Bruk av containere i NHN

Bacheloroppgave: 121

Trondheim, våren 2019

Av: Henrik Roksvaag og Ole Valla Dønnem

NTNU
Kunnskap for en bedre verden

Forord

Denne bacheloroppgaven er den avsluttende delen av vår utdannelse på NTNU Trondheim, Bachelor i Informatikk: Drift av datasystemer 2016-2019. Vi har gått i dybden på et tema vi hadde begrenset kunnskap om fra før, noe som både har vært utfordrende, dog like spennende som lærerikt.

Vi vil først og fremst spesielt takke vår veileder Stein Meisingseth for god veiledning, støttende og oppmuntrende ord, og god hjelp under hele prosessen. Takk for at du svarer på mail, bruker av din tid på oss og hjelper oss når vi trenger veiledning og nye synspunkter.

Videre vil vi takke Nikolai Thingnes Leira og Ole Morten Berg ved Norsk Helsenett SF for gode tilbakemeldinger og tips. Samt for å legge til rette for at vi kom i mål med bacheloroppgaven. Vi vil også rette en takk til Rune Andreas Grimstad, Halvard Vasstveit, Jack Brennan og Thomas Juberg for hjelp underveis i oppgaven.

Vi vil også gi en stor takk til alle som har bidratt til at vi er der vi er etter tre år med skole, spesielt vil Henrik Roksvaag rette en stor takk til Robert Aakerholm. Alle forelesere vi har hatt på NTNU, spesielt Bjørn Klefstad og Grethe Sandstrak som har vist menneskelig forståelse da Henrik Roksvaag ble pappa i fjor sommer.

Vi vil også gi en stor takk til samtlige medstudenter for god dialog og godt samvær disse tre årene, da spesielt Marius Myhre, Kiet Nguyen og Tormod Lien.

Etter mye arbeid kan vi nå levere en bachelor vi er stolte av. Den har bidratt til å utvikle vår egen kompetanse på området. Vi håper andre kan ha glede av å lese denne oppgaven, og gjøre seg noen refleksjoner rundt hvordan og hvorfor Kubernetes er et verktøy for fremtiden.

Trondheim, 18.05.2019

Ole Valla Donnem

Henrik Roksvaag

Sammendrag

Sammendrag av Bacheloroppgaven

Tittel: Bruk av containere i NHN

Dato: 20.05.2019
Deltakere: Henrik Roksvaag og Ole Valla Dønnem

Veiledere: Stein Meisingseth og Nikolai Thingnes Leira
Oppdragsgiver: Norsk Helsenett SF
Kontaktperson: Ole Morten Berg

Nøkkelord: Docker, Windows Server 2019, Kubernetes, containere, containerteknologi, skalering, monitorering

Antall sider: 230

Opphavsrett og tilgjengelighet: Bacheloroppgaven (det skriftlige arbeidet) skal være undergitt utsatt offentliggjøring i 3 år.

Sammendrag:

Bacheloroppgaven ser på bruk av Kubernetes som verktøy for å drifte en av Norsk Helsenett SF sine eksisterende applikasjoner. Oppgaven omhandler bruk av Azure skytjenester, Docker for konvertering av applikasjon, automatisk skalering av applikasjon basert på last ved bruk av Kubernetes. Vi ser videre på bruk av monitoreringsverktøy, og hvilke utfordringer og muligheter man har med monitorering av et container-basert driftsmiljø. Videre vurderer vi sikkerhetsaspektene ved å ta i bruk Kubernetes som driftsplattform. Vi nevner her spesifikt tilgangskontroll og logging av dette gjennom bruk av audit logging.

Summary of Graduate Thesis

Project Title: Using containers in NHN

Date: 20.05.2019
Authors: Henrik Roksvaag and Ole Valla Dønnem

Supervisor: Stein Meisingseth and Nikolai Thingnes Leira
Employer: Norsk Helsenett SF
Contact Person: Ole Morten Berg

Keywords: Docker, Windows server 2019, containers, container technology, scalability, monitoring, Kubernetes

Pages: 230

Abstract:

The Bachelor thesis looks at the use of Kubernetes as a tool for running one of Norsk Helsenett SF's existing applications. The task concerns the use of Azure cloud services, Docker for application conversion, automatic application scaling based on load using Kubernetes. We also look at the use of monitoring tools, and the challenges and opportunities with monitoring a container-based operating environment. Furthermore, we consider the safety aspects of using Kubernetes as the operating platform. We specifically mention access control and logging of this through the use of audit logging.

Oppgavetekst

Bruk av containerteknologi i NHN

Norsk Helsenett SF ønsker å få utredet mulighetene for drift av Windows applikasjoner ved bruk av containerteknologi. Bruk av containere til å drifte applikasjoner er for tiden veldig i vinden. Mange bedrifter har begynt prosessen med å endre sin infrastruktur til å passe en container-basert driftsmodell, der det er mulig. Det er viktig for Norsk Helsenett å holde seg oppdatert på ny teknologi som kan gjøre dem mer effektive og levere bedre tjenester i fremtiden. NHN ønsker å se på om mulighetene rundt containerteknologi er noe som kan hjelpe med dette.

Norsk Helsenett SF drifter allerede noen av sine tjenester som er Linux-baserte ved bruk av containere, men dette er ikke noe som har blitt gjort på Windows applikasjonene deres. De ønsker derfor at en del av oppgaven skal være å forsøke å ta en eksisterende applikasjon (HelseID) som i dag driftes etter en mer tradisjonell modell, og konvertere den til å kjøre som containere.

Norsk Helsenett SF har også driftsansvaret for tjenester som er utsatt for uventet høy pågang (NRK, 2017). Når i tillegg Norsk Helsenett SFs driftsmodell er mer reaktiv, kan det føre til at tjenestene oppleves som trege for brukere eller krasjer helt før Norsk Helsenett SF har fått muligheten til å skalere tjenestene. Mulighetene rundt skalering ved bruk av containere blir derfor et viktig moment i denne oppgaven.

Norsk Helsenett SF behandler sensitiv informasjon og de følger Norm for informasjonssikkerhet. Det blir derfor viktig å ivareta dette i løsninger som blir fremstilt.

Innholdsfortegnelse - Overordnet

Bruk av containere i NHN

Forstudierapport

Av: Henrik Roksvaag og Ole Valla Dønnem

NTNU
Kunnskap for en bedre verden

Revisjonshistorie

Dato	Versjon	Beskrivelse	Forfattere
25.01.19	1.0	Første utkast ferdig	Henrik Roksvaag & Ole Valla Dønnem
18.02.19	1.1	Implementert endringer basert på tilbakemeldinger av veiledere	Henrik Roksvaag & Ole Valla Dønnem
03.03.19	1.2	Ferdigstilt kap.7 kostnadsestimering	Henrik Roksvaag & Ole Valla Dønnem
30.04.19	1.3	Fjernet kostnadsestimering, lagt til kilder, henvist til kilder i tekst og generell ferdigstilling	Henrik Roksvaag & Ole Valla Dønnem

Innholdsfortegnelse - Forstudierapport

1. Introduksjon

Hensikten med denne forstudierapporten er å gi oppdragsgiver (Norsk Helsenett SF) og prosjektgruppen en felles forståelse for hvordan bacheloroppgaven kan gjennomføres på en tilfredsstillende måte. Fokuset i forstudiet er å se på framtidige muligheter med en container-basert driftsløsning. Norsk Helsenett SF har gitt uttrykk for at dagens driftsmodell er særdeles reaktiv og at de ønsker å se på om en container basert driftsmodell kan føre til at de får en mer proaktiv driftsmodell. Det har også vært uttrykt ønske om at NHN vil følge med i tiden, og at det i de seneste årene har vært mye snakk om containere, og da spesielt Docker og Kubernetes.

Prosjektgruppen vil i forstudierapporten komme med et forslag av hva som trengs for å implementere en container-basert driftsløsning.

Det er ønskelig å gjennomføre to konkrete piloter:

1. En egenlaget applikasjon driftet i Docker og Kubernetes
1. Drift av NHN applikasjon i Docker og Kubernetes

1.1 Hensikten med dokumentet

Hensikten med dette dokumentet er å innhente innledende informasjon om containerteknologi til bruk på en Windows plattform, med hovedsakelig fokus på Windows Server 2019. Fokuset i forstudiet er å se på framtidige muligheter med en container-basert driftsløsning. Forstudierapporten beskriver videre bakgrunnen til bachelorprosjektet, hvilke mål vi ønsker å oppnå og hvilke avgrensninger som har blitt gjort. Vi skal se på dagens løsning samt kostnad av implementasjon av ny driftsmodell. Vi ser videre på kritiske suksessfaktorer og utfører en interessentanalyse og en risikoanalyse.

