettere å konfigurere og endre mellom forskjellige konfigurasjoner når man bruker properties-filer. I stedet for å endre på koden for hver konfigurasjon, kan man velge hvilke properties-filer man skal laste inn når man skal kjøre applikasjonen. Dette gjør også at man kan kjøre flere forskjellige konfigurasjoner av en applikasjon samtidig, uten å måtte endre på selve kildekode til programmet.

Properties er bygget opp på følgende måte:

“navn på property” = “verdi”.

Eksempel: `spring.datasource.url=https://example.com`.

Dette betyr at “example.com” er det stedet spring henter og sender data til og fra.

Mes-monitoring-client bruker blant annet følgende properties for konfigurasjon:

`mes.monitoring.server.url` - Dette er url, eller adressen, til serveren som kjører mes-monitoring-server. Det er til denne adressen alle kallene til REST APIet sendes.

```
# Properties for authentication
mes.monitoring.server.username
mes.monitoring.server.password
```

Brukernavn og passord til serveren brukes for å autentisere et REST API-kall. Dette gjøres for å sikre at det kun er brukere som er lagt til i databasen som kan sende logger til serveren.

```
# Enable/disable mes-monitoring logging using @ConditionalOnProperty
mes.monitoring.enabled
```

Her er det tydelig at properties også har kommentarer, angitt med “#” foran, for å forklare hva de forskjellige properties er og eksempelvis hvilke verdier som er gyldige.

28

<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-application-property-files>

9.2.2 Mes-monitoring-server

Mes-monitoring-server er den selvstendige server-applikasjonen som kjører på en server i skyen. Den inneholder et REST API, consumers med køer til loggdata-entitetene, databasehåndtering av entitetene, og autentiserings- og krypteringsfunksjonalitet.

9.2.2.1 REST API

REST APIet er grensesnittet som tar imot loggdataene på server-siden. Vårt REST API er bygget opp av én kontroller-klasse for hver type loggdata-entitet.

Vi bruker Spring sin “@RestController”-annotering²⁹ på hver av kontrollerne, og “@RequestMapping” på metodene som tar imot HTTP-forespørlene. Dette gjør at man spesifiserer hvilken metode som blir kjørt når man sender en forespørsel til den “mappingen”, eller “routing”-en. Eksempelvis vil “@RequestMapping(“/queueElement”)” bety at det er denne metoden som kjøres når man sender en Http-forespørsel til <https://example.com/queueElement>.

I tillegg til “RequestMapping” har metodene også “@RequestBody”-annotering hvor vi spesifiserer hvilken datatype, altså hvilken loggdata-entitet, som denne metoden skal håndtere. Dette gjør at man enklere kan håndtere objekter som parametere i APIet, ettersom Spring håndterer omforming av Java-objekter til JSON og tilbake automatisk. Da slipper vi eksplisitt å hente ut hver enkelt verdi som kommer i HTTP-forespørselen til REST APIet. Man får også en feilmelding dersom man sender feil datatype til denne metoden/adressen, og forespørselen blir forkastet. Dette er gjort for å unngå prosessering av feil, eller ugyldige, kall til REST APIet.

I metoden som mottar en loggdata-entitet utføres en validering av entiteten. Dersom entiteten er ugyldig, dvs. har feil type data, mangler eller har tomme felter, vil det bli sendt en feilmelding tilbake som forklarer dette. Klienten loggfører da denne feilmeldingen, og har mulighet til å rette opp de nødvendige punktene for at loggene skal bli godkjente.

Dersom entiteten er gyldig, med alle nødvendige data, returneres det en melding om at entiteten er mottatt og akseptert. Entiteten blir deretter sendt til sin respektive consumer for behandling.

9.2.2.2 Consumer

Consumerne inneholder køer av loggdata-entiteter, og fungerer som mellomlagring mellom REST API og databasen. Consumerne kaller på DAO'en tilhørende de respektive entitetene, og

29

<https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-first-application.html#getting-started-first-application-annotations>

frigjør på den måten REST APIet ved at APIet kun mottar og sender entitetene videre uten å utføre noe særlig logikk.

Køene i consumerne er av typen `LinkedBlockingQueue`, og er trådsikre. Vi bruker denne typen køer for å sikre at entitetene blir håndtert kronologisk, og at det ikke oppstår feil på grunn av multi-threading og race-conditions.

Consumerne har metoder for å sette inn og hente ut og prosessere entitetene i køene. Metodene for uthenting og prosessering av entitetene kjøres kontinuerlig på egne tråder, og behandler entitetene med “first-in, first-out”-prinsippet, dvs. kronologisk etter hvert som entitetene legges inn.

Etter at REST APIet sender entiteten til consumeren, som i tur legger den inn i køen, blir entiteten hentet ut. Deretter blir den prosessert ved å sende den videre til DAO'en som da prøver å enten legge til eller oppdatere en rad med entitetens verdier.

9.2.2.3 DAO

DAO, eller Data Access Object, er objekter som fungerer som grensesnitt mot en database. Vi har implementert egne DAO'er for hver type loggdata-entitet. Disse DAO'ene inneholder metoder for innsetting og oppdatering av `QueueElement`-data, og innsetting av alle andre datatyper som vi loggfører. Dette er gjort fordi `QueueElement` er den eneste typen data hvor innholdet endrer seg. De andre entitetene er mer egnet som tidsseriedata, og vil ikke endres over tid.

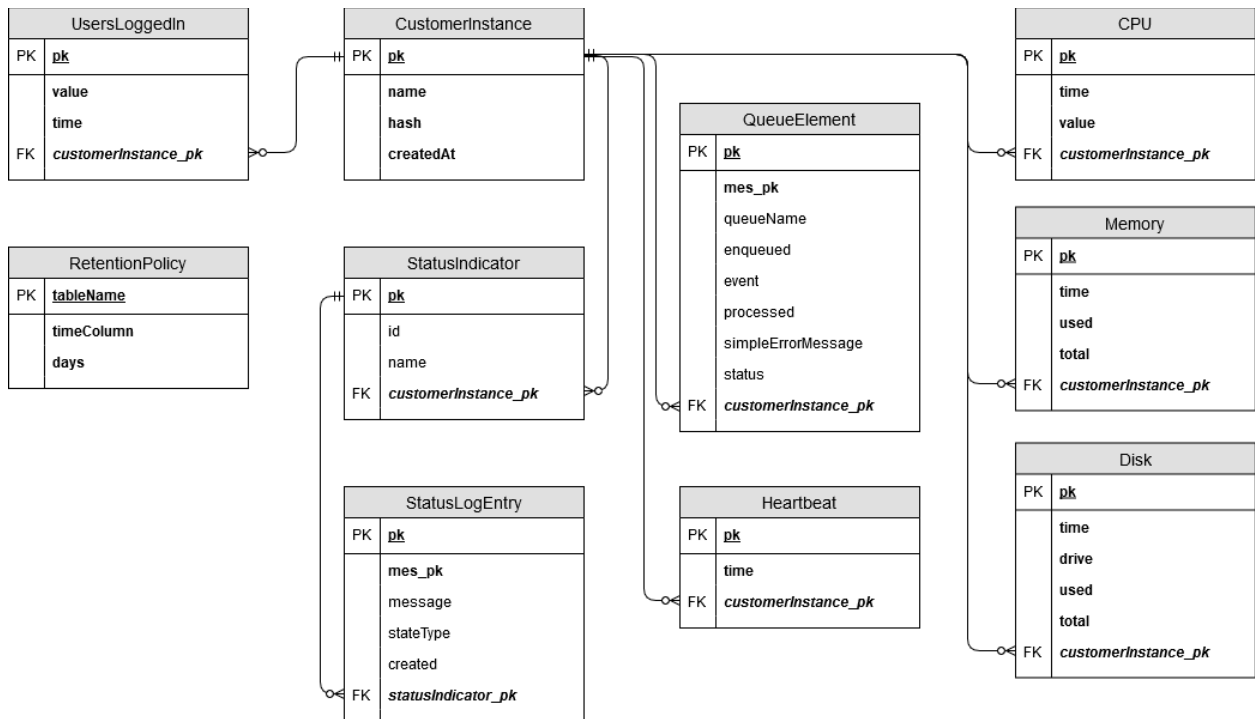
DAO'ene inneholder også forhåndsdefinerte SQL-spørringer, `MapSqlParameterSource`³⁰ for å legge inn verdier i spørringen og `NamedParameterJdbcTemplate` for å lage “prepared statements” med navngitte verdier. Vi bruker Spring sin `JdbcTemplate` til å lage “prepared statements” for å unngå potensielle SQL-injections. Ved å bruke “prepared statements” vil ingen brukere får lov til å skrive egne SQL-spørringer, men er isteden avhengige av å bruke ferdigskrevet SQL med sine egne verdier.

30

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/namedparameter/MapSqlParameterSource.html>

9.3 Database

9.3.1 Oppsett



Illustrasjonen over er et Entitet-Relasjons-diagram (ER-diagram) som viser hver tabell i databasen og deres relasjoner til hverandre. Hver boks er en tabell, og hver rad i hver boks er en kolonne i tabellen. Rader med “PK” foran seg, og som er understreket, er primærnøkler, mens rader med “FK” foran seg, og som er i *kursiv*, er fremmednøkler som peker til andre tabeller. Hver rad i **fet** er en “NOT NULL”-kolonne som ikke kan inneholde “NULL”-verdier.

Hver kunde hos Goodtech kan ha én eller flere instanser av MES kjørende, og som følger valgte vi å ha en tabell over kundeinstanser (CustomerInstance). Dette er en essensiell del av systemet da vi uten denne tabellen ikke vet hvilken kundeinstans, og dermed hvilken MES, som har sendt inn en gitt rad med loggdata.

Vi har en tabell for StatusLogEntry og en for QueueElement, på lik linje med MES-ens egen database hos hver kunde, men vi har kun valgt ut de kolonnene vi bryr oss om for våre visualiseringsformål. StatusIndicator er identisk til MES-ens egen StatusIndicator-tabell, men vi har lagt til en “customerInstance_pk”-kolonne da StatusIndicator-er kun er unike for hver kundeinstans, men kan forekomme flere ganger i tabellen vår. Dette gir også StatusLogEntry-tabellen en transitiv kobling til CustomerInstance-tabellen gjennom

StatusIndicator-tabellen. QueueElement-tabellen har derimot en direkte forbindelse med CustomerInstance-tabellen. Vi så ingen grunn til å endre på Goodtech sitt eksisterende skjema for disse tre tabellene, annet enn å sløyfe de kolonnene som ikke er nødvendige for oss, samt legge til kundeinstansen som en kolonne. Det gjorde det lett for oss å se hvordan vi leser inn hver rad med StatusLogEntry og QueueElement. Som følger ender videre tolking av dataene opp i visualiseringsdelen av systemet vårt.

CPU-, Memory- og Disk-tabellene inneholder de respektive målingene fra hver kundeinstans. I CPU-tabellen har vi "value"-kolonnen som inneholder den spesifikke CPU-prosentbruken i det øyeblikket målingen ble gjort. Memory- og Disk-tabellene har "used"-kolonnen som inneholder den nåværende bruken av minne og diskplass, i tillegg til "total"-kolonnen som inneholder den nåværende kapasiteten på minne og diskplass i det målingen ble gjort. Disse tre tabellene har en direkte kobling til CustomerInstance-tabellen. Originalt hadde vi disse tre tabellene sammen som én tabell, men vi kom fort på at det burde være mulig for en kundeinstans å ha mer enn én disk som skal loggføres. Vi delte dermed opp tabellen i tre tabeller; CPU, Memory og Disk; og la til "drive" i Disk-tabellen.

UsersLoggedIn-tabellen har målinger på hvor mange påloggede brukere det er på MES-en i det gitte øyeblikket, hvor "value"-kolonnen inneholder antallet brukere. Denne tabellen har også en direkte kobling til CustomerInstance-tabellen.