1.2 Oversikt over dokumentet

Kapittel 2: Bakgrunn for prosjektet
Beskriver dagens system og modellen NHN bruker i sitt nåværende Windows applikasjon driftsmiljø. Vi tar for oss en skissering av dagens system og ser på eventuelle problemer med dagens løsning. Vi ser også på fremtidige behov for NHN i form av å flytte drifting av applikasjoner over på en ny container-basert løsning.

Kapittel 3: Prosjektmål
Tar for seg hva prosjektgruppen og NHN ønsker å oppnå med bacheloroppgaven. Videre tar vi for oss omfanget av oppgaven, og de forutsetningene som har blitt gjort. Vi har designet et milepæl diagram i Microsoft Project 2016 som tar for seg de viktigste milepælene i bacheloroppgaven.

Kapittel 4: Interessenter og rammebetingelser
Kapittel 4 inneholder beskrivelser av interne og eksterne interessenter i bacheloroppgaven. Forskjellige interessenter vil som regel ha forskjellig behov til oppgaven, og til resultatet. Det er i dette kapittelet satt en del rammebetingelser for bacheloroppgaven fra både NHN og NTNU sin side.

Kapittel 5: Kritiske suksessfaktorer
I kapittel 5 tar vi for oss de kritiske suksessfaktorer som må være på plass for å få et vellykket resultat. Kritiske suksessfaktorer beskriver hvilke faktorer som vil være kritisk avgjørende for prosjektets utfall.

Kapittel 6: Risikoanalyse
Kapittelet om risikoanalyse inneholder en risikoanalyse. Der vi tar for sannsynlighet, konsekvens og risikofaktoren for at uønskede hendelser som kan negativt påvirke bachelorprosjektet skulle oppstå.

Kapittel 7: Retningslinjer og standarder
Retningslinjer og standarder tar for seg kravene til dokumentasjon, kvalitetsgjennomgang, standard, og håndtering av ønskede endringer fra interessenter i prosjektet.

Kapittel 8: Prosjektorganisering
Prosjektorganisering viser hvem som er involvert i prosjektet og hvordan prosjektgruppen har fordelt arbeidet seg imellom.

Kapittel 9: Anbefaling om videre arbeid
Inneholder en oppsummering på av det vi har funnet ut av i forstudiet, og en anbefaling om videre arbeid med oppgaven.

Kapittel 10: Kilder
Inneholder kildehenvisninger til de ressursene som er brukt i dette dokumentet.

1.3 Definisjoner og forkortelser

Forkortelse	Begrep	Definisjon
NHN	Norsk Helsenett SF	
	DevOps	Er en felles tilnærming for organisasjonsmessige og prosessmessige forbedringer, som understreker behovet for samarbeid mellom team for utvikling (Development/Dev) og drift (Operations/Ops).
	On Premise	Med on-premise menes det at du bruker dine egne fysiske servere og IT-infrastruktur til å hoste og drifte.
	Docker	Docker er et dataprogram som utfører virtualisering på operativsystemnivå.
	Kubernetes	Kubernetes er en åpen kildekode-basert plattform for orkestrering, utrulling og håndtering av container-baserte applikasjoner.
	Open-source	Åpen kildekode. Betyr at kildekoden til et dataprogram er gjort tilgjengelig (ofte på Internett) for alle.
OMS	Operations Management Suite	Er en samling verktøy for å administrere on-premise og sky infrastruktur.
SaaS	Software as a service	Gir brukere mulighet til å koble til og bruke skybaserte apper over Internett (for eksempel Office 365)

IaaS	Infrastructure as a service	Leveres som en datainfrastruktur som en tjeneste over et nettverk. Brukerne har kontroll over applikasjoner, servere, operativsystem og lagring.
PaaS	Platform as a Service	Brukerne utvikler applikasjoner i leverandørens nettsky-infrastruktur gjennom å benytte programmeringsspråk og verktøy støttet av leverandøren.
	Octopus Deploy	Automatisert utrulling av nye versjoner av applikasjoner.
	Microsoft Azure	Microsoft Azure er en samling av skytjenester for utvikling, bruk og administrering av intelligente programmer via et globalt datasenternettverk
SCOM	System Center Operations Manager	Monitoreringsløsning til Microsoft.
	Splunk	Web-basert tjeneste som analyserer maskingenerert data i sanntid.

2. Bakgrunn for prosjektet

Norsk Helsenett SF (NHN) er et statlig foretak, eid av Helse- og omsorgsdepartementet (HOD). NHN sitt oppdrag er å levere og videreutvikle en sikker, robust og hensiktsmessig nasjonal IKT-infrastruktur for effektiv samhandling mellom aktørene i helse- og omsorgstjenesten. NHN har også ansvaret for teknisk drift av en rekke nasjonale tjenester og register, som for eksempel portalen helsenorge.no og kjernejournal (Norsk Helsenett SF, 2019).

Drift 2 hos NHN er i vurderingsfasen for å skifte modell på hvordan de drifter tjenester NHN leverer. De har i dag en tradisjonell driftsmodell og er interessert i å se hvilke muligheter og fordeler man kan få med en overgang til en mer moderne container-basert driftsmodell i sitt Windows applikasjonsmiljø. Med å ta i bruk en container-basert driftsmodell kan man potensielt:

- Forenkle utrulling, konfigurering og oppdateringer av applikasjoner.
- Øke effektivisering av hardware.
- Forbedre skalerbarhet.
- Gjøre utvikling og drift (DevOps) mer effektivt og agilt.

Med tidligere versjoner av Windows Server har det ikke vært spesielt god støtte for bruk av containerteknologi og det har ikke vært veldig stor interesse rundt dette i NHN, selv om de tar i bruk container teknologi på et par av linux applikasjonene de drifter og har hatt gode erfaringer med det.

Som plattform for oppgaven, ønsker både arbeidsgiver og prosjektgruppen at det er ønskelig å ta i bruk Azure. Det er samtidig uttrykket et ønske om å se på en On Premise implementasjon. Da NHN sine tjenester hovedsakelig er driftet On Premise. Azure følger en bruksbasert prismodell som kan tilby følgende:

- Enklere skalering av infrastruktur og tjenester
- Tilgang til og utrulling av teknologier som container-basert applikasjonsutvikling og drift
- Virtuelle maskiner og nettverk (IaaS)
- Rene plattformtjenester for web og database uten behov for drift av operativsystem (PaaS)
- Enkel tilgang til programvare SaaS)

Azure kan gi lavere driftskostnader enn tradisjonell drift i eget datasenter, vi sier ofte at det vil gi lavere TCO (Total cost of ownership).

Vurderingen av skytjenester er i tråd med regjeringens “Nasjonal Strategi for bruk av skytjenester” (Regjeringen, 2016) og Digitaliseringsrundskrivet, der det heter at:

> *"Bruk av skytjenester kan gi økt fleksibilitet og mer kostnadseffektiv bruk av IKT. Virksomheter som etablerer nye eller oppgraderer eksisterende fagsystemer eller digitale tjenester, eller endrer eller fornyer avtaler knyttet til drift, skal vurdere skytjenester på linje med andre løsninger.. Det er en forutsetning at valgt løsning tilfredsstiller virksomhetens krav til informasjonssikkerhet."*
> (Regjeringen, 2017)

Med lansering av Windows Server 2019 har Microsoft investert mye i å forbedre støtte for bruk av containerteknologi og har hatt et nært samarbeid med Docker i utviklingsfasen for å få til dette. Docker er et open-source prosjekt som tar i bruk containerteknologi til å lage og bruke containere, for både Windows og Linux. Man kan se på containere som modulære, ekstremt lettvektige virtuelle maskiner.

NHN ønsker en utredning og vurdering av en modernisert container-basert driftsmodell, med hovedfokus på Windows Server 2019, Docker, og Kubernetes. Dette skal skje ved å sammenligne en moderne container basert driftsmodell opp mot systemet de i dag bruker. Vi vil se på brukervennlighet, skalerbarhet, kostnader og sammenligne egenskaper til de to driftsmodellene.

2.1 Beskrivelse av problemer og behov

2.1.1 Beskrive av problem og behov

NHN drifter mange tjenester, både interne og eksterne, og flere av dem er store, kompliserte og tunge systemer. Disse tjenestene kan være kritiske for deres kunder og det er derfor viktig at de er sikre og effektive, både med tanke på tilgjengelighet (oppetid) og kostnad. NHN konkurrerer også med andre bedrifter om anbud for å drifte både private og offentlige tjenester og det er derfor viktig at de holder seg oppdatert på ny teknologi og mulighetene rundt dette.