I tillegg til monitoreringsdataene vi mottar fra hver instans ute i MES-ene innså vi at det er fordelaktig å se om selve monitoreringsprogrammet vårt kjører på instansene i det hele tatt. I første omgang brukte vi CPU-målingene for å se om programmet kjørte, da vi hadde en fast 30-sekunders intervaller for CPU-målinger. Etterhvert ble det luftet et ønske fra Goodtech om at hver måling i programmet kunne skrues av og på individuelt, i tillegg til at intervallene for hver måling måtte være justerbar. Som følger kunne vi ikke garantere at CPU-målinger ble sendt inn, og vi fikk behov for en ny måling, "Heartbeat", som kun er en fast periodisk alltid-på "måling" som blir sendt inn periodisk. Ingen reelle data blir målt i denne målingen, da vi kun noterer at instansen var aktiv i det gitte øyeblikket målingen ble gjort. På denne måten kan vi til og med ha en instans som er konfigurert til å ikke sende inn noen målinger i det hele tatt, men allikevel kunne se at programmet kjører.

RetentionPolicy-tabellen er en tabell som inneholder konfigurasjon mer enn faktisk data. Hver rad beskriver en av tabellene nevnt over, hvilken kolonne i den gitte tabellen som representerer tid, og hvor mange dager rader i den tabellen skal beholdes før de regnes som foreldet. Retention policy regnes som aktivert på kun de tabellene som er ført opp i RetentionPolicy-tabellen. Hvis en ny tabell skal håndteres av retentionPolicy-scriptet (se seksjonen "[Retention Policy](#)") legges den bare til i tabellen med det antallet dager rader skal beholdes.

Tidspunkter lagret i databasen er antatt å være i Coordinated Universal Time (UTC)-tidssonen, en tidssone som er basert på solens posisjon ved 0 lengdegrader, men som også er ekvivalent

med GMT+1 uten sommertid. SQL sin DATETIME-type har ingen form for tidssone bygget inn, så det er ikke direkte mulig å hente ut tidssoneinformasjon fra hvert tidspunkt definert i databasen, men vi har tatt beslutningen om at alle tidspunkter i databasen skal og vil bli behandlet som om de er i UTC.

I tillegg til tabellene over har vi to “views” i databasen. Et “view” er en tabell som er konstruert med en forhåndsdefinert SELECT-spørring. Vi har viewsene “CustomerInstance12w” og “StatusIndicator12w” som forenkler spørringer i visualiseringen hvor vi ønsker å se på de siste 12 ukene med data for StatusLogEntry- og QueueElement-tabellene³¹.

9.3.2 Tabellforklaring

I kolonner hvor vi ønsker å lagre tekst bruker vi datatypen “NVARCHAR” i stedet for “VARCHAR” siden “NVARCHAR” støtter full Unicode, mens “VARCHAR” kun støtter 8-bits ASCII-tegn. 8-bits ASCII støtter de fleste tegn vi kommer til å bruke, men å begynne tidlig med full støtte for 8-bits ASCII i tillegg til hele unicode-settet var fordelaktig å gjøre mens vi hadde sjansen.

CustomerInstance-tabellen har en kolonne “hash”, som er tiltenkt for lagring av kryptografisk hashede passord for hver kundeinstans. Vi bruker hashingalgoritmen bcrypt, variasjon 2b, slik at hver passordhash ikke blir lengre enn 60 tegn. Vi forsøkte først å lagre dem i en BINARY-kolonne, men opplevde noen vansker med å konvertere de binære dataene til tekst i REST APIet, så vi endret kolonnetypen til NVARCHAR(60). Passordhashene kommer sannsynligvis aldri til å inneholde andre tegn enn de som er tillatt i 8-bits ASCII, men vi føler allikevel det er lurt å velge den typen som har lavest sjanse til å potensielt forvrengte data i dette tilfellet.

Tabellene “StatusLogEntry”, “StatusIndicator” og “QueueElement” er delvis basert på Goodtech sin MES-database slik den ser ut hos hver kunde:

- StatusIndicator representerer en spesifikk prosess som det kan forekomme en feil på. I “StatusIndicator” har vi “id” og “name”, slik de er oppført i den gitte MES-en hos en kunde.
 - “id” er et unikt identifiserende navn for en gitt StatusIndicator som er unik per kunde, da det ikke kan forekomme flere enn én StatusIndicator-rad med en gitt “id” i MES-en hos en kunde.
 - “name” er et lite brukt felt, ifølge Goodtech, men vi tok det med i databasen vår da Goodtech ytret et ønske om å ha den med til videre utvikling fra deres side. Slik vi har observert er “name” som oftest det samme som “id” for en gitt StatusIndicator.

³¹ Se [QueueElement og StatusLogEntry i Visualiseringsdelen](#) nedenfor.

- StatusLogEntry er en tabell som inneholder feilmeldinger fra MES-en som har forekommet på en spesifikk prosess, representert i databasen som en StatusIndicator. I StatusLogEntry har vi “message”, “stateType” og “created”, samt hovednøkkelen til den gitte StatusLogEntry-raden slik den forekommer i MES-en til en kunde på “mes_pk”.
 - “message” inneholder selve feilmeldingen til feilen som har forekommet.
 - “stateType” er alvorlighetsgraden av feilen; enten 1 for “warning” eller 2 for “critical”.
 - “created” er tidspunktet den gitte StatusLogEntry-raden ble opprettet, og dermed når feilen forekom.
- QueueElement er en blandet kø for hver av prosessene i MES-en. En gitt QueueElement-tabell kan inneholde kø-elementer fra flere forskjellige prosesser, som alle leser og skriver til QueueElement-tabellen for å kommunisere med hverandre, hvor de da setter informasjon som skal sendes rundt i kolonnen “xmlContent”, en kolonne vi har utelatt i representasjonen vår her da vi ikke har behov for innholdet til en gitt QueueElement-rad. I QueueElement har vi “queueName”, “enqueued”, “event”, “processed”, “simpleErrorMessage” og “status”, samt hovednøkkelen fra MES-en i “mes_pk”.
 - “queueName” er navnet på den spesifikke køen en rad tilhører, og lar en prosess vite hvilke kø-elementer som tilhører den.
 - “enqueued” er det tidspunktet det gitte kø-elementet ble lagt inn i QueueElement-tabellen, og dermed lagt inn i køen.
 - “event” er det tidspunktet MES-en mottok kø-elementet fra prosessen, hvorpå MES-en la kø-elementet inn i køen.
 - “processed” settes av den prosessen som har prosessert det gitte kø-elementet, og representerer det tidspunktet prosessen var ferdig med å prosessere kø-elementet og innholdet dens. Vil være “NULL” frem til den er prosessert.
 - “simpleErrorMessage” vil være “NULL” frem til det gitte kø-elementet blir prosessert med en feilmelding. Den originale tabellen i MES-en har både “errorMessage” og “simpleErrorMessage”, hvor “errorMessage” inneholder fulle stack-traces fra Java, mens “simpleErrorMessage” inneholder kun selve feilmeldingen.
 - “status” starter som “UNREAD”, som symboliserer at kø-elementet har blitt opprettet i køen, men ingen prosess har prosessert den ennå. Etter at kø-elementet har blitt prosessert blir “status” enten “OK”, “FAILED” eller “FAILED_BUT_SIGNED”, hvorpå “FAILED” og “FAILED_BUT_SIGNED” symboliserer at det er en feilmelding i “simpleErrorMessage”.

9.3.3 Retention Policy

Databasen vår vil potensielt etterhvert bli veldig stor. Siden hver kundeinstans sender data til databasen vil den inneholde veldig mye data som ikke lengre er relevant for Goodtech etter en viss tid. MSSQL har dessverre ingen innebygd funksjonalitet for å slette gammel data som ikke lengre er relevant. Slik vi så det, så har vi to valg:

- Et script utenfor databasen, men fortsatt på samme maskin, kan utføre en spørring som sletter gamle rader fra databasen periodisk.
- Vi kan legge inn en “trigger”, som er en form for spørring som utføres når spesifikke hendelser forekommer (som for eksempel innføring av ny data), som sletter gamle rader i tabellen det blir ført inn nye rader på.

Vi valgte det første alternativet da det andre alternativet virket betraktelig mer komplisert, og krever at vi manuelt setter opp en trigger på hver tabell som skal vedlikeholdes av en retention policy. I tillegg til hver eneste INSERT i hver av tabellene kjøre triggeren for den gitte tabellen, noe som er langt ifra nødvendig.

Vi endte opp med å håndheve en retention policy på 180 dager ved hjelp av et Javascript-script, kjørt gjennom NodeJS, som kjøres daglig på serveren som en cron-oppgave. Vi valgte NodeJS og Javascript her igjen da vi kunne gjenbruke mye kildekode fra Javascripten vi brukte for å bearbeide testdata tidligere i retention policy-scriptet. I tillegg er det lett nok å starte en NodeJS-applikasjon fra cron.

9.3.4 Normalisering

Alle tabellene våre tilfredsstill minimum BCNF, men noen av dem tilfredsstill høyere normaliseringsnivåer.

Tabellene CPU, Disk, Memory og UsersLoggedIn tilfredsstill 5NF da de ikke kan deles opp ytterligere uten å introdusere “surrogatnøkler”, som er identifiserende verdier som ikke har noe med de faktiske dataene i raden å gjøre, ofte i form av et løpenummer.

StatusLogEntry og QueueElement kan deles opp ytterligere for å normaliseres til 4NF og eventuelt 5NF, men vi valgte å ikke normalisere ytterligere enn BCNF da vi ønsket å holde dem så like MES-ens StatusLogEntry- og QueueElement-tabeller.

Heartbeat er i praksis i 6NF om den ikke hadde hatt en “pk”-kolonne, da den ville bestått av kun “time” og “customerInstance_pk”.

RetentionPolicy-tabellen kan heller ikke deles opp ytterligere, og er dermed i 5NF.

9.4 Visualisering

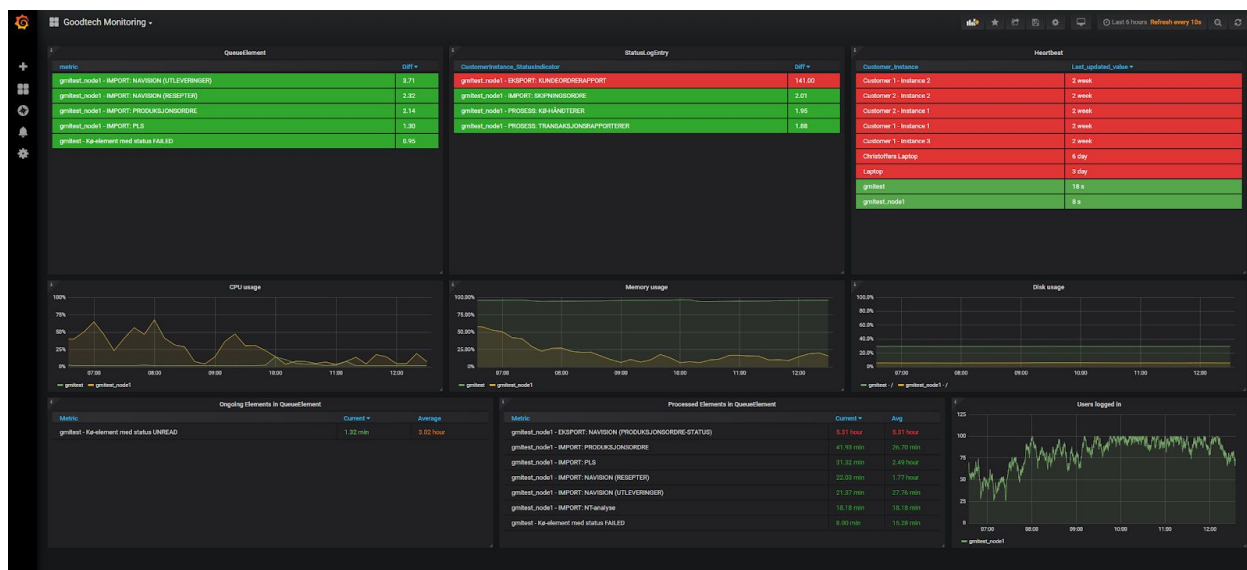
Vi endte opp med å dele visualiseringssiden opp i to deler: en infoskjerm og et detalj-panel. Infoskjermen er tenkt til å gi varsler om kritiske feil for alle kundeinstanser på en iøyenfallende og oversiktlig måte ved å bruke sterke, tydelige farger for å indikere at noe har gått galt. Infoskjermen er tiltenkt å være på for eksempel en TV lengre unna, gjerne hengende fra taket

eller hengende på en vegg. Om informasjonen på infoskjermen ikke er tilstrekkelig så kan en gå inn på detalj-panelet for å se dypere på hva som har gått galt for en gitt kundeinstans.

Grafana passet oss og vårt bruk fordi det hadde ferdigstilte moduler/grafar, innebygde funksjoner for å representere data i tidsintervaller og mulighet for å oppgi flere databaser. Det var også generelt enkelt å sette opp og krever minimalt med vedlikehold.

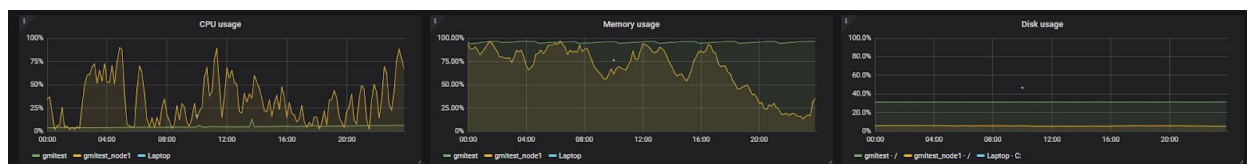
I og med at det ikke var noe kravspesifikasjoner utenom et generelt mål om å få visualisert tabeller i MES, så sto vi ganske fritt til hva vi skulle visualisere. Goodtech hadde selv ikke tenkt ut noen bestemte krav til dette og de ønsket å se hva vi kunne komme opp med. Dette kan være befriende, men også veldig skremmende i et bachelorprosjekt. Selv synes vi at vi taklet problemstillingen greit, og fikk visualisert det vi mente var viktigst, i samarbeid med Goodtech der de kom med noen ønsker om konkrete grafer.

9.4.1 Infoskjerm / Hoved-Dashboard



Hoved-dashbordet ble laget med enkelhet i tankene. Det skal være lett å se hvor det har gått galt og når det gikk galt. Bruksområdet til det er ment å være en tavle og det er ikke ment å bli manipulert. Derfor er mange av grafene statiske og ikke bruker tidsfilter funksjonen i Grafana.

9.4.1.1 CPU, disk og minne

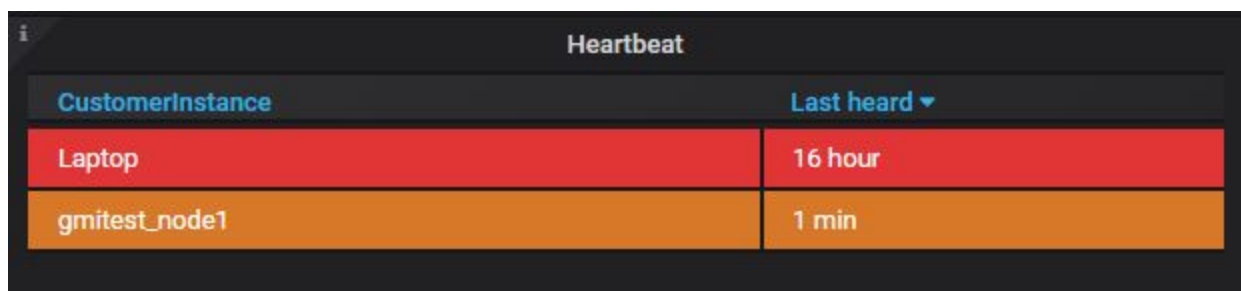


Disse tre modulene viser CPU-, minne- og diskbruk delt per kunde. CPU blir sendt til DB fra loggbehandleren i et prosenttall så det er ikke noe annet en avg() funksjon som blir brukt for å hente denne verdien med spørringen. I minne- og diskbruk loggføres totalt tilgjengelig plass i tillegg til totalt brukt plass. Dette blir da et enkelt regnestykke der vi deler hvor mye som er brukt på totalt tilgjengelig.

Alle tre er multi-serielle grafer der CPU og minne lager en egen linje for hver kundeinstans og disk lager en for hvert par med instanser og diskenheter, slik at om en instans har flere disker som monitoreres blir hver disk en linje på grafen.

Dataene for disse tre grafene presenteres som en prosent brukt av totalt tilgjengelige ressurser. Som følger spiller det ingen rolle hvor kraftig en maskin er eller hvor mye ledig plass den har i minne eller på disk, hvis den nærmer seg kritiske nivåer vil den vises som kritisk.

9.4.1.2 Heartbeat-modul



CustomerInstance	Last heard
Laptop	16 hour
gmitest_node1	1 min

Denne modulen ser på når siste rad i Heartbeat-tabellen ble lagt inn per kundeinstans. For å oppnå denne spørringen henter vi ut maks av time kolonnen og setter denne verdien inn i en "DATEDIFF"-funksjon³². DATEDIFF tar inn tre parametere, tidsenhet (sekunder, minutter, timer osv.), starttiden som skal bli kalkulert og sluttidspunkt som skal bli kalkulert. Dette vil da si at vi får ut tiden fra da den siste verdien ble lagt til i databasen til "nå". Får å ikke få mange unødvendige desimaltall i denne spørringen så runder vi ned med "FLOOR()" funksjonen³³. Vi illustrerer når vi hørte sist fra en kundeinstans i form av tabellen over. Kundeinstansens navn er på høyre side og hvor lenge siden vi sist hørte fra kundeinstansen er på høyre side. Tidspunktet er konvertert fra rene sekunder til en mer menneskelesbar skala som endrer seg til minutter, og etterhvert timer, etterhvert som det blir lengre tid siden vi sist så en Heartbeat-rad fra kundeinstansen.

9.4.1.3 QueueElement og StatusLogEntry

Disse tabellene er to av de tabellene som holder styr på feilmeldinger innad kundens system. Disse gir en indikasjon på hvilke feilmeldinger som er kritiske og trenger tilsyn. Spørringene er

³² <https://docs.microsoft.com/en-us/sql/t-sql/functions/datediff-transact-sql?view=sql-server-2017>

³³ <https://docs.microsoft.com/en-us/sql/t-sql/functions/floor-transact-sql?view=sql-server-2017>

nesten helt like med unntak av noen forskjeller med hvordan StatusLogEntry henter ut hvilken kundeinstans dataene hører til da hver StatusLogEntry tilhører en StatusIndicator, og hver StatusIndicator tilhører en CustomerInstance. Dette er også en av de mest avanserte spørringene som blir brukt.

Disse tabellene ble utviklet for å utjevne dataen og gi en indikator på hvor noe har gått galt. Spørringene tar feilmeldinger de siste "X" timene og deler dette på gjennomsnittet av feilmeldinger de siste 12 ukene på gitt queueName/statusIndicator i gitt kundeinstans. På denne måten vil de feilmeldingene som blir sendt 20 ganger i timen bli vektlagt mindre enn en annen feilmelding som oppstår en gang i uken. Tallet som ligger i kolonnen "Diff" sier hvor mange ganger en viss feilmelding har feilet de siste X timene i forhold til gjennomsnittet. Om et felt har "Diff" på 3 vil det si at den har feilet 3 ganger så mye de siste X timene fra hva som er normalt for denne feilmeldingen.

QueueElement		StatusLogEntry	
CustomerInstance - QueueName	Diff	CustomerInstance - StatusIndicator	Diff
gmitest - Kø-element med status FAILED	1.07	gmitest_node1 - PROSESS: TRANSAKSJONSRAPPORTERER	0.03
gmitest_node1 - IMPORT: PLS	0.01	gmitest_node1 - IMPORT: SKIPNINGSORDRE	0.01
gmitest_node1 - IMPORT: NAVISION (RESEPTER)	0.01	gmitest_node1 - PROSESS: KØ-HÅNTERER	0.01

For å oppnå denne spørringen så bruker vi to delspørringer hvor en av disse har enda en delspørring.

Vi tar for oss spørringen for QueueElement først. Her bruker vi to av funksjonene som Grafana tilbyr kalt "\$__timeGroup()" og "\$__timeFilter()". Den første delspørringen henter ut antall gjennomsnittlige feil 12 uker tilbake, eller fra da den bestemte kundeinstansen ble opprettet om det er under 12 uker siden. Denne grensen er brukt får å få et greit gjennomsnitt av hver feilmelding, der det ikke tar for lang tid å kjøre spørringen i dette tidsintervallet. Grunnen til at den starter med intervallet fra da kundeinstansens opprettes er for å ikke få for store perioder med "ingen feil" mens kundeinstansen ikke var aktiv i tidsrommet som blir brukt da dette vil gi et feil utslag i tabellen. Dette er da en dato kalt "timeSpan" hentet ut av et view i databasen kalt "CustomerInstance12w". Videre henter denne delspørringen ut kundeinstansens primærnøkkel ("pk"), navn og "queueName", en indikator for hvor feilmeldingen kom fra. Gjennomsnittet blir oppnådd ved å ta å telle antall feilmeldinger i denne tabellen via "COUNT()"³⁴ funksjonen og dele dette på en "DATEDIFF()" funksjon satt til timer, startdato (timeSpan) og "GETUTCDATE()"³⁵, som gir oss nåværende tidspunktet. DATEDIFF returnerer da antall timer siden hver kundeinstanse ble opprettet eller 12 uker tilbake. I "WHERE"-klausulen av spørringen spesifiseres det at det er kun rader som inneholder status "FAILED" for å kun få feilmeldinger. Denne variabelen er gitt navnet "AVERAGE".

³⁴ <https://docs.microsoft.com/en-us/sql/t-sql/functions/count-transact-sql?view=sql-server-2017>

³⁵ <https://docs.microsoft.com/en-us/sql/t-sql/functions/getutcdate-transact-sql?view=sql-server-2017>

I den andre delspørringen blir det hentet ut antall feil de siste X timene. Vi henter ut kundeinstansens pk og queueName likt som i første delspørring og igjen telle antall linjer i denne del-tabellen med COUNT() for å få antall feil de X timene. I "WHERE"-klausulen av spørringen blir det spesifisert igjen at det er kun feilmeldinger som skal vises ved å filtrere på at "status" skal være "FAILED". Deretter innsnevres tidsintervallet som blir sett på, da X timer, med en "DATEADD"³⁶ hvor vi sier at "event" skal være større enn "DATEADD" med parameterne time, -X og "GETUTCDATE()". Dette vil da bare gi feilmeldingene den siste timen, denne variabelen er gitt navnet "ErrorsLastHour".

De to delspørringene blir samlet på kundeinstans pk og queueName i den ytre spørringen for å sørge for at begge tabellene linker sammen kundeinstansene og queueNamene. "ErrorsLastHour" blir delt på "AVERAGE" for å gi et tall som skal være en indikator på hvor mange flere feil det har vært de siste X timene i forhold til hva som er den normale gjennomsnittlig antall feil per X timer. I denne uthentingene blir det også satt en "CASE" for å unngå tilfeller der gjennomsnittet kan være 0. Det blir også kalt på kundeinstans pk som blir gjort om til kundeinstans navn og queueName. Dette blir samlet i en kolonne som heter "metric".

Som nevnt tidligere så er disse spørringene til dels like for QueueElement og StatusLogEntry. Forskjellen mellom dem er at StatusLogEntry henter ut kundeinstans pk gjennom StatusIndicator-tabellen, samt at den benytter "StatusIndicator12w" i stedet for å se på starttiden for sammenligningen. I stedet for queueName for å skille mellom køer brukes StatusIndicator-tabellens "id".