Innimellom nevnes en av tjenestene NHN drifter i media eller noe blir et hett tema som gjør at pågangen på en tjeneste kan bli mye høyere enn vanlig. Per dags dato er ikke det systemet og driftsmodellen som brukes veldig bra til å håndtere dette og ved behov for skalering kan det ta lengre tid enn ønsket for å få det på plass. Med andre ord er skalerbarheten ikke helt tilfredsstillende ved unormal høy last og vi vil derfor se på mulighetene Docker/Kubernetes har for å løse dette problemet.

Flere av tjenestene NHN drifter kan sees på som ekstremt kritiske og nedetid ved utrulling av nye tjenester eller oppdateringer av nåværende tjenester er derfor noe som er et veldig viktig moment for NHN. I dag brukes Octopus Deploy som er et meget godt verktøy og NHN har ikke noen planer om å finne en erstatning for dette, men det er ønskelig å se på mulighetene for integrasjon av Kubernetes med Octopus Deploy for å øke effektiviteten av utrullinger og minske nedetid av tjenester så mye som mulig.

Med en tradisjonell driftsmodell blir ikke hardware ressurser utnyttet til sitt fulle. En server har gjerne en eller to tjenester som kjører på seg og disse tjenestene krever gjerne ikke de fulle ressursene serveren har tilgjengelig til enhver tid. Software installert på disse serverne kan også være overflødig i forhold til det tjenestene trenger og bruker mer ressurser enn nødvendig. Et eksempel kan være operativsystemet som ligger i grunn, det er ofte mange services/roles og prosesser som kjører som ikke er direkte nødvendige for tjenestene som skal kjøres oppå dette. Med et slikt oppsett er det vanskelig å utnytte de ledige ressursene man har tilgjengelig til enhver tid og skalering (både opp og ned) som vi har nevnt tidligere er vanskeligere eller tar lengre tid enn ønsket.

NHN har meget strenge krav til sikkerhet, da de behandler mye sensitiv personinformasjon og drifter kritiske nasjonale tjenester. Det er allerede et godt sikkerhetssystem på plass og det er derfor viktig at en eventuell supplementær driftsmodell følger dette. Med bruk av container teknologi kjøres gjerne mange tjenester eller deler av forskjellige tjenester på samme host. Det er derfor viktig at man har et klart skille mellom de forskjellige tjenestene/applikasjonene og tilgangsrettigheter blir ekstremt viktig å kunne behandle på en enkel og grei måte.

Problem:

- Ikke tilfredsstillende skalerbarhet ved for eksempel unormal høy last.
- Ved behov for skalering, kan det ta lang tid før det er på plass.
- Nedetid ved utrulling (også i produksjonsmiljø).

Behov:

- Behov for å følge med i tiden.
- Økt effektivitet av ressurser.
- Bedre skalerbarhet.
- Container integrasjon med Octopus Deploy.

2.2 Dagens system

NHN omfatter mange forskjellige systemer og flere forskjellige måter å drifte disse systemene på. Da dette prosjektet er begrenset til seksjonen Drift 2 og mer spesifikt applikasjonen HelseID velger vi å se mest på disse. Vi har nevnt flere ganger tidligere at det i dag brukes en mer tradisjonell driftsmodell. Det vi mener med det i denne sammenhengen, når vi sammenligner det med en container basert modell, er en modell der applikasjoner kjøres på enten egne fysiske servere eller virtuelle servere og som regel er de eneste applikasjonene som kjører på disse. I HelseIDs tilfelle kjøres det på 4 virtuelle maskiner i produksjon, 4 virtuelle maskiner i QA og 2 virtuelle maskiner for testmiljø. I kapittel 2.1 snakket vi om hvorfor dette kan føre til

ikke optimal utnyttelse av ressurser. Monitorering skjer via SCOM og varsler blir sendt ut når det oppdages ting som f.eks. høy last på serverene/applikasjonene. Når dette skjer må som regel noen manuelt starte prosessen med å skalere applikasjonen for å behandle den økte lasten og dette kan ha alvorlige konsekvenser hvis kritiske systemer blir trege/utilgjengelige. Som vi ser fører alt dette til at den tradisjonelle modellen blir sett på som mer reaktiv og NHN har et ønske om å bli mer proaktiv.

En container basert modell vil teoretisk gjøre mye av dette automatisk, forhåpentligvis før brukere i det hele tatt merker nedetid, og ved å kombinere det med å kjøres i et skymiljø som Azure vil ressurser kunne utnyttes til sitt fulleste og skalering bli mye enklere.

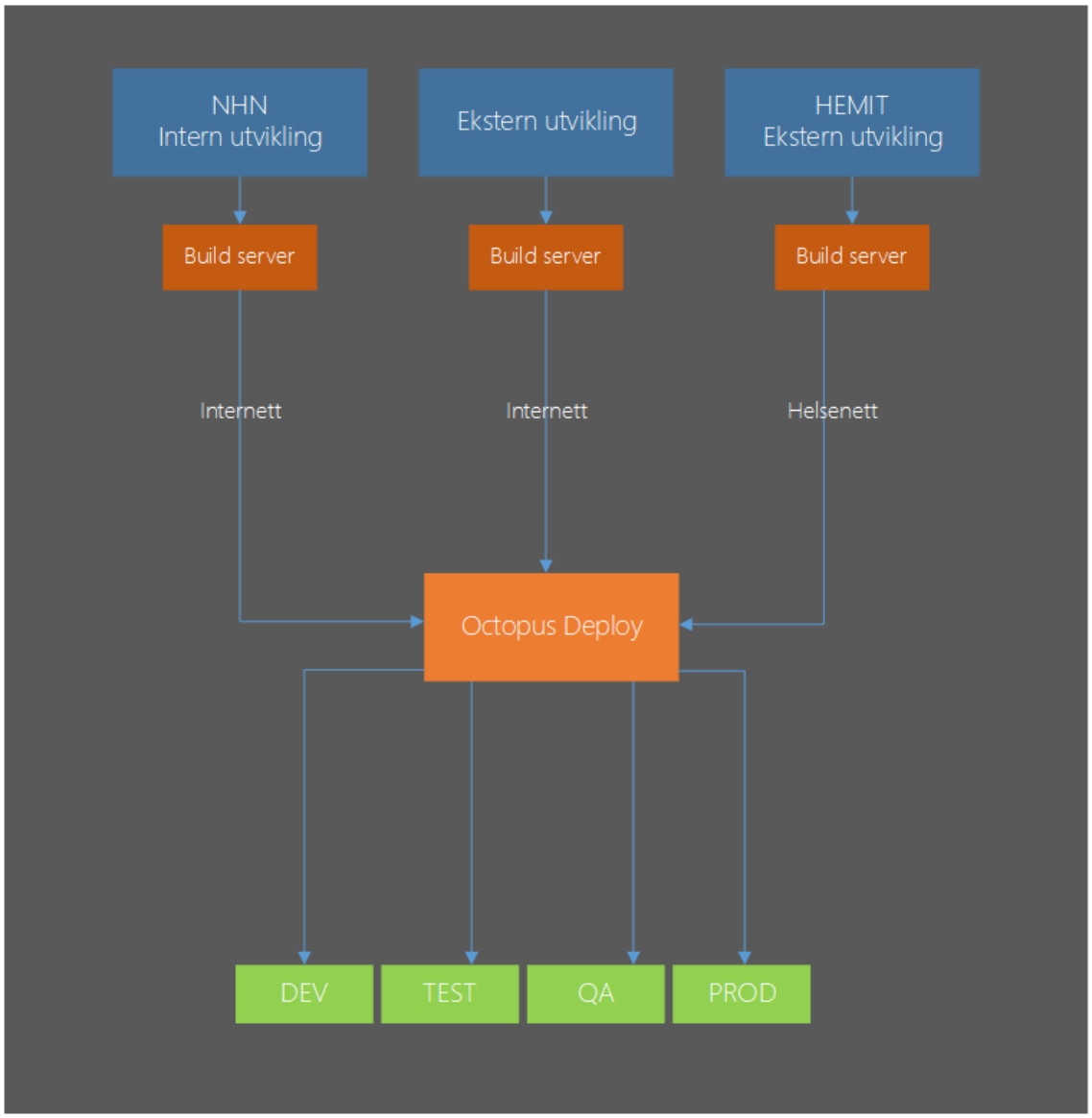
For utrulling av oppdateringer til applikasjoner brukes Octopus Deploy og dette er noe en eventuell ny supplementær driftsmodell må kunne integreres med. Under er Octopus Deploy kort forklart.

2.2.1 Octopus Deploy

Octopus Deploy er en automatisert server for distribusjon av programvare. Den er designet for å forenkle distribusjon av ASP.NET-applikasjoner, Windows Services og databaser. Octopus Deploy brukes til å distribuere applikasjoner sikkert til servere som er lokalisert on premise eller i skyen. Det har et webbasert grensesnitt, som kan brukes til å modifisere og utføre deploys av applikasjoner.