9.4.1.3.1 Modulær spørring

Disse spørringene bruker ingen av Grafanas macro-funksjoner for tidsfiltrering og gruppering, og vil være statisk til en hver tid uavhengig av Grafanas globale tidsfilter. Den blir ikke endret ved å endre tidsintervallet i Grafana, men den har samme mulighet for å bli endret hvis man går inn i spørringen. Dette ble gjort ved å ikke ha noen hardkodet from for tidsrom, selv om i praksis så er det hardkodet. Ved å gange "COUNT()" feltet i den første delspørringen (AVERAGE) på fks 12 så vil denne delspørringen dele gjennomsnittet opp i 12 timers bolker. Man må også endre et tall i "DATEADD()" funksjonen i den andre delspørringen (ErrorLastHour), spesifikt det tallet som sier hvor langt tilbake i tid denne tabellen skal se på. Dette er gjort for å enkelt kunne endre på tabellen om Goodtech har nytte for det.

Spørringen var originalt tenkt til å se på den siste timen med feilmeldinger. Dette viste seg å være problematisk da det er visse veldig kritiske feil som forekommer veldig sjeldent, og vil da bare være synlige i 1 time før de ville forsvunnet fra tavlen igjen. Et annet problem er at om man øker antall timer som blir sett tilbake på så får man med flere feilmeldinger som feiler en gang per dag som vil gi et feil bilde av hva som er kritisk. Dette er noe av grunnen til at spørringen

³⁶ <https://docs.microsoft.com/en-us/sql/t-sql/functions/dateadd-transact-sql?view=sql-server-2017>

endte med å bli modulær der det kan bli tilpasset etter levert oppgave hvor en har virkelig data å jobbe med.

9.4.1.4 QueueElement-spesifikke tabeller

En unik ting i dette prosjektet er at tabellen QueueElement er en kø av elementer som skal bli utført og håndtert. Data som kan være interessant å se på er hvor lenge et element har stått i køen og hvor lang tid det tok for hvert enkelt element å bli ferdig. Vi delte dette opp i to forskjellige moduler på hovedpanelet hvor den ene viser hvor lenge et element har stått i kø og den andre viser hvor lang tid hvert kø-element tok for å bli ferdig.

Noen måter vi fant ut å differensiere om et kø-element er blitt ferdig var om det har en feilmelding i “simpleErrorMessage”-kolonnen, at kolonnen “processed” ikke er “NULL”, eller om det har en status “unread”. Her igjen bygger begge modulene på samme prinsipp med noen forskjeller.

Ongoing Elements in QueueElement		
CustomerInstance - QueueName	Current	Average
Laptop - IMPORT: PRODUKSJONSORDRE	6.98 day	6.98 day
gmittest - Kø-element med status UNREAD	6.40 day	6.91 day

Processed Elements in QueueElement			
CustomerInstance - QueueName	Current	Avg	
gmittest_node1 - IMPORT: PRODUKSJONSORDRE	4.77 min	5.78 hour	
gmittest_node1 - IMPORT: NT-analyse	33.38 min	2.35 hour	
gmittest_node1 - EKSPORT: NAVISION (PRODUKSJONSORDRE-STATUS)	2.50 min	2.32 hour	
gmittest_node1 - IMPORT: NAVISION (UTLEVERINGER)	7.75 hour	2.19 hour	
gmittest_node1 - IMPORT: NAVISION (RESEPTER)	1.25 min	2.17 hour	
gmittest_node1 - IMPORT: PLS	6.48 min	1.93 hour	
gmittest_node1 - IMPORT: NAVISION (PRODUKSJONSORDRE)	12.65 min	50.46 min	
gmittest - Kø-element med status FAILED	27.00 min	15.35 min	

9.4.1.4.1 Processed elements in QueueElement

Spørringene starter med å definere feltene den skal inneholde. Vi bruker timeGroup i denne setningen hvor den stykker opp tid i hvert trettiende sekund. Kundeinstansen og queueName blir også hentet ut. Til slutt bruker vi en “DATEDIFF”-funksjon med parametrene, sekunder, enqueued-kolonnen (starttid), og processed-kolonnen (sluttid). “DATEDIFF” brukes her på samme måte som i Heartbeat modulen. Vi filtrerer med “timeFilter”-makroen til Grafana i WHERE-klausulen. Vi spesifiserer også at processed ikke skal være NULL, da dette indikerer at kø-elementet ikke er ferdig prosessert, og at statusen skal være “FAILED”. Da kø-elementer som tok 0 tid for å fullføres, da de ofte blir plukket opp og prosessert umiddelbart, inkluderer vi en HAVING-klausul som sier at DATEDIFF feltet ikke skal være 0 eller mindre.

9.4.1.4.2 Ongoing elements in QueueElement

Forskjellene fra denne spørringen og “Processed elements in QueueElement” er at sluttidspunktet er satt til “GETUTCDATE()” for å få tiden “nå” da elementene fortsatt er i køen. Videre defineres det at ingen rader skal ha en satt processed-kolonne, som betyr at vi henter ut rader som ikke har blitt prosessert ennå. I modulen så blir det satt at current og average

kolonnene i denne tabellen skal være representert i tid og her blir tallet grønt, oransje eller rødt for å indikere at det har stått lenge uten å bli fullført.

9.4.1.5 Users logged in

Vi grafer hvor mange brukere som er logget inn i systemet over tid. Ved hjelp av den MSSQL-spesifikke aggregeringsfunksjonen “LAST_VALUE”³⁷ henter vi ut siste rad for en gitt kundeinstans i UsersLoggedIn-tabellen. LAST_VALUE ser mer ut som en fullverdig spørring i seg selv enn en funksjon. Den krever at en spesifiserer hva som skal hentes ut, hvordan dataene den henter ut fra skal sorteres, samt hvilken tabell den skal hente ut fra. Alternativt kan også antallet rader som skal tas med spesifiseres.

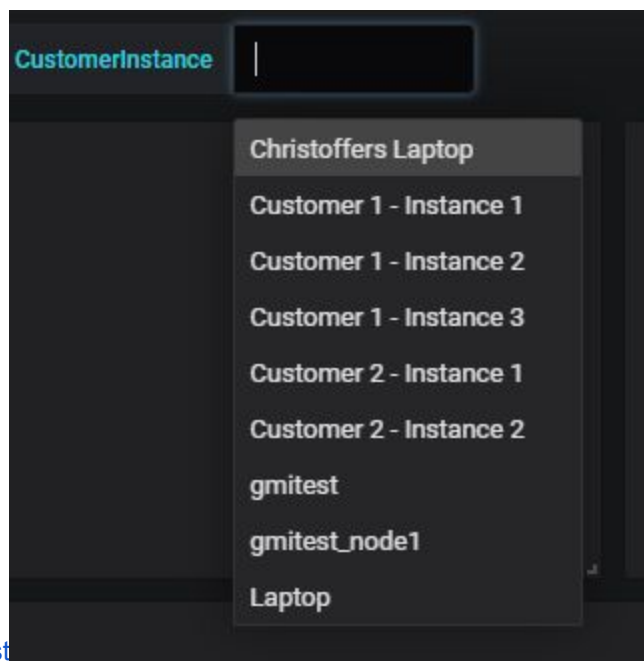
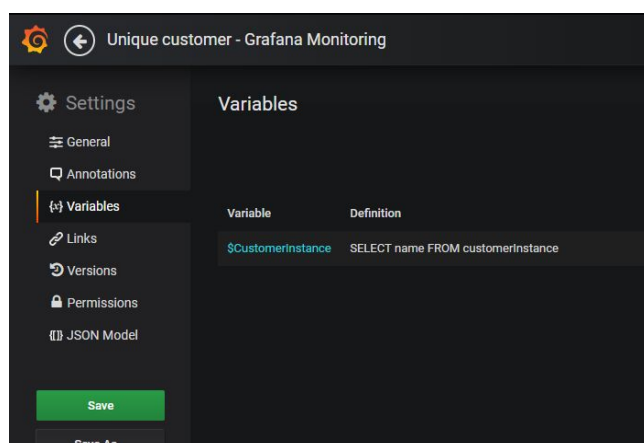
9.4.2 Unik kunde dashboard

Dette panelet inneholder mange av de samme modulene som infoskjermen, men ment for å vise informasjonen mer detaljert og på kundeinstans-basis.

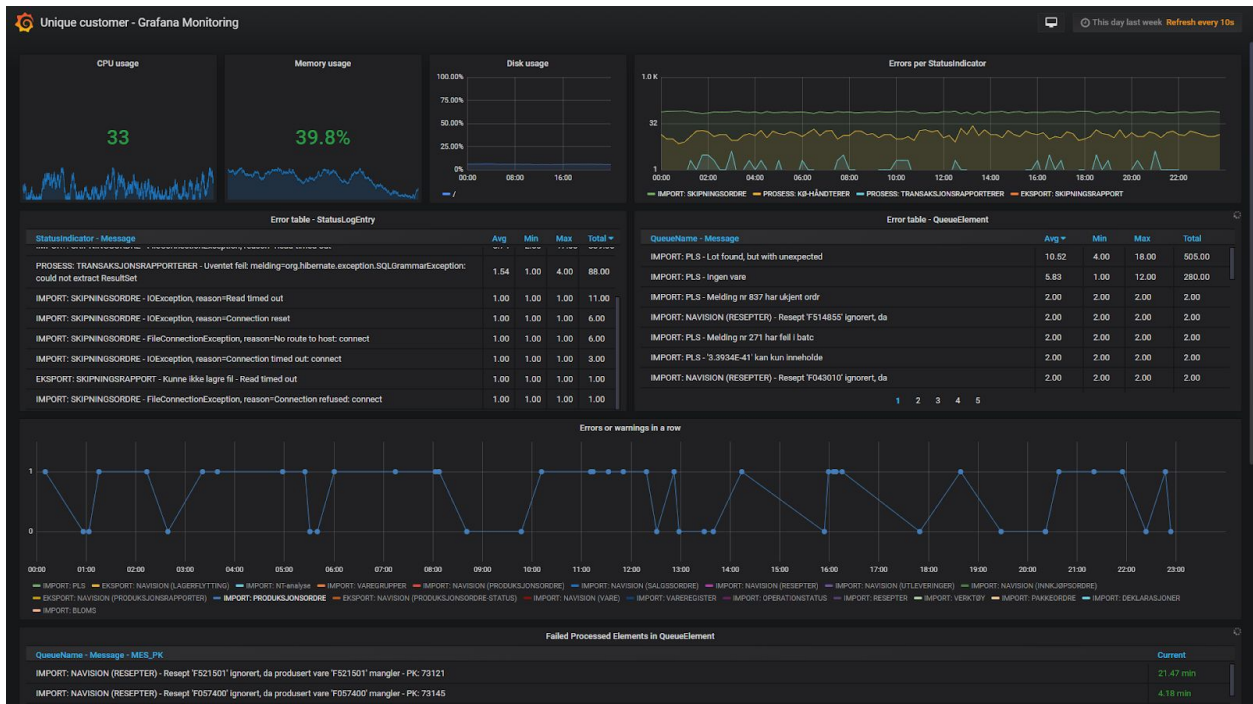
I dette dashboardet bruker vi en annen funksjonalitet Grafana tilbyr: egendefinerte variabler i panelet. Dette kan brukes for å sette verdier i et felt som man vil bytte dynamisk fra utenfor spørringene. Denne verdien kan så refereres til i spørringene til hver modul.

I vårt tilfelle bruker vi dette for å kunne velge hvilken kundeinstans vi ser på. Dette blir brukt i alle spørringene i dette panelet. I stedet for å velge alle kundeinstanser i SELECT delen, og spesifisere dem som en del av “metric”-kolonnen i resultatet av spørringen, inneholder spørringene en WHERE-klausul der kundeinstansen skal være lik “\$CustomerInstance”-variabelen for panelet. Det er da dette som differensierer de spørringene som er like fra hoveddashboardet til dette dashboardet, samt at de inkluderer feilmeldingsteksten som er generert.