Hos NHN pushes applikasjonspakker enten fra interne eller eksterne utviklere. Trafikkflyten blir som følger.

1. Utviklere commiter kode til et versjonskontrollsystem (for eksempel gitlab). Eller gjennom tett integrasjon direkte i utviklingsverktøy.
2. Kode blir pushet til en build server. Her går trafikken over internett for NHN sin interne utviklingsavdeling, og for alle andre eksterne utviklere, bortsett fra for Hemit som går over Helsenettet.
3. Pakker blir pushet til Octopus server, som er klart til å rulles ut. Pakker følger en ITIL prosess, som gjør at de kan automatisk deploye en ny versjon til utviklingsmiljøet (DEV) når det kommer inn en ny pakke. Når denne har blitt testet og verifisert ok, så kan pakken distribueres videre til TEST, så til QA-miljø hvis det er et QA-miljø for den bestemte applikasjonen og så til PROD i et bestemt endringsvindu.
4. Octopus oppdaterer miljøet og oppdaterer config filer, skifter på app settings, og så videre.
5. Octopus grensesnittet vil til enhver tid vise versjon på de forskjellige miljøene.



Figur 1: Octopus forklaring

3. Prosjektmål

3.1 Effektmål

Effektmål beskriver hvorfor prosjektet er etablert, og beskriver en ønsket fremtidig situasjon som skal oppnås ved å gjennomføre prosjektet (målsetting).

Effektmålene sier noe om hva NHN ønsker å oppnå med prosjektet.

- Kartlegging av bruk av Kubernetes og Docker til å supplere drift av applikasjoner hos NHN, da spesielt på Drift 2 og applikasjonen HelseID.
- Kartlegging av å gå over til en mer proaktiv driftsmodell i forhold til dagens driftsmodell som er mer reaktiv.
- Kartlegge en kost/nytte analyse av dagens system opp mot et nytt system.
- Vurdering av sikkerhet tilknyttet tilgangsrettigheter hvis flere applikasjoner er hostet på samme node.

3.2 Resultatmål

Resultatmål beskriver de konkrete målene med prosjektet. Det innebærer spesifikasjoner til systemet, tidsfrister og kostnader for prosjektet.

- Ta i bruk Kubernetes og Docker på Windows for å drifte en av NHN sine tjenester.
- Monitorere tjenesten ved bruk av SCOM og/eller SPLUNK.
- Ta i bruk automatisk skalering av applikasjon basert på last og antall forespørsler mot applikasjonen.
- Utføre test av self healing mulighetene til Kubernetes ved feil på applikasjon.
- Kartlegge kostnader for bruk av Kubernetes, Docker og monitoreringsverktøy.
- Vurdere mulighetene for å implementere Kubernetes i Octopus Deploy.

3.3 Prosessmål

Prosessmål sier hva prosjektgruppen ønsker å oppnå med selve prosjektarbeidet. Dette omfatter hvilke forventninger deltakerne har til prosjektarbeidet og hva prosjektgruppen i sin helhet vil få ut av prosjektet.

- Økt kompetanse på bruk av Windows Server 2019, Servercore og Nanoserver.
- Økt kompetanse og forståelse ved bruk av Docker, Kubernetes og

Octopus Deploy på Windows.
- Økt kompetanse og forståelse for hvordan man planlegger implementasjon av en ny løsning i en bedrift.
- Økt kompetanse på bruk av MS Project og prosjektdokumentasjon.

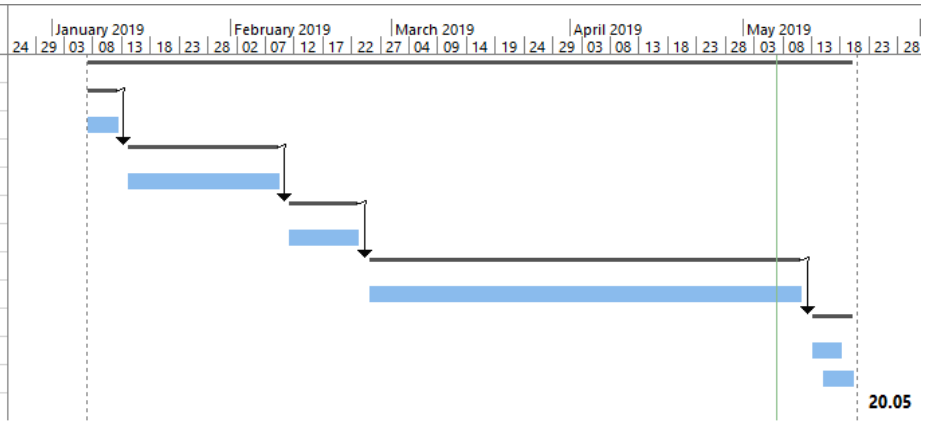
3.4 Prosjektets omfang

Beskrivelsen av prosjektets omfang skal avgrense hva prosjektet beskjeftiger seg med og ikke, for å være så klargjørende som overhodet mulig.
Prosjektgruppen tar for seg bruk av containerteknologi hovedsakelig for Drift 2 sine behov. Vi ser ikke på NHN under det hele. Vi ser også på dette som et isolert prosjekt hos NHN. Vi har på bakgrunn av dette, ikke valgt å inkludere NHN sine kunder med i en interessentanalyse.

3.5 Prosjektets milepæler og hovedaktiviteter

Prosjektets milepæler og hovedaktiviteter er fremstilt grafisk ved hjelp av et Gantt-diagram designet i Microsoft Project 2016. Diagrammet fremstiller prosjektets hovedoppgaver, milepæler og estimert tidsforbruk.

ID	Task Mode	Task Name	Duration	Start	Finish	Predecessors
1		**Bacheloroppgave**	**95,13 days**	**Mon 07.01.19**	**Mon 20.05.19**	
2		**Oppstart**	**5 days**	**Mon 07.01.19**	**Fri 11.01.19**	
3		Planlegging	5 days	Mon 07.01.19	Fri 11.01.19	
4		**Forstudie**	**20 days**	**Mon 14.01.19**	**Fri 08.02.19**	**2**
5		Forstudierapport	20 days	Mon 14.01.19	Fri 08.02.19	
6		**Design**	**10 days**	**Mon 11.02.19**	**Fri 22.02.19**	**4**
7		Designrapport	10 days	Mon 11.02.19	Fri 22.02.19	
8		**Drift**	**55 days**	**Mon 25.02.19**	**Fri 10.05.19**	**6**
9		Driftsrapport	11 wks	Mon 25.02.19	Fri 10.05.19	
10		**Avslutning**	**5,13 days**	**Mon 13.05.19**	**Mon 20.05.19**	**8**
11		Sluttrapport	5 days	Mon 13.05.19	Fri 17.05.19	
12		Presentasjon	3 days	Wed 15.05.19	Mon 20.05.19	
13		Innlevering	0 days	Mon 20.05.19	Mon 20.05.19	



4. Interessenter og rammebetingelser

4.1 Interessentanalyse

4.1.1 Eksterne interessenter (NHN)

De ansatte i NHN er representert av Ole Morten Berg som sitter i styringskomiteen og er seksjonsleder for Drift 2. NHN er også representert av Nicolai Thingnes Leira som er teknisk kontaktperson og veileder under prosjektperioden.

NHNs suksesskriterier for prosjektoppgaven er at det blir gjort en analyse av Kubernetes og Docker, og sammenligner denne opp mot dagens driftsmodell. I tillegg til dette er det viktig for NHN at løsningen skal tilfredsstille "Normen for informasjonssikkerhet". Dette er fordi seksjonen Drift 2 hos NHN behandler store mengder sensitive data. NHN er også interessert i resultatet av en kostnadsanalyse av Kubernetes og Docker sett opp mot dagens driftsmodell.

Da bachelorprosjektet er et internt test-prosjekt, blir ikke sluttbrukerne, som vil være de øvrige ansatte i Drift 2 tatt med i vurderingen. Hvis løsningen skulle blitt implementert hos NHN ved en senere anledning er det naturlig at øvrig driftspersonell ved seksjonen er godt informert underveis i prosjektet. Prosjektgruppen vil allikevel bruke de øvriges ansatte kunnskap og meninger om løsninger underveis i prosjektet.

4.1.2 Interne interessenter (NTNU)

Prosjektgruppen
Prosjektgruppen som den blir omtalt i dette dokumentet, består av Henrik Roksvaag og Ole Valla Dønnem. Prosjektgruppen har selv ansvaret for fremgang i prosjektet, og at det blir gjennomført iht. retningslinjene og rammene gitt av NTNU og NHN.

Prosjektgruppen er agenten i prosjektet, og er den interessenten som har ansvar for planlegging, dokumentasjon og implementasjon av det nye driftssystemet. Prosjektgruppen skal følge problembeskrivelsen, og levere et produkt til oppdragsgiver. Suksesskriterier for prosjektgruppen er å lage en løsning som er tilfredsstillende, og i tråd med de retningslinjer og rammene gitt av NHN og NTNU. I tillegg er det viktig for prosjektgruppen at prosjektet gir merverdi både for NHN og for prosjektgruppen. Med det mener vi at NHN kan bruke vårt prosjekt som kunnskap til videre prosjekter, samt at prosjektgruppen også sitter med nyttig erfaring og kunnskap.

Veiledere
Veileder fra NTNU er universitetslektor Stein Meisingseth. Stein er både veileder og

sitter i styringskomiteen. Han har ansvar for å veilede prosjektgruppens arbeid, men også å kvalitetssikre det arbeidet som blir gjort, samt at det arbeidet er i tråd med retningslinjene som er gitt i emnet "IDRI3001 Bacheloroppgave i drift av datasystemer". Stein vil bidra med kunnskap, rådgiving og veiledning som kan øke sjansene for at prosjektgruppen leverer et tilfredsstillende resultat.

4.1.3 Tabell

Nedenfor oppsummeres suksesskriteriene til hver interessent, og bidraget hver interessent har til prosjektet.

Oppdragsgiver (NHN)

Interessent	Suksesskriterier	Bidrag til prosjektet
Oppdragsgiver	Det gjøres en analyse av containerteknologi som driftsmodell. Det gjøres en kost/nytte analyse av dagens driftsmodell opp mot en container basert driftsmodell. Løsningen vil ivareta Norm for informasjonssikkerhet og personvern i helse og omsorgstjenesten.	Veiledning av dagens system og løsninger. Sette rammer og mål for oppgaven. Bistå med teknisk veiledning gjennom prosjektet. Tilrettelegging av testmiljø ved behov.

NTNU

Interessent	Suksesskriterier	Bidrag til prosjektet
Prosjektgruppen	Levere en tilfredsstillende løsning iht. Rammene og betingelse gitt av veileder og oppdragsgiver Merverdi i form av økt kunnskap.	Ansvarlig for planlegging, implementasjon og dokumentasjon av prosjektet.
Veileder	Prosjektet gjennomføres i henhold til NTNUs Retningslinjer.	Kvalitetssikring og veiledning

4.2 Rammebetingelser

4.2.1 NTNU

Tidsfrister

- Prosjektoppgaven i sin helhet, det vi sil prosjektoppgave, individuelt refleksjonsnotat samt presentasjon skal være levert på Inspera senest 20.05.2019.

Generelle krav

- Det er et krav om at det blir avholdt presentasjon av oppgaven. Dette gjøres på NHN sine lokaler, med videooverføring til NHN sine kontorer i Oslo, Bergen og Tromsø.
- Det skal være signert en prosjektavtale mellom prosjektgruppen, NHN og NTNU.
- Vi skal ha prosjektmøter til faste tider (annenhver torsdag kl. 10:30).
- Skriftlig møtereferat og møteinnkalling fra alle møter.
- Timelister og Ukesrapport skal oppdateres jevnlig og være tilgjengelig for veileder v/NTNU.

4.2.2 NHN

- Prosjektgruppen har skrevet under på taushetserklæring og sikkerhetsregler for NHN og det er et krav at disse underskrevne dokumentene blir etterfulgt.

5. Kritiske suksessfaktorer

5.1 Suksessfaktorer

5.1.1 Suksessfaktorer for prosjektgruppen

Suksessfaktorer beskriver hvilke faktorer som vil være kritisk avgjørende for både oppdragsgiver og systemutvikler i forhold til prosjektets utfall. Vi anser følgende punkter som kritisk avgjørende, og vil dermed prioritere disse for å oppnå et vellykket prosjekt:

- Prosjektgruppen har et godt arbeidsmiljø med god planlegging og effektiv arbeidsfordeling.
- Prosjektgruppen har tilstrekkelig med tid til å innhente kunnskap om Kubernetes, Octopus Deploy samt NHNs nåværende driftsmodell.
- Tilgang til et fungerende testmiljø i Azure og eventuelt et testmiljø hos oppdragsgiver.
- At prosjektgruppen realiserer oppdragsgivers ønsker i form av å levere en løsning som overholder kravene som er gitt.
- At det ikke oppstår kritiske programvarefeil som forhindrer testing av løsning.
- Prosjektet i sin helhet og alle dets deler blir ferdig til endelig frist.

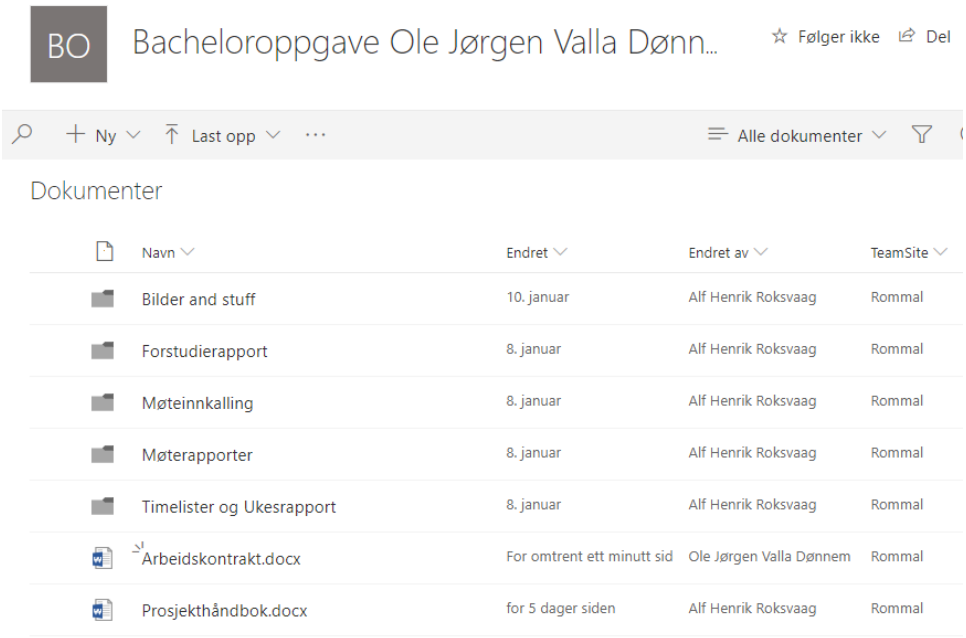
5.1.2 Suksessfaktorer for oppdragsgiver

- Prosjektet resulterer i anvendbar kunnskap for NHN og da spesielt seksjon Drift 2 hos NHN.
- Sette realistiske rammer for prosjektet, og følge med på at disse blir overholdt.

5.2 Informasjonsbehov

Når det kommer til informasjonsformidling så er det viktig å inkludere alle partene som er involvert i prosjektet. Prosjektgruppen må holde oppdragsgivere og veiledere fra NHN og NTNU oppdatert om prosjektets fremdrift. Dette vil skje ved fastsatte møtetidspunkt annen hver uke, og hyppigere når det nærmer seg viktige milepæler i prosjektet eller ved behov.

Alt av prosjektmateriell blir distribuert til oppdragsgiver og veiledere ved hjelp av en SharePoint TeamSite hos NTNU. Prosjektgruppen administrerer denne siden, mens oppdragsgivere og veiledere er medlemmer. Her vil det til enhver tid finnes oppdatert dokumentasjon på møtereferater, møteinnkallelser, fremdriftsplan, timelister etc.



Figur 2: Sharepoint TeamSite

En gang hver andre uke avholdes det statusmøter med oppdragsgiver og veiledere. Her informeres det om fremdrift fra prosjektgruppen, og videre arbeid. Det er også naturlig at disse møtene blir brukt til annen veiledning samt synspunkter på levert arbeid. Det vil også være mulig å øke hyppigheten på disse møtene spesielt rundt viktige milepæler i prosjektet. Senest tre dager før møtedato finner sted skal prosjektgruppen sende oppdatert saksagenda med eventuelt tilhørende dokumenter for gjennomlesning til oppdragsgiver og veiledere. Dette er fordi oppdragsgiver og veiledere skal få tid til å lese gjennom og komme med synspunkter.

Prosjektet har følgende dokumenter:

- Forstudierapport
- Designrapport
- Driftsrapport
- Sluttrapport
- Prosjekthåndbok
- Presentasjonsmateriell
- Individuelt refleksjonsnotat

Det vil også bli gjennomført en muntlig presentasjon på NHN sine lokaler, denne muntlige presentasjonen vil også være tilgjengelig skriftlig.

Nedenfor følger en tabell som viser informasjonsbehovet tilknyttet denne oppgaven.