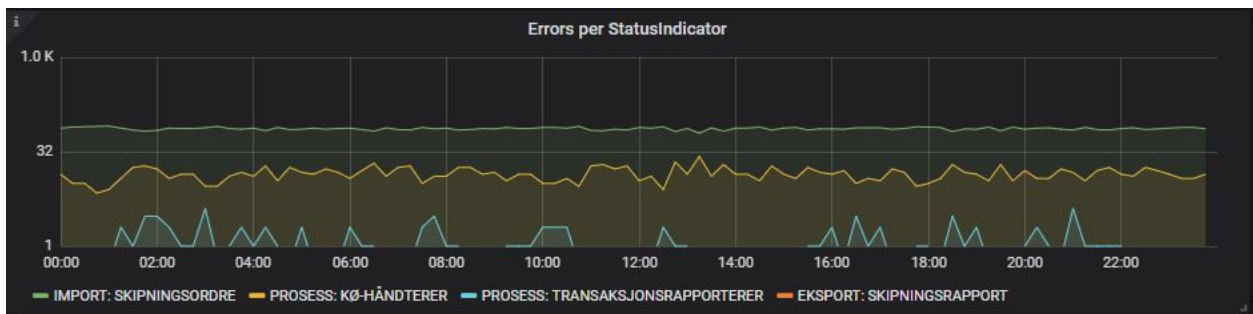
De spørringene som er unike for dette dashboardet skal vi se nærmere på her.



³⁷ <https://docs.microsoft.com/en-us/sql/t-sql/functions/last>



9.4.2.1 Errors per statusindicator



Denne modulen viser antall feilmeldinger i StatusLogEntry-tabellen skilt på StatusIndicator-id for hver StatusLogEntry tilhørende statusindikator. Spørringen starter med å definere timeGroup-makroen til Grafana, videre henter den ut hver StatusIndicator og teller over radene i denne tabellen med funksjonen COUNT(). I WHERE-klausulen spesifiserer vi at den skal ha filtrene på Grafanas globale tidsfilter, samt at kundeinstansen skal være lik \$CustomerInstance-variabelen.

Denne grafen har en logaritmisk Y-akse. En logaritmisk skala tydeliggjør små endringer i forhold til store endringer, slik at ikke enten små endringer blir for flate til å synes i grafen eller for store så de er utenfor grafen.

9.4.2.2 Error Table - StatusLogEntry / QueueElement

StatusIndicator - Message	Avg	Min	Max	Total
IMPORT: SKIPNINGSORDRE - Timeout	37.97	27.00	46.00	3.65 K
IMPORT: SKIPNINGSORDRE - FileConnectionException, reason=Connection timed out: connect	27.71	18.00	38.00	2.66 K
PROSESS: KØ-HÅNDBTERER - Forventet tidsforbruk overskredet	14.27	7.00	27.00	1.37 K
IMPORT: SKIPNINGSORDRE - FileConnectionException, reason=Read timed out	8.74	2.00	17.00	839.00
PROSESS: TRANSAKSJONSRAPPORTERER - Uventet feil: melding=org.hibernate.exception.SQLGrammarException: could not extract ResultSet	1.54	1.00	4.00	88.00
IMPORT: SKIPNINGSORDRE - IOException, reason=Read timed out	1.00	1.00	1.00	11.00
IMPORT: SKIPNINGSORDRE - IOException, reason=Connection reset	1.00	1.00	1.00	6.00
IMPORT: SKIPNINGSORDRE - FileConnectionException, reason=No route to host: connect	1.00	1.00	1.00	6.00

Denne modulen viser gjennomsnittlige, min, maks og totale antall feilmeldinger over det tidsintervallet tidsfilteret ser på. Dette blir oppnådd ved å sette timeGroup for å stykke opp tabellen i bolker. Videre henter vi ut statusindikatoren "id"-kolonne og "message"-kolonnen fra StatusLogEntry som blir "metric" til tabellen vist over. Til slutt COUNT()-er vi over pk i StatusLogEntry for å få antall feilmeldinger. I WHERE-klausulen blir tidsfiltre satt, det blir videre presisert at stateType ikke skal være lik 0 for å filtrere ut meldinger som ikke er feilmeldinger. Vi filtrerer også på at message-kolonnen ikke skal være "NULL" da dette gjør at tabellen ikke kan vises. Det blir også satt filter for hvilken kundeinstans som blir brukt og kobling mellom StatusIndicator-tabellen og CustomerInstance-tabellen.

Dette blir da satt inn i en tabell for tidsserie aggregasjon i Grafana. Dette gjør at man kan velge hvordan dataene skal representeres. Som sagt tidligere velger vi å sette inn kolonner for gjennomsnittet, min, maks og totale verdier for disse.

9.4.2.3 Errors or warnings in a row



Dette er en graf som indikerer om et gitt queueName har feilet med et bestemt mønster. Spørringen starter med å bli delt opp i gitte tidsintervaller via Grafanas TimeGroup funksjon. Videre hentes det ut queueName og kundeinstans. Det blir satt et CASE hvor det sette at om statusen er Ok så skal den settes til 0, ellers skal den være 1. Dette blir satt inn i en modul som har linjer og prikker.



Dette er den siste spørringen som ble jobbet med og idéen bak modulen/spørringen var at etter hver feilmelding så skulle grafen gått opp med en, og bli satt til 0 når neste ok melding ble sendt. Dette ville gitt et bedre visuelt bilde og en bedre indikasjon om noe feilet ofte og i et bestemt mønster, dette kan peke ut planlagte operasjoner som regelmessig feiler. Grafen vil da fått et trappe-lignende utseende.

10 Sikkerhet og risiko

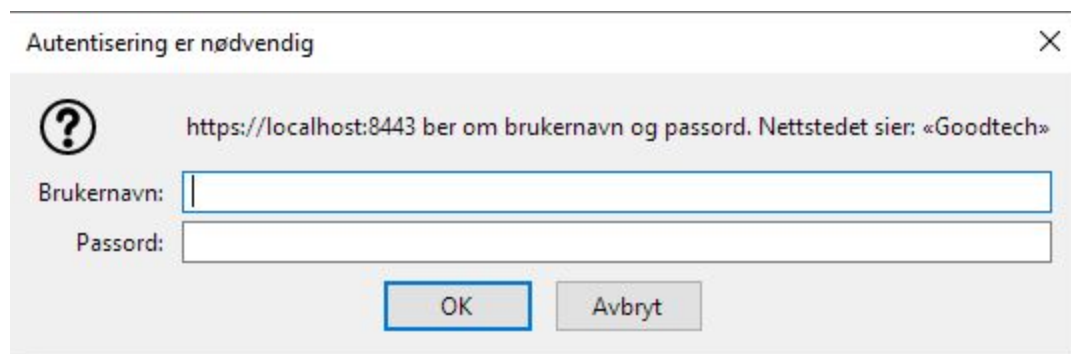
10.1 Backend

Vi har implementert sikkerhet til mes-monitoring-prosjektene i form av bruker-autentisering og kryptering av nettverkstrafikk, i tillegg til å håndtere databasekall internt på serveren med “prepared statements”.

REST APIet tilrettelegger for at all loggdata kan bli sendt gjennom dette APIet, istedenfor at SQL-spøringer skal sendes direkte mot databasen. Dette betyr at databasen ikke trenger å ha åpne porter ut mot nettverket, som igjen vil hindre potensielle angrep mot databasen.

REST APIet tillater kun HTTP-POST-forespørsler. Dette er et designvalg begrunnet med at APIet kun skal ta imot logger og ikke vise eller returnere noe innhold utenom statuskoder og eventuelle feilmeldinger. Det har derfor ikke vært nødvendig å tilrettelegge for menneskelig interaksjon med serveren på dette stadiet.

Autentisering av brukere gjør at kun eksisterende brukere med gyldig brukernavn og passord får lov til å sende data til databasen. Brukerne, med brukernavn og hashede passord, lastes inn fra databasen og legges til gjennom en `InMemoryUserDetailsManagerConfigurer`³⁸. Dvs. brukerne lastes inn ved oppstart av serveren, og legges til i en intern bruker-database. Alle klientene som sender data til serveren legger ved brukernavn og passord i HTTP-headeren, som brukes til å validere en “innlogging” til serveren. Disse brukernavnene og passordene blir sjekket opp mot den interne bruker-databasen og innloggingen blir godkjent eller avslått avhengig om brukernavn-og-passord-kombinasjonen er gyldig eller ikke.



Autentiserings-dialogvindu som blir presentert når man prøver å koble til mes-monitoring-server-applikasjonen gjennom en nettleser.

38

<https://docs.spring.io/spring-security/site/docs/4.2.5.RELEASE/apidocs/org/springframework/security/config/annotation/authentication/configurers/provisioning/InMemoryUserDetailsManagerConfigurer.html>

HTTP Status 401 - Full authentication is required to access this resource

Feilmeldingen over er den man får når man oppgir en ugyldig kombinasjon av brukernavn og passord.

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed May 15 12:25:05 UTC 2019

There was an unexpected error (type=Not Found, status=404).

No message available

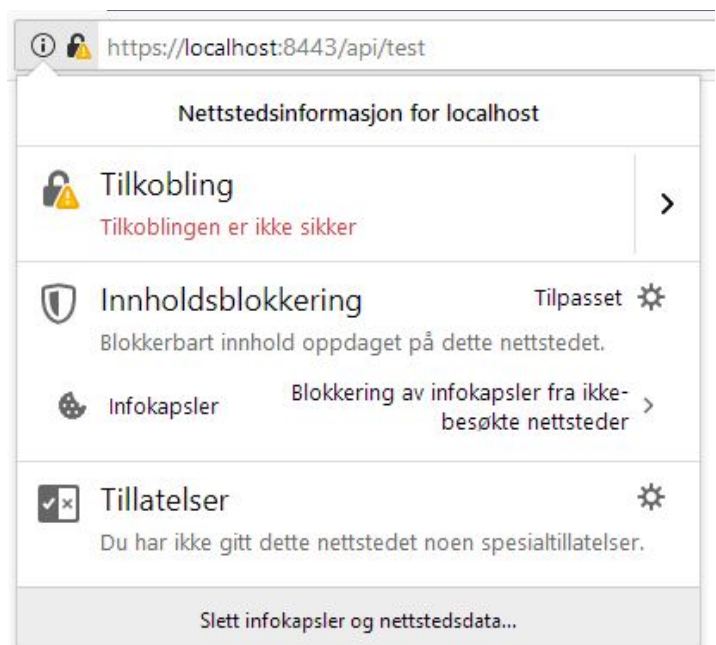
Standard feilmelding fra Spring. Denne får man når man logger inn med korrekt brukernavn og passord, siden REST APIet ikke støtter GET-forespørsler (som nettlesere bruker)

Kryptering av nettverkstrafikk gjør at det er vanskeligere for uvedkommende å få tak i informasjonen i meldingene som sendes gjennom nettet. Det vil fortsatt være mulig å snappe opp, eller "sniffe", nettverkstrafikken, men når denne informasjonen er kryptert, vil den være uleselig for mennesker.

Krypteringen vi bruker for å sikre nettverkstrafikken mellom klienten og serveren er en RSA-2048 privat og offentlig nøkkelpar som er brukt sammen med en SHA-256 hash-funksjon for å lage et public key certificate.

Serveren har den private nøkkelen, og holder denne nøkkelen hemmelig, mens den gir ut den offentlige nøkkelen til alle som prøver å koble seg til serveren, via det genererte sertifikatet.

Dette gjør at en klient kan spørre serveren etter en offentlig nøkkel, og bruke denne nøkkelen til å kryptere meldingen som skal sendes til serveren. Når serveren mottar den krypterte meldingen, bruker den sin private nøkkel til å dekryptere innholdet i meldingen. Deretter kan serverens interne regler avgjøre hva som skal skje med meldingen. Eksempelvis kan den godta innholdet og sende det videre for behandling, eller så kan den avvise innholdet og gi en passende feilmelding i retur.