Oppdragsgiver (NHN)

Hvem som er mottaker	Hva slags type informasjon	Hvor ofte	Hvordan blir informasjon gitt	Hvorfor gjør vi det
Oppdragsgiver	Status på prosjektet	Faste møtetidspunkt annenhver torsdag kl 10:30 på NHN sine lokaler	Møter og dokumentasjon gitt i møteinnkallingene	For å gi jevnlig oppdatering av prosjektets status og fremgang.
Oppdragsgiver	Prosjektrapport	Kontinuerlig	Sharepoint	For å gi en samlet plass med oppdatert informasjon om prosjektets status.
Veileder hos NHN	Status på prosjektet	Faste møtetidspunkt annenhver torsdag kl 10:30 på NHN sine lokaler	Møter og dokumentasjon gitt i møteinnkallingene	For å gi jevnlig oppdatering av prosjektets status og fremgang.

NTNU

Hvem som er mottaker	Hva slags type informasjon	Hvor ofte	Hvordan blir informasjon gitt	Hvorfor gjør vi det
Prosjektgruppen	Veiledningsinformasjon	Faste møtetidspunkt annenhver torsdag kl 10:30 på NHN sine lokaler, ellers kontakt på mail og ikke oppsatte møter	Faste møter, muntlige møter og skriftlig kontakt for eksempel e-post	Kvalitetskontroll, veilede prosjektgruppen.
Veileder hos NTNU	Status på prosjektet	Faste møtetidspunkt annenhver torsdag kl 10:30 på NHN sine lokaler	Møter og dokumentasjon gitt i møteinnkallingene	For å gi jevnlig oppdatering av prosjektets status og fremgang.

Veileder hos NTNU	Prosjektrapport	Kontinuerlig	Sharepoint	For å gi en samlet plass med oppdatert informasjon om prosjektets status.

6. Risikoanalyse

I forstudiet vil det gjennomføres en enkel form for risikoanalyse basert på prosjektgruppens egne tanker og ideer. Vi kommer til å se på uønskede hendelser som kan negativt påvirke prosjektets gang og utførelse, og finne tiltak som kan forebygge risikoen for at disse uønskede hendelsene oppstår. Hver hendelse vil få en tallverdi for sannsynlighet, konsekvens og risikofaktor samt mulige tiltak vil presenteres.

Det vil i designrapporten gjennomføres en risikoanalyse med fokus på risikoer ved overgang til drift med bruk av containerteknologi, Docker og Kubernetes.

6.1 Sannsynlighet

Sannsynligheten for at en uønsket hendelse oppstår får en tallverdi fra 1 til 5, der 1 er minst sannsynlig og 5 er mest sannsynlig. Verdiene defineres på denne måten:

Verdi	Gradering	Definisjon
5	Daglig	Flere enn 3 hendelser i uken
4	Ukentlig	Færre enn 3 hendelser i uken
3	Månedlig	Flere enn 5 hendelser i året
2	Årlig	Færre enn 5 hendelser i året
1	Meget sjeldent	Sjeldnere enn én gang årlig

6.2 Konsekvens

Konsekvensen ved at en uønsket hendelse oppstår får en tallverdi fra 1 til 5, der 1 er minst sannsynlig og 5 er mest sannsynlig. Verdiene defineres på denne måten:

Verdi	Gradering	Definisjon
5	Svært kritisk	Svært kritisk effekt på prosjektets resultat
4	Kritisk	Kritisk effekt på prosjektets resultat
3	Alvorlig	Alvorlig effekt på prosjektets resultat
2	Mindre alvorlig	Mindre alvorlig effekt på prosjektets resultat
1	Ubetydelig	Liten til ingen effekt på prosjektets resultat

6.3 Risikomatrise

En risikomatrise vil gi en visuell fremstilling av risikofaktoren for våre vurderte hendelser. Man kan i figuren under se hvor alvorlig risikoen er basert på et produkt av verdiene sannsynlighet og konsekvens. I grønn sone er akseptansekriteriet ikke overgått og vi har da en akseptabel risiko. Gul indikerer risiko som bør vurderes med hensyn til tiltak som kan redusere risikoen. Rød indikerer uakseptabel risiko, og tiltak må her iverksettes for å forsøke redusere denne.

Konsekvens	Svært kritisk	5	10	15	20	25
	Kritisk	4	8	12	16	20
	Alvorlig	3	6	9	12	15
	Mindre alvorlig	2	4	6	8	10
	Ubetydelig	1	2	3	4	5
		Sjeldent	Årlig	Månedlig	Ukentlig	Daglig
		Sannsynlighet				

Farge		**Beskrivelse**
Rød		Uakseptable risiko, tiltak skal gjennomføres, høy prioritet
Gul		Vurderingsområde, tiltak skal vurderes, middels prioritet
Grønn		Akseptabel risiko, tiltak kan vurderes, lav prioritet

6.4 Prosjektets uønskede hendelser og mulige tiltak

Her vil de uønskede hendelsene som er analysert og mulige tiltak presenteres i en tabell.

- **Hendelse**
  De hendelsene bachelorgruppen har identifisert som uønsket
- **Beskrivelse**
  Beskriver den uønskede hendelsen
- **S**
  Den tallfestede sannsynligheten for at den uønskede hendelsen oppstår
- **K**
  Den tallfestede konsekvensen ved at den uønskede hendelsen oppstår
- **R**
  Risikofaktoren, produktet av sannsynlighet (S) og konsekvens (K) for den uønskede hendelsen
- **Tiltak**
  Mulige forebyggende tiltak for å unngå at den uønskede hendelsen oppstår

Hendelse	Beskrivelse	S	K	R	Tiltak
Langvarig fravær	Langvarig fravær (over 1 uke) ved sykdom etc. kan få alvorlige konsekvenser for bacheloroppgaven	1	4	4	Å sørge for at vi er i rute med fremdriftsplan, og helst være foran skjema
Tap av gruppemedlem	Et eller begge gruppemedlemmene slutter	1	5	5	God kommunikasjon, åpenhet om misnøye og toleranse ved uenigheter
Misforståelser	Misforståelser om hva som skal gjøres, når ting skal gjøres, hvordan det skal gjøres etc.	3	2	6	Ukeplaner, hyppige statusmøter, rutiner for distribuering av kritisk informasjon
Manglende veiledning	Kan føre til feiltolkning av oppgaven, mangel på kritisk informasjon etc.	1	4	4	Hyppige statusmøter, god kommunikasjon mellom bachelorgruppen og veileder. Veileder er tilgjengelig
Manglende kunnskap eller informasjon	Manglende/dårlig dokumentasjon på Kubernetes for Windows eller at bachelorgruppen dårlig behersker teknologien (Docker, Kubernetes, containerteknologi generelt).	3	4	12	Kontinuerlig kompetansebygging. Tilgang til ansatte hos NHN med relevant kompetanse som kan hjelpe og er tilgjengelige for bachelorgruppen
Tap av data	Brukerfeil eller maskin/programvarefeil kan føre til tap av kritiske dokumenter og data for prosjektets gang	1	5	5	Sikkerhetskopiering og rutiner for dette. Lagre alt både lokalt og på SharePoint
Testmiljø	Bachelorgruppen får ikke tilgang til et godt testmiljø fra NHN eller NTNU tidlig nok i prosessen	2	5	10	Ha andre løsninger klare enn den mest ønskede, hvis plan A (Azure fra NTNU) ikke kommer i live tidsnok må det være en plan B (VMWare fra NTNU) or gjerne en plan C (NHN) på plass.

Store skift i bruk av programvare/teknologien	Det skjer en eller flere store endringer i måten programvare eller teknologien fungerer midt i prosjektet som påvirker nytten for tilskaffet kunnskap og kompetanse	2	3	6	Kontinuerlig kompetansebygging innen området containerteknologi. Følge tett med på oppdateringer og nye lanseringer innen dette området. Være tilpasningsdyktige.
Intern uenighet	Uenighet mellom medlemmene i bachelorgruppen eller uenighet mellom bachelorgruppen og NHN/NTNU om tolkning av oppgaven, retningen oppgaven tar etc.	2	3	6	Legge til rette for åpenhet i gruppen. Respekt og toleranse mellom partene. Fokus på å finne en løsning, være åpne for kompromisser.

6.5 Prioritering av tiltak

Her vil det presenteres hvordan uønskede hendelser og implementering av mulige tiltak blir prioritert i form av en tabell. Hendelser med en risikofaktor over verdien 4 blir sett på som alvorlige nok til tiltak bør eller skal implementeres hvis det ikke allerede er gjort.