Illustrasjonen til høyre viser hvordan en kryptert kanal fortsatt vises som usikker, siden sertifikatet ikke er utstedt av CA.

Ettersom vi jobbet med en løsning som kun skulle være tilgjengelig internt på Goodtech sitt nettverk, ble vi enige om å generere et “self-signed certificate” fremfor å få et sertifikat utstedt av en tredjepart, Certificate Authority(CA). Dette begrunnet vi med at det ville være ugunstig å få utstedt et sertifikat uten et eget tilhørende domene (eksempel: example.com).

Et problem med “self-signed certificate” er at man ikke beskyttes mot et “man-in-the-middle”-angrep uten en tredjepart til å godkjenne sertifikatet. Etter en samtale med veileder hos Goodtech, fant vi ut at vi ikke trengte å tenke på dette i denne omgang, siden all trafikk skal gå gjennom det interne nettverket, og at et “man-in-the-middle”-angrep på deres interne nett ville vært så kritisk at dataene vi bruker i vårt prosjekt er en bagatell i denne sammenhengen.

Vi måtte implementere en egen konfigurasjonsklasse for å generere en RestTemplate³⁹ som kunne håndtere trafikk gjennom HTTPS i mes-monitoring-client-prosjektet. Her konfigureres det blant annet at man skal akseptere “self-signed certificate”, for å kunne bruke vårt sertifikat til å kryptere meldingene. Denne RestTemplate’n brukes så av klienten hver gang den skal sende data til serveren.

39

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html>

Bildet under er en skjermdump av nettleseren Mozilla Firefox sin presentasjon av sertifikatet vi laget for å kryptere nettverkstrafikken til mes-monitoring-server.

Klarte ikke bekrefte sertifikatet fordi utstederen er ukjent.

Utstedt til

Vanlig navn (CN)	Tommy Abelsen
Organisasjon (O)	Goodtech
Organisasjonsenhet (OU)	Industrial IT
Serienummer	51:5E:F2:C6

Utstedt av

Vanlig navn (CN)	Tommy Abelsen
Organisasjon (O)	Goodtech
Organisasjonsenhet (OU)	Industrial IT

Gyldighetsperiode

Starter den	mandag 29. april 2019
Utløper	torsdag 26. april 2029

Fingeravtrykk

SHA-256 fingeravtrykk	52:D2:81:4F:62:6F:7E:96:2D:98:86:A0:1F:3F:97:69: 4C:D4:39:A5:2C:DA:5A:F4:A4:6B:88:2E:7D:F9:30:E0
SHA1 fingeravtrykk	AD:FD:EA:0B:47:B8:A1:63:4E:50:08:4E:5F:B9:A8:56:19:B5:0F:7C

10.2 Database

10.2.1 Intranett

Databasen ligger på Goodtech sitt intranett, og er ikke tilgjengelig for utenforstående. Vi hadde behov for å bruke verktøy som SSMS, slik at vi ikke kunne begrense databasen til å kun være tilgjengelig fra maskinen den, REST APIet og Grafana-serveren sitter på. Dette ville ytterligere sikret databasen fra uautoriserte inntrengere.

10.2.2 Brukere og tilgangskontroll

Grafana har sin helt egne bruker den benytter for å hente ut data fra databasen. Denne brukeren, som heter "grafana", har kun "SELECT"-tillatelse i databasen. På den måten kan den ikke føre inn nye data, oppdatere eksisterende data eller slette data fra databasen. Grafana har ikke behov for noe mer enn å hente ut data, så disse begrensningene på denne brukeren er ikke til noen hindring for Grafana i seg selv, men minker potensielle sikkerhetsrisikoer om brukerens innloggingsdetaljer skulle bli lekket. Grafana beskriver også viktigheten av å begrense database-brukerens rettigheter da Grafana ikke validerer spørringene som blir brukt i dashbordene til Grafana. (Grafana, u.å., Using Microsoft SQL Server in Grafana)

10.2.3 Backup

Vi satte ikke opp noen form for backup av databasen. Vi gjorde den store tabben og tenkte ikke på backup på noen måte, og dermed satte vi ikke opp noen form for backup-prosedyrer i det hele tatt. Optimalt sett ville vi satt opp periodisk backup som blir utført daglig, hvor opptil en uke med backuper blir lagret for eventuelle hendelser som krever gjenoppretting fra backup. MSSQL har innebygd funksjonalitet for opprettelse av backup av databasen, og med SSMS er det veldig lett å sette opp backup av databasen for en MSSQL-database.

10.3 Visualisering

Når det gjelder sikkerhet i produktet så skjer de fleste tiltakene i backend og database. Grafana har en egen tilgangskontroll med brukerkontoer som kan ha forskjellige rettigheter. Vi har to forskjellige brukere i Grafana: en admin-bruker som har alle rettigheter, og en display-bruker som bare kan se på dashbordene. Det er også en annen rolle som heter "editor" som lar deg kun manipulere og se på spørringer uten å opprette nye dashbord, men vi har ikke implementert en bruker med denne rollen.

Den begrensede display-brukeren lar oss vise dashbordene på flere maskiner uten å kompromittere resten av databasen eller visualiseringsløsningen. Admin- og editorrollene er reservert til de som utvikler moduler til visualiseringsløsningen. Dette vil øke sikkerheten mot angrep så godt det kan gjøres i denne løsningen.

Grafana har sin egen innebygde versjonskontroll. Dette er et indirekte sikkerhetsaspekt i det tilfellet der en person får tilgang til admin brukeren og ødelegger det som er laget der. Ved hjelp av versjonskontrollsystemet til Grafana kan dermed tidligere arbeid gjenoprettes.

I dette prosjektet så hadde vi ikke tenkt så mye på back up av Grafana-instansen. Grafana har mulighet for å eksportere alle dashbord og moduler i JSON format. Det ble lagret etter store endringer på dashbordene, men til daglig ble det ikke tatt noen back up, og vi har ikke satt opp noen automatiske rutiner for backup av konfigurasjonen.

11 Videre utvikling

I dette kapitlet tar vi for oss flere måter å videreutvikle dette prosjektet. Oppgaven er bygget opp til å bli videreutviklet der Goodtech har planer om å bruke dette i drift.

11.1 Backend

Det kan være hensiktsmessig å implementere et administrator-grensesnitt på REST API-serveren. Med et slikt grensesnitt kan man få mulighet til å legge til og endre brukere, med brukernavn og passord, og interne roller og rettigheter.

Det kan også være nyttig å se hvilke andre typer data man kan få nyttig og relevant informasjon fra. Eksempelvis andre entiteter fra MES, og eventuelt andre loggdata, som systemlogger, fra klient-maskinene. Ved å legge til flere typer logger, vil man nødvendigvis måtte tilrettelegge for dette i både databasen og mes-monitoring-prosjektene, med tilhørende klasser for sending og mottak av disse loggene.

Det kan med fordel brukes en bedre og mer sikker autentiseringsløsning enn det vi endte opp med, "basic access authentication". Eksempelvis JSON Web Token (JWT) eller en OAuth-flyt. Dette vil medføre at man slipper å sende brukernavn og passord for hvert eneste kall til REST APIet. Likevel kan dette medføre andre typer problemer, eksempelvis når man må logge inn på nytt når en utstedt token blir ugyldig, ettersom brukeren av APIet er en datamaskin som ikke vet hvordan man logger inn på et nettsted slik en menneskelig bruker intuitivt vil gjøre.

Til tross for kontinuerlig oppdatering og forbedring av kodebasen, så er det vanskelig å holde høy kvalitet på all kode, spesielt for relativt uerfarne programmerere som bachelorstudenter. Derfor er det anbefalt å gå gjennom koden for å se forbedringspotensialet. Et konkret eksempel det er verdt å se på er klassen `MonitoringEntityController` i mes-monitoring-client-prosjektet. Der bør man se på muligheten for å generalisere post-metodene for hver enkelt entitet til eksempelvis én generell insert- og én generell update-metode. På den måten slipper man å legge til en ny metode for hver nye entitet, eller datatype, som skal loggføres.

For å bevare alle loggdataene, bør man implementere en løsning som tar vare på loggene dersom databasekallene feiler. Per nå blir loggene slettet fra køen, selv om de ikke blir satt inn eller oppdatert i databasen. Dette vil føre til feil dersom en oppdatering, fra oppdateringskøen, blir forsøkt utført før en innsetting til databasen blir gjort. I tillegg har vi ikke fått laget noen mekanismer for å bevare loggene dersom det blir kommunikasjonsbrudd mellom klienten og serveren eller mellom serveren og databasen.

Det bør også lages automatiske tester for hele prosjektet, både i klient- og server-modulene. Dette inkluderer enhetstester (Unit tests) for alle offentlige metoder, med mock-modeller for REST APIet, DAO'ene og Consumerne. Spring har støtte for dette med blant annet MockMVC og minnedatabaser i test- og mock-bibliotekene (Spring, 2019).

Vi har kun skrevet automatiske tester for entitetene, for å sjekke om valideringsfunksjonen virket som den skulle. Likevel har vi testet hele prosjektet med integrasjonstester og systemtester

under hele utviklingsperioden. Vi begrunnet denne beslutningen med at det var viktigere å se at det helhetlige produktet fungerer enn alle de individuelle elementene var grundig testet. Dersom vi skulle gjort noe annerledes, ville det blant annet inkludert grundig enhetstesting, ettersom dette ville gjort det lettere for videre utvikling, samt fungere som eksempelkode for andre som tar over produktet senere.

11.2 Database

Vi holdt oss til bare kundeinstanser i stedet for å ha en ekstra tabell over kunder i denne omgang, men for en dypere visualisering på kunde-basis kan en kunde-tabell legges til uten problemer, hvor hver kundeinstans tilhører en gitt kunde. På den måten kan man se for eksempel hva slags feil som forekommer oftest hos en gitt kunde i forhold til andre kunder, over alle kundeinstansene deres.

Memory- og Disk-tabellene sine “used”- og “total”-kolonner har typen “BIGINT” som er en 64-bits signert integer-talltype og lar oss lagre tall opptil 9,223,372,036,854,775,807. Alminnelig “INT” viste seg å være for lite for oss da dens 32-bits presisjon kun tillot tall opptil 2,147,483,647, som vil si rundt 2 Gigabyte. Siden de fleste maskiner per i dag har både mer minne enn 2 GB, for ikke å nevne mer diskplass enn 2 GB, så valgte vi å gå for “BIGINT” for disse kolonnene. Dette viste seg derimot å være problematisk i ettertid da vi fant ut at MSSQL ikke lar oss utføre visse aritmetiske operasjoner som deling på “BIGINT”-kolonner. Vi kunne heller valgt å gjøre kolonnene om til “DECIMAL”-typen som tillater store tall og ikke trenger å konverteres for aritmetiske operasjoner. Grunnet tidspress og fungerende løsninger rundt dette problemet valgte vi å ikke endre fra “BIGINT” til “DECIMAL” for disse kolonnene.

RetentionPolicy-tabellen sjekker ikke om et gitt tabellnavn faktisk matcher en eksisterende tabell. MSSQL har metadata-tabeller for å lese ut tabellnavn over eksisterende tabeller og lignende, så med “triggers” ville det vært mulig å legge inn begrensninger på at rader i RetentionPolicy-tabellen må matche en eksisterende tabell i databasen vår. Grunnet tidsbegrensninger og lite behov for en slik begrensning valgte vi å ikke implementere denne triggeren i RetentionPolicy-tabellen.

11.3 Visualisering

I praksis blir vi aldri “ferdig” med visualiseringen til dette prosjektet da ønsker, krav og behov endres kontinuerlig til enhver tid.

- Flere grafer som måler andre elementer/tabeller innad i Goodtechs DB.
- Videre finpussing på modulene vi allerede har.
- Vi kan benytte flere av Grafanas funksjonaliteter. Deriblant varsling når noe kritisk skjer, en større utbredelse av panelvariabler eller om det er et ønske for en tilpasset plugin for noe spesifikt.

Grunnen til at dette ikke var tatt med i prosjektet var at vi ikke hadde tid, det funket ikke med det vi hadde så langt i utvikling eller at vi ikke hadde noen umiddelbar nødvendighet.

Det kan bli gjort endringer for å generalisere de grafene som allerede er i løsningen der noe kode går igjen i flere spørringer. Om Goodtech vil tilby dette produktet for sine kunder så må det bli satt opp en egen løsning for dette, om dette blir da en ny instans av Grafana eller en helt ny løsning blir en vurderingssak igjen. Det er ikke mangel på videreutvikling i denne delen av prosjektet. Det blir heller et spørsmål om tid investert og penger spart ved å ha produktet.

11.3.1 Varsling

I prosjektskissen står det at varsling per epost eller SMS var ønskelig. Vi tok ikke dette med i forprosjektrapporten vår fordi vi mente dette var noe som kunne være en del av en videreutvikling og ikke var et fokusområde til oppgaven.

Etter at vi hadde jobbet en del med oppgaven og kommet et stykke ut i prosjektet der vi hadde begynte å spørre om mer å gjøre ble dette temaet tatt opp igjen. Til da så hadde vi kun jobbet med multi-serielle grafer, det vil si grafer som har flere enn en linje i grafen og/eller tabellen. Grafana har støtte for varsling og etter hva vi har sett et greit oppsett for å få dette til å gå, men det var noen faktorer som gjorde at vi ikke inkluderte dette i prosjektet.

Grafanas løsning på varsling er en blanding av “For-If”-løkker. Man setter i modulen at f.eks. gjennomsnittet av spørring A, fra 5 minutter tilbake til nå, skal alltid ligge under 20. Om grafen overstiger denne verdien vil Grafana sette denne varslingen fra “OK” til “ventende”. Grafana vil ikke sende noe varsling i denne perioden. Når varslingen har kjørt i lengden definert ovenfor først da vil den gå fra “ventende” til “varsler” og deretter sende notifikasjoner til gitte kanaler. Dette er for å unngå scenarioer der for eksempel en modul som viser CPU-bruk ikke skal fyre av varslinger der den ligger på 99% bruk i 1 minutt.

Med disse begrensningene så kom vi fram til to grunner hvorfor vi ikke inkluderte dette i oppgaven. Som forklart ovenfor så er de aller fleste av modulene en form for multi-seriell spørring. Dette er noe Grafana ikke støtter for varsling. Da varsling ble tatt opp som tema så hadde vi ikke tenkt at dette skulle være et problem. Slik det fungerer nå; en varsling går av i en modul, den står og går i “varsler” stadiet og sender en notifikasjon. Om en annen instans i samme modul går over grensen som er definert så vil den da ikke bli sendt som er varsel fordi selve modulen/spørringen er allerede i “varsling” modus. Dette vil da si at i de tilfellene der flere instanser må varsle vil bare den første gi en notifikasjon.

Som nevnt ovenfor blir varslinger definert med “er over” eller “er under” eller lignende uttrykk og til slutt et nummer. Dette nummeret er ikke variabelt, altså man kan ikke si at en unik linje i en multi-seriell graf skal ha denne verdien og en annen skal ha en annen verdi for grense til varsling. Dette blir et problem med hvordan vi har utviklet hele visualiseringen.

Som et siste punkt så var det ønskelig at varslingen skulle være via SMS. Dette har Grafana ikke støtte for (Grafana, u. å., Alert Notifications). Det har kun støtte for epost og andre kanaler som ikke blir brukt av Goodtech, eller kanaler de ikke ønsker slike varslinger over. Dette vil da si at vi måtte lage en standalone modul som kjører de samme spørringene som visualiseringen for å følge med på om ting må varsles.

Noen av disse problemstillingene kunne blitt unngått ved å dele opp spørringene til å være for hver instans, altså ha flere av samme spørring per modul med forskjellige kundeinstanser. Dette vil igjen skape mer stress på serveren der den samme spørringen må kjøres for X antall kunder for 9 moduler, tatt hoved-dashbordet i betraktning. Det ville også være unødvendig mye vedlikehold ved å gjøre dette på denne måten der man i en startfase trenger å endre på varslingsgrensen da dette ikke er gitt i alle modulene.

Tatt alt i betraktning så vi at vi ikke hadde tid til å utføre dette og at den eneste løsningen vi har kommet frem til krevde for mye vedlikehold av Goodtech i senere tid. Dette medførte at vi ikke tok dette med i sluttproduktet.

12 Alternativer

12.1 Backend

Vi ble enige i kravspesifikasjonen at vi skulle benytte oss av Java for å utvikle backend-delen av prosjektet vårt, siden Goodtech sitt MES allerede er skrevet i Java. Det var derfor ikke aktuelt å vurdere alternative programmeringsspråk til å utføre denne delen av oppgaven.

Det som kunne vært et reelt alternativ er det som faktisk står i kravspesifikasjonen; å lese dataene fra en database plassert hos kunden. Dette ville medført flere spørringer mot databasen, samt håndtering av loggene basert på tid for å unngå dobbeltlagring. Man ville også vært avhengig av at databasen var tilgjengelig, noe man ikke er i den løsningen vi endte opp med.

Videre kunne det vært et enklere, dog en del mindre sikkert, alternativ å hviteliste IP-adresser og brukernavn fremfor å bruke en autentiseringsløsning slik vi har gjort. Dette alternativet ville man måtte endre på uansett, dersom man skulle implementere løsningen i en produksjonssetting.

12.2 Database

Som tidligere nevnt forsøkte vi å bruke InfluxDB for langtidslagring av dataene i en kort periode, uten hell, men det finnes andre alternativer som potensielt kunne vært mer egnet for oppgaven vår.

12.2.1 InfluxDB

Det er viktig å ta for oss hvilke deler av InfluxDB som faktisk egnet seg til oppgaven vår, og hvordan vi eventuelt kunne forholdt oss til InfluxDB om vi hadde utformet databasen vår annerledes. Da vi forsøkte å bruke Influx DB hadde vi en mer relasjonsorientert tankegang rundt dataene våre. I tillegg hadde vi på det tidspunktet ikke funnet ut hva slags data som skulle visualiseres, eller hvordan dataene skulle visualiseres. Dermed prøvde vi veldig mange forskjellige typer spørringer, mange med en mer relasjonsorientert database i tankene, på InfluxDB. InfluxDB er en tidsserieorientert database (influxdata, u.å.). Vi hadde store vansker med å skrive mer kompliserte spørringer mot InfluxDB grunnet vår mangel på erfaring med andre databasesystemer enn relasjonsorienterte databasesystemer som MySQL og MSSQL.

Hadde vi heller tenkt på individuelle "målinger" i stedet for å føre data fra MES-ene inn i databasen vår hadde vi nok hatt bedre hell. Per nå så har vi hovedsakelig to tabeller som var store problemer i InfluxDB: tabeller over antall feilede StatusLogEntry- og QueueElement-rader

den siste timen i forhold til gjennomsnittlig antall feil hver time de siste tre månedene. Hadde vi benyttet InfluxDB så hadde det nok gitt mer mening å dele opp StatusLogEntry og QueueElement i flere målinger i databasen, hvorpå vi da kunne bestemt hva slags data som skulle inn i hver måling, i tillegg til å bearbeide dataene slik vi ønsket dem, i loggbehandleren. Vi kunne hatt en måling hver for "feil den siste timen i forhold til de siste tre månedene" hvor det bare ville vært å lese den siste raden for å få de nyligste dataene, for eksempel, hvor selve kalkulasjonen for hver av målingene ville forekommet i loggbehandleren, eller lignende.

Selv etter å ha filosofert mer rundt riktig bruk av InfluxDB føler vi at valget vårt om å gå tilbake til MSSQL har vært et riktig valg da vi på det tidspunktet vi byttet tilbake ikke visste helt hvordan vi skulle visualisere dataene, for ikke å nevne at vi ikke visste helt hva som skulle loggføres. MSSQL tillot oss å test rundt helt fritt i denne perioden med usikkerhet.

12.2.2 Elasticsearch

Elasticsearch, utviklet av elastic, er en analytisk og søkbar database som har evne til å analysere dataene i databasen og lar en gjøre avanserte søk og spørringer på dataene i databasen (elastic, u.å.).

I gruppen vår har kun Tommy erfaring med Elasticsearch. Vi tittet ikke mye på Elasticsearch som et alternativ da vi heller ønsket en enklere og mer tilgjengelig databaseløsning gitt det vi kunne fra før av, da vi hovedsakelig har hatt erfaring med MSSQL fra studiene våre.

Elasticsearch, sammen med andre produkter fra Elastic, kunne gitt oss et ekvivalent, om ikke bedre, resultat til dette prosjektet. Elasticsearch, sammen med visualiseringsløsningen Kibana, dataprosesseringsmotoren Logstash og dataauthentingsmodulene Beats, ville gitt oss en total løsning på hele monitoreringsoppgaven vår beskrevet i denne rapporten. Dessverre hadde vi ikke tilstrekkelig erfaring eller viten om Elasticsearch og relaterte produkter til å ta oss tid til å se på Elasticsearch, Kibana, Logstash og Beats.

Nøyaktig hvordan denne oppgaven ville blitt med elastic sine produkter er vanskelig å si, men med tanke på at oppgaven vår i stor grad går ut på dataanalyse ville vi nok ikke hatt et problem i lengden med å få Elasticsearch til å fungere for oss. Selve Elasticsearch-databasen er en NoSQL-database hvor spørringene utføres i form av JSON-strenger som beskriver hvilke data vi vil ha ut, samt hvordan de skal finnes frem, hvordan de skal aggregeres osv. (elastic, u.å.).

Den store utfordringen vi ser om vi hadde brukt Elasticsearch hadde vært å tilpasse dataene til databasen. På lik linje som med InfluxDB så er ikke Elasticsearch egnet for å føre relasjonsdata inn direkte, og det gir mer mening å tilpasse dataene til databasen for å få fullt utbytte av Elasticsearchs analyse og søkemotor.

12.2.3 PostgreSQL og MySQL

PostgreSQL og MySQL er ikke veldig forskjellige i forhold til MSSQL til vårt bruk, da alle tre er SQL-baserte relasjonsorienterte databaser som støtter slikt som fremmednøkler, indeksering, delspørringer og joins, samt alle datatypene vi har behov for. PostgreSQL har dog støtte for egendefinerte typer og har støtte for egendefinerte aggregeringsfunksjoner (The PostgreSQL Global Development Group, u.å.).

MySQL, derimot, har ikke støtte for noe spesifikt som MSSQL ikke støtter, annet enn at MySQL støttes av flere programmeringsspråk enn MSSQL. Annet enn dette er det ingen relevante forskjeller som skiller MSSQL og MySQL fra hverandre til vårt formål.

PostgreSQL kunne vært et potensielt alternativ til MSSQL, men vi mistenkte at det ikke ville hatt en særlig betydning på sluttproduktet. MySQL ville i praksis ikke gitt noen forskjell for oss da MSSQL og MySQL er funksjonelt nesten identiske for vårt formål, og siden Goodtech har god erfaring med MSSQL fra før av, samt lisens for MSSQL vi kan bruke, valgte vi å gå for MSSQL i stedet for MySQL eller PostgreSQL.

12.2.4 Prometheus

På lik linje med InfluxDB er Prometheus en tidsseriedatabase (Prometheus, u.å.). Mens InfluxDB er mer fokusert på målinger har Prometheus et større fokus på hendelser, og lagring av hendelsesdata. Denne forskjellen utarter seg i hvordan de to databasene lagrer data på disk, samt i form av at InfluxDB har støtte for tidsstempler med nanosekundsprevisjon, mens Prometheus kun har støtte for opptil millisekundsprevisjon. InfluxDB har som tidligere nevnt en "tag-field" inndeling av data per punkt i en måling (rad i en tabell), hvor InfluxDB har visse begrensninger på hvordan den kan behandle hver av dem. Prometheus har ingen slik inndeling per hendelse som er lagret, og hver hendelse har et sett med attributter som beskriver hendelsen (Prometheus, u.å.).