Hendelse	S	K	R	Prioritet
Manglende kunnskap eller informasjon	3	4	12	Tiltak skal implementeres
Testmiljø	2	5	10	Tiltak skal implementeres
Misforståelser	3	2	6	Tiltak vurderes
Store skift i bruk av teknologien	2	3	6	Tiltak vurderes
Intern uenighet	2	3	6	Tiltak vurderes
Tap av gruppemedlem	1	5	5	Tiltak vurderes
Tap av data	1	5	5	Tiltak vurderes
Manglende veiledning	1	4	4	Tiltak unødvendig
Langvarig fravær	1	4	4	Tiltak unødvendig

Som man kan se i tabellen så er det å ikke ha et godt testmiljø på plass når det trengs og manglende kunnskap/informasjon de største risikoene for suksessen til dette prosjektet/denne bacheloroppgaven. Dette kommer av at den teknologien og programvaren som det utredes for i denne oppgaven er noe både prosjektgruppen og NHN har lite erfaring med og er ganske nytt, spesielt når vi tenker på Windows containere og Windows Server 2019 som har kommet med mye nytt så sent som bare et par måneder siden.

Noen av de mulige tiltakene er allerede blitt implementert eller delvis implementert helt i starten av prosjektet. Disse er:

- Faste og nokså hyppige statusmøter
- Tilgang til kompetanse internt i NHN (Drift 2 teamet og spesielt Nikolai)
- Undersøkt flere forskjellige måter å få tilgang til et testmiljø
- Grundig arbeid med informasjonsinnhenting tidlig i prosjektet.

7. Retningslinjer og standarder

Vi skal i dette kapittelet ta for oss hvordan retningslinjer og standarder prosjekter skal forholde seg til.

7.1 Krav til dokumentasjon

Kravet til dokumentasjonen av bachelor består av syv deler:

- Forstudie
- Designrapport
- Driftsrapport
- Sluttrapport
- Prosjekthåndbok
- Presentasjonsmateriell
- Individuelt refleksjonsnotat

Disse syv delene herunder kalt Prosjektoppgaven skal være levert på Inspera senest 20.05.2019. I tillegg til dette skal det leveres et individuelt refleksjonsnotat, med samme tidsfrist.

Oppgaven blir levert i to deler på Inspera. Den første delen vil inneholde: Forstudie, Designrapport, Driftsrapport, Sluttrapport, Prosjekthåndbok og Individuelt refleksjonsnotat.

Del 2 vil inneholde følgende: skriftlig presentasjonsmateriell, kildekode eller andre vedlegg.

- Presentasjon skal ha filnavn «p» + oppgavenummeret, f.eks. p001.pdf.
- Annet skriftlig materiale skal ha filnavn lik oppgavenummeret, f.eks. 001.pdf, 040.pdf.
- Til så skal refleksjonsnotatet også ha filnavn lik oppgavenummeret, men vi skal i tillegg her bruke studentens navn, for eksempel 001_OlaNordmann.pdf

7.2 Krav til kvalitetsgjennomganger

Alle i prosjektgruppen er med på å kvalitetssikre dokumentene og piloten. Medlemmene i prosjektgruppen skal drive med kvalitetsgjennomganger minst en gang i uken. Det er også ønskelig at veiledere og andre aktører gir tilbakemelding hver andre uke. Dette gjøres i felleskap for at gruppen skal kunne kommunisere, og rette på dokumentet ved behov. Før levering av dokumenter og pilot, så skal det gjennomgås en stor kvalitetskontroll der også styringskomiteen er med og gir tilbakemelding. Det ses på som et krav fra prosjektgruppen at kvalitetskontrollørene

gjennomgår oppgaven i sin helhet minst to ganger i løpet av prosjektperioden. Det ses på som viktig at prosjektgruppen og styringskomiteen har fokus på gjennomgang og tilbakemelding av disse rapportene når milepælene forfaller, spesielt rundt ting som skal leveres og vurderes.

7.3 Krav til standarder og metoder

Bacheloroppgaven vil rette seg etter NTNU IDIs krav for bacheloroppgaver (NTNU 2019a).
Som referansestil vil prosjektgruppen ta i bruk Harvard-stilen. Harvard-stilen er mye brukt innenfor samfunnsfag, teknologi og naturvitenskap. Riktig bruk av Harvard-stilen i tekst og referanselisten vil bli nøye gjennomgått sammen med styringskomiteen og kvalitetskontroll. I tillegg til dette vil oppgaven basere seg på maler som er gitt av NTNU IDI (NTNU 2019b).

7.4 Endringshåndtering

Ved ønske om en endring fra veileder fra NTNU eller NHN, styringskomite eller prosjektgruppen, eller andre interessenter, skal ønsket håndteres på følgende måte:

- Dokumenter endringens innhold.
- Analyser hvilke konsekvenser endringen får for prosjektet.
- Beregne kostnader/nytte av endringen.
- Møte om godkjenning av endringen. Endringer skal godkjennes av veileder fra NTNU, prosjektgruppen og styringskomiteen. Godkjenner ikke noen av disse partene endringen vil ikke endringen bli gjennomført.
- Loggfør endringen.
- Juster planer slik at implementasjon av endringen blir innebygd.
- Informer interessentene. Alle interessenter som er relevante skal informeres.
- Gjennomfør endringen.

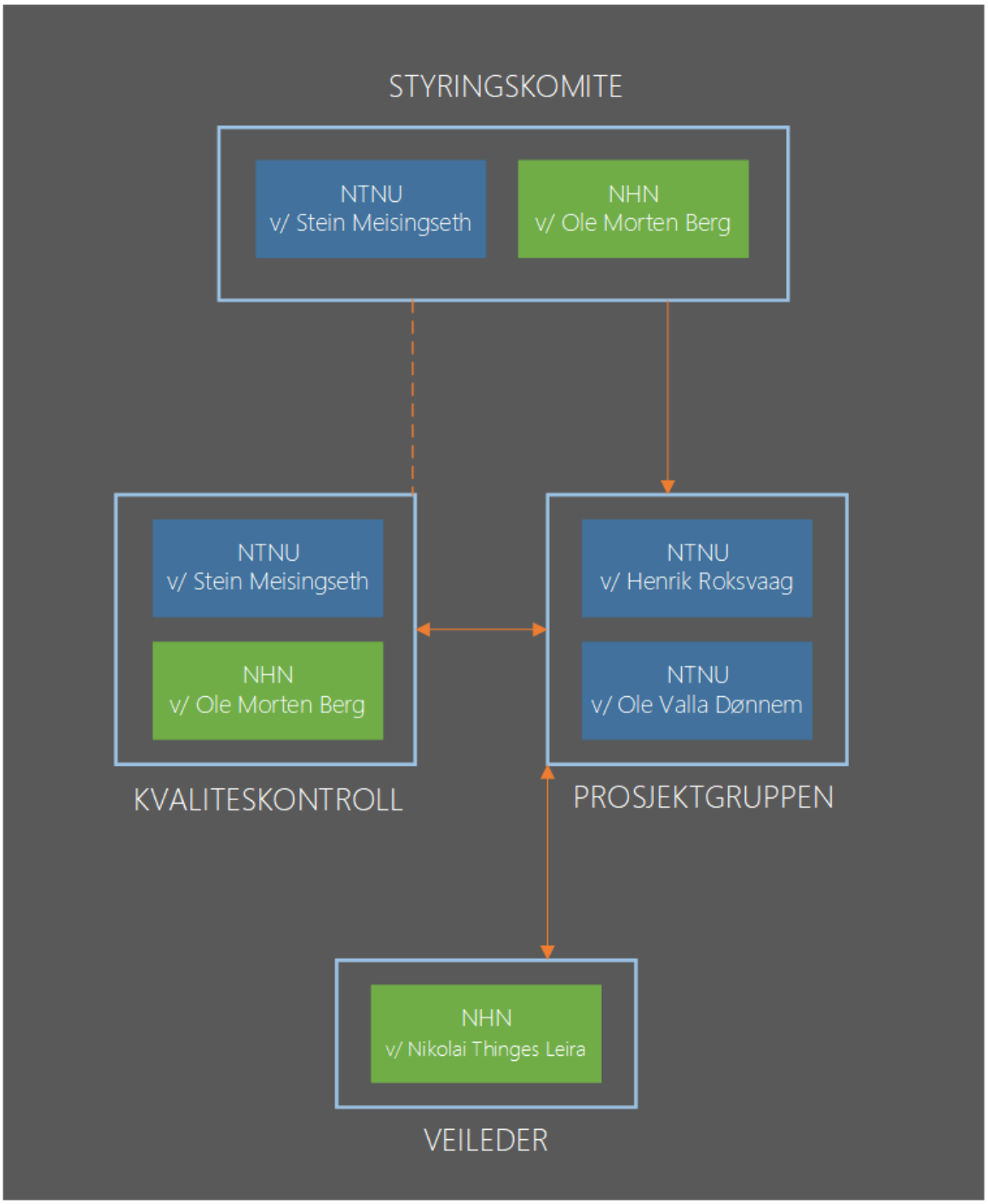
8. Prosjektorganisering

Prosjektorganisering tar for seg hvordan prosjektet er organisert, og hvordan arbeidet er fordelt, se forøvrig (figur 3).

Norsk Helsenett SF er oppdragsgivere i denne bacheloroppgaven. Det er vedtatt en styringskomité som består av Stein Meisingseth fra NTNU Trondheim og Ole Morten Berg som representerer NHN. Stein Meisingseth og Ole Morten Berg vil også stå for kvalitetskontroll i oppgaven. Styringskomitéen har ansvaret for å vurdere om prosjektet tilfredsstiller de kravene som er gitt, og vurdere kvaliteten på arbeidet.

Bachelorprosjektet har ikke en prosjektleder, dette ansvaret blir fordelt mellom prosjektgruppens to medlemmer. Prosjektgruppen kommuniserer direkte med styringskomitéen når det er behov for veiledning eller for å gi tilbakemeldinger vedrørende bachelorprosjektet.

Prosjektgruppen sin jobb er å utarbeide dokumentasjon og en løsning som lever opp til kravene gitt av styringskomitéen.



Figur 3: Prosjektorganisering

9. Anbefaling om videre arbeid

NHN har et ønske om å undersøke mulighetene med å drifte deres applikasjoner på en ny måte. Skulle en slik endring realiseres er det nødvendig å innhente kunnskap om dette er mulig å gjennomføre. NHN har et ønske om å få kartlagt mulighetene og begrensningene i å gå over til en container-basert driftsmodell. Risikoanalysen avdekker ingen store risikoer forbundet med gjennomføringen av prosjektet.

En risikovurdering av sikkerheten knyttet til bruk av løsninger kommer i et senere stadium av prosjektet. Dette skyldes behovet for mer dyptgående kunnskap og praktisk erfaring med produktene og skytjenesten Azure.

Prosjektet er av en utforskende art. Det skal ikke utvikles og implementeres et nytt system,noe som medfører at det er lite kostnader knyttet til selve gjennomføringen av prosjektet.

Prosjektet anbefales videreført med de rammene og planene som er lagt frem i forstudierapporten.

10. Kilder

Amazon (2019): *AWS Total Cost of Ownership (TCO)*
Tilgjengelig fra: https://awstcocalculator.com/
(Hentet 02.02.2019)

Containership (2018): *Kubernetes? Docker? What is the difference?*
Tilgjengelig fra: https://blog.containership.io/k8svsdocker/
(Hentet 16.01.2019)

DatadogHQ (2018)*: 8 surprising facts about Docker adoption*
Tilgjengelig fra: https://www.datadoghq.com/docker-adoption/
(Hentet 31.01.2019)

Direktoratet for e-helse (2018): *Norm for informasjonssikkerhet*
Tilgjengelig fra: https://ehelse.no/personvern-og-informasjonssikkerhet/norm-for-informasjonssikkerhet
(Hentet 31.01.2019)

Dzone (2019): *Kubernetes vs. Docker*
Tilgjengelig fra: https://dzone.com/articles/kubernetes-vs-docker
(Hentet 16.01.2019)

Kubernetes (2019): *Running Kubernetes on Azure*
Tilgjengelig fra: https://kubernetes.io/docs/setup/turnkey/azure/
(Hentet 19.01.2019)

Microsoft (2017): *Introducing AKS (managed Kubernetes) and Azure Container Registry improvements*
Tilgjengelig fra: https://azure.microsoft.com/nb-no/blog/introducing-azure-container-service-aks-managed-kubernetes-and-azure-container-registry-geo-replication/
(Hentet 19.01.2019)

Microsoft (2019a): *Introduction to Azure Kubernetes Service*
Tilgjengelig fra: https://docs.microsoft.com/en-us/azure/aks/intro-kubernetes
(Hentet 19.01.2019)

Microsoft (2019b): *Priskalkulator – Azure*
Tilgjengelig fra: https://azure.microsoft.com/nb-no/pricing/calculator/
(Hentet 02.02.2019)

Mindtools (2018): *Risk Analysis and Risk Management*
Tilgjengelig fra: https://www.mindtools.com/pages/article/newTMC_07.htm
(Hentet 20.02.2019)

Norsk Helsenett SF (2019): *Norsk Helsenett*
Tilgjengelig fra: https://www.nhn.no

(Hentet 15.01.2018)

NRK (2017): *Systemet knelte etter Nytt på Nytt-spøk*
Tilgjengelig fra: https://www.nrk.no/norge/systemet-knelte-etter-nytt-pa-nytt-spok-1.13445055
(Hentet 01.01.2019)

NTNU (2019a): *Bacheloroppgaven våren 2019*
Tilgjengelig fra: http://iie.ntnu.no/fag/hpr/FellesVeiledning2019.html
(Hentet 10.01.2019)

NTNU (2019b): *Maler og standarder - NTNU*
Tilgjengelig fra: http://www.aitel.hist.no/fag/maler-standarder/
(Hentet 10.01.2019)

Project Management Institute (2018): R*isk analysis and management: A vital key to effective project management*
Tilgjengelig fra: https://www.pmi.org/learning/library/risk-analysis-project-management-7070
(Hentet 20.02.2019)

Regjeringen (2016): *Nasjonal strategi for bruk av skytjenester*
Tilgjengelig fra:
https://www.regjeringen.no/contentassets/4e30afec51734d458596e723c0bdea0e/nasjonal_strategi_for_bruk_av_skytenester.pdf
(Hentet 03.03.2019)

Regjeringen (2017): *Digitaliseringsrundskrivet*
Tilgjengelig fra:
https://www.regjeringen.no/no/dokumenter/digitaliseringsrundskrivet/id2569983/
(Hentet 10.01.2018)

Bruk av containere i NHN

Designrapport

Av: Henrik Roksvaag og Ole Valla Dønnem

NTNU
Kunnskap for en bedre verden

Revisjonshistorie

Dato	Versjon	Beskrivelse	Forfattere
22.04.19	1.0	Første utkast	Henrik Roksvaag & Ole Valla Dønnem
26.04.19	1.1	Implementert endringer basert på tilbakemelding fra veiledere	Henrik Roksvaag & Ole Valla Dønnem
03.05.19	1.2	Lagt til kapittel 3. HelseID	Henrik Roksvaag & Ole Valla Dønnem
18.05.19	1.3	Endelig versjon Ferdigstilling basert på endelig tilbakemelding fra veiledere	Henrik Roksvaag & Ole Valla Dønnem

Innholdsfortegnelse - Designrapport

1. Innledning

1.1 Dokumentets hensikt

Hensikten med dette dokumentet er å beskrive valg av løsninger og teknologi som skal tas i bruk i vår oppgave. Vi vil gå i detalj og beskrive alle verktøyene vi vil ta i bruk, samt at vi vil komme med eksempler på hvordan verktøyene fungerer. Vi vil også se på sikkerheten både i vårt Azure miljø og i vårt container-baserte miljø. Dette dokumentet vil fungere som et grunnlag for videre arbeid i vår driftsrapport.

Dette dokumentet vil også inneholde en teoretisk beskrivelse av piloten som skal gjennomføres i driftsdokumentet.

1.2 Oversikt over innholdet

Kapittel 1: Innledning
Inneholder en innledning til dokumentet, oversikt over innhold, samt definisjoner og forkortelser.

Kapittel 2: Bakgrunn og oversikt
Beskriver bakgrunnen for designrapporten, kort oversikt over momentene i planlagt løsning, forutsetninger og avhengigheter.

Kapittel 3: HelseID
I dette kapittelet vil vi kort beskrive applikasjonen HelseID, som vi skal konvertere til å kjøre som containere i vårt testmiljø.

Kapittel 3: Teori om Docker
I dette kapittelet blir det beskrevet hvordan Docker fungerer. Vi går igjennom hva Docker er for noe, og hva det ikke er, samtidig som vi kommer med noen eksempler på hvordan det brukes.

Kapittel 4: Teori om Kubernetes
I dette kapittelet gjør vi et dypdykk i Kubernetes, og forklarer hvordan det fungerer. Vi tar oss for hvordan REST APIet fungerer, hva de ulike delene består av og kommer med en del eksempler på hvordan Kubernetes blir brukt.

Kapittel 5: Teori om Azure
Vi forklarer her hvordan Azure fungerer, og teknologien som taes i bruk. Vi forklarer også hvorfor Azure er en god plattform for containerdrift og hvilke verktøy man har tilgjengelig for å administrere containerdrift i Azure.

Kapittel 6: Teori om monitoreringsverktøy
Vi tar her for oss flere monitoreringsverktøy for å monitorere vårt Kubernetes cluster. Vi diskuterer fordeler og ulemper med verktøyene og forklarer hvem vi ønsker å gå videre med.

Kapittel 7: Design av Pilot
Vi