Uten å ha prøvd Prometheus får vi dermed inntrykk av at Prometheus er mer egnet for den typen data vi håndterer enn InfluxDB, på den måten vi håndterer den på. I loggbehandleren bruker vi bokstavelig talt hendelseslyttere ("event listeners") for å lytte på når nye StatusLogEntry- og QueueElement-objekter opprettes. Prometheus kunne dermed passet oss veldig godt, men vi valgte heller å få for en mer kjent database i form av MSSQL for å redusere kompleksiteten av dette prosjektet.

12.3 Visualisering

Det er som nevnt tidligere andre visualiseringsverktøy som kunne blitt benyttet i dette prosjektet deriblant chronograf, D3.js⁴⁰, Plotly⁴¹, Cluvio⁴² og Chronograf⁴³. Cluvio, Plotly og Chronograf fungerer på en ganske lik måte som Grafana med små forskjeller. Chronograf er kun for InfluxDB, noe vi var innom, men vi ikke hadde hadde ikke noe utgangspunkt i, så plattformen kunne vært egnet i at den tilbyr det samme som Grafana. Likevel ble det feil DB løsning for vår backend. Cluvio og Plotly fungerer veldig likt som Grafana etter hva vi så på deres hjemmesider, både i funksjon og design.

D3.js er det sterkeste alternativet som vi kunne benyttet. Dette biblioteket bruker Javascript, CSS og html for å hente ut og visualisere spørringer. Det tilbyr mye mer kontroll over utseende på siden og mer kontroll over hvordan moduler kan se ut. D3.js har ingen innebygd tidsfilter slik Grafana har, så dette er noe vi eventuelt måtte bygd selv. På en annen side så har det mange tredjeparts-biblioteker som vi kunne tatt i bruk, uten å ha gjort et grundig søk på dette, men det er klart ved å se på GitHub dokumentasjonen under plugins at det er mer støtte for dette enn Grafana.

Vi så ingen hensikt i å lage en egen web-applikasjon for visualiseringen. Det ville ta vekk fokuset på selve spørringene som ble kodet, noe som var mer kritisk å få riktig enn å ikke bruke tredjepartsverktøy. Vi så også etter en løsning som passet prosjektet med tanke på tid og kravspesifikasjonene som ble satt. Det ble anbefalt av Goodtech å bruke Grafana da det var noen i firmaet som allerede hadde tatt dette i bruk. Etter litt etterforskning så at Grafana var godt egnet til dette prosjektet. Det var enkelt å sette seg inn i, krever minimalt med vedlikehold, og hadde de basisfunksjonene som vi var ute etter.

⁴⁰ <https://d3js.org/>

⁴¹ <https://plot.ly/>

⁴² <https://www.cluvio.com/>

⁴³ <https://www.influxdata.com/time-series-platform/chronograf/>

13 Oppsummering

Denne rapporten tar for seg planlegging, utførelse og dokumentasjon av bacheloroppgaven “Goodtech Monitoring”, hvor vi har utviklet en komplett løsning for visualisering av loggdata hentet fra MES-instansene hos Goodtech sine kunder. Vi begrunner valgene og endringene vi har utført, redegjør for alternativer til vår løsning og reflekterer over potensielle forbedringer og videreutvikling.

Rapporten forklarer om kommunikasjon mellom de individuelle delene av løsningen: klient, server, database og visualisering, i tillegg til dataflyten gjennom hele løsningen fra MES til visualisering. Videre skriver vi om utviklingsprosessen knyttet til prosjektet, sikkerhetsaspektene rundt nettverkskommunikasjon mellom modulene, og drøfter forbedringspotensialet som vi ikke rakk å utføre eller som ble nedprioritert.

Opgaven har vært en lærerik prosess hvor vi har fått gode innblikk i planlegging, design, utførelse, dokumentasjon og vedlikehold av et større programvaresystem. Vi har i tillegg jobbet med relevant teknologi og verktøy som brukes av IT-konsulenter i arbeidslivet. Dette er verdifulle erfaringer vi tar med oss videre.

Ved å jobbe tett sammen i en gruppe over en lengre periode har vi fått innblikk i hvordan det er å jobbe på et prosjekt med andre med forskjellig kompetanse og nivå av faglig forståelse. Vi har komplementert hverandres kompetanser og bidratt til individuell utvikling på disse områdene. Ved å ha vært tilstede hos oppdragsgiver under hele utviklingsprosessen har vi fått et godt innblikk i hvordan en normal arbeidsdag er for en konsulent i et norsk teknologikonsern.

I løpet av prosjektet har vi oppdaget en del fallgruver som ofte skjer i slike prosjekter, og sammen klart å løse de fleste av disse problemene. Disse erfaringene gjør at vi føler vi står sterkere rustet til arbeidslivet, og vi kan identifisere og løse disse utfordringene tidligere.

Vi er fornøyde med prosjektoppgaven og løsningen vi leverer som en helhet, og tilbakemeldingene vi har fått underveis tilsier at Goodtech kan bruke vår bacheloroppgave i deres produksjonssystemer med kun noen få endringer.

14 Referanser

- Anchor Modeling. (u. å.). About. Hentet fra http://www.anchor modeling.com/?page_id=2 den 21. mai 2019.
- Chacon, S. & Straub, B. (2014). About version control. *Pro Git*. Hentet fra <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>
- elastic. (u.å.). Elasticsearch: RESTful, Distributed Search & Analytics. Hentet fra <https://www.elastic.co/products/elasticsearch> den 16. mai 2019.
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J.(1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley
- GeeksForGeeks. (u.å.). Introduction of 4th and 5th Normal form. Hentet fra <https://www.geeksforgeeks.org/dbms-introduction-of-4th-and-5th-normal-form/> den 21. mai 2019.
- Grafana. (u.å.). Using Microsoft SQL Server in Grafana. Hentet fra <https://grafana.com/docs/features/datasources/mssql/#database-user-permissions-important> den 21. mai 2019.
- influxdata. (u.å.). InfluxDB - The Open Source Time Series Database. Hentet fra <https://www.influxdata.com/products/influxdb-overview/> den 16. mai 2019
- Microsoft. (2019). mssql-jdbc/tdsparser at master. Hentet fra <https://github.com/microsoft/mssql-jdbc/blob/5c5c4d3e4ee73c1570bcf7b4c1447057bf4dda9f/src/main/java/com/microsoft/sqlserver/jdbc/tdsparser.java> den 20. mai 2019.
- Prometheus. (u.å.). Prometheus - Monitoring system & time series database. Hentet fra <https://prometheus.io/> den 16. mai 2019.
- Repository. (u.å.). I *Merriam-Webster Dictionary*. Hentet fra <https://www.merriam-webster.com/dictionary/repository> den 15. mai 2019.
- Reschke, J. (2015). *The 'Basic' HTTP Authentication Scheme* (s3-5). Hentet fra <https://tools.ietf.org/html/rfc7617> den 16. mai 2019.
- Spring (2019). *Testing*. Hentet fra <https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/testing.html> den 11. mai 2019.
- Studytonight. (u.å.). 1ND, 2NF, 3NF and BCNF in Database Normalization. Hentet fra <https://www.studytonight.com/dbms/database-normalization.php> den 21. mai 2019.
- The PostgreSQL Global Development Group. (u.å.). PostgreSQL: About. Hentet fra <https://www.postgresql.org/about/> den 16. mai 2019.
- Winand, M. (u.å.). More indexes, slower INSERT. *Use The Index, Luke*. Hentet fra <https://use-the-index-luke.com/sql/dml/insert> den 15. mai 2019.
- Grafana, u. å. Alert Notifications, Alerting, notification, Supported Notification Types. Hentet fra <https://grafana.com/docs/alerting/notifications/#supported-notification-types>

15 Vedlegg

15.1 Vedlegg 1 - Ordliste

- API: Application Programming Interface. En del av et system eller program som er tilgjengelig for andre systemer og programmer å interagere med for å kommunisere med systemet.
- Basic access authentication: En autentiseringsmetode der brukernavn og passord blir sendt sammen med en HTTP-forespørsel (request) (Resche, 2015, s. 3-5). Brukernavn og passord blir oversatt til Base64-kode, men ikke kryptert eller hashet (Reschke, 2015, s. 8).
- Escaping (om tekststrenger): Escaping er en måte å hindre at verdier, eller ord, i en tekststreng adskilt av mellomrom, eller linjer, blir tolket som flere strenger etter hverandre. Escaping kan også brukes for å hindre at enkelte tegn, som for eksempel anførselstegn eller skråstreker, blir tolket som kommandoer fremfor tegnsetting eller bokstaver i en tekst.
- Indeks (i en database): en optimalisert oversikt over en gitt tabells kolonne som øker ytelsen ved søking gjennom tabellen.
- Kundeinstans: en instans av MES som kjører på systemene til en kunde. En kunde kan ha flere kundeinstanser.
- MES: Manufacturing Execution System. Et system som holder en overordnet kontroll på flere enheter i en produksjonsprosess og kan kontrollere dem basert på overordnede forespørsler og instruksjoner.
- MSSQL: Microsoft SQL Server 2017
- Prepared statement: En forhåndsagret SQL-spørring. En slik spørring kan utføres flere ganger, med forskjellige verdier, uten å måtte endre på spørringen. Prepared statements gjør det i tillegg umulig å endre på SQL-delen av spørringen, så man unngår SQL injections.
- Random walk-algoritme: en algoritme hvor en gitt verdi blir generert ved å legge til en tilfeldig generert endringsverdi på den nåværende verdien.
- Repository (Repo): En lagringsplass (Repository, Merriam-Webster). Brukes i kombinasjon med versjonskontrollsystem for å beskrive en adresse, mappe eller server hvor filer og historikk lagres. Inneholder et historietre administrert av et VCS som holder rede på endringer over tid i en gitt mappe.
- REST: Representational State Transfer. En type API som eksponerer et system eller et program for utenomverdenen på tilkoblede datanettverk. Benytter HTTP som overføringsprotokoll.
- Retention Policy: hvor lenge en gitt rad med data skal oppbevares i databasen. Eksempel: en retention policy på 180 dager vil slette *alle* rader fra databasen som er eldre enn 180 dager.

- Scope: Et sett med krav og spesifikasjoner som beskriver en prosess eller et endelig produkt.
- Scope creep: Utvidelse av prosjektets scope underveis i planlegging eller utviklingen. Scope creep fører til uforutsett arbeid ettersom det legges til funksjonaliteter og krav som ikke var planlagt på forhånd.
- SQL injection: SQL-injeksjon er en angrepsmetode mot et datasystem der man manipulerer SQL-spørringer for å hente ut data man i utgangspunktet ikke har tilgang til.
- SSMS: Microsoft SQL Server Management Studio 2017
- Tidsserie/tidsrekke (en. time series): En serie, eller rekke, med datapunkter i kronologisk orden. Brukes gjerne for å logge en gitt variabel over tid.
- Trådsikkerhet: Et begrep som brukes om ressurser eller tjenester som kan aksesseres av to eller flere tråder (prosesser) samtidig. En ressurs eller tjeneste er trådsikker dersom samtidige kall eller operasjoner på denne ressursen eller tjenesten ikke medfører at prosessene overskriver, ødelegger for, eller påvirker hverandre på andre uønskede måter.
- Delspørring (en. subquery): en databasespørring som forekommer inne i en annen databasespørring. Hver databasespørring produserer en tabell, og det kan være ønskelig å utføre enda en spørring på en tabell fra en annen spørring. Man kan dermed pakke spørringen rundt den andre spørringen, og behandle den som en réell tabell.
- Versjonskontrollsystem (VCS): Et system som lagrer endringer gjort på filer over tid. Systemet gjør det mulig å se hvilke filer som er endret, hva endringene er, og når endringene er gjort. Det er også mulig å tilbakestille endringer til en tidligere versjon, for eksempel når en endring gjør at man har mistet/slettet filer eller andre typer problemer har oppstått (Chacon & Straub, 2014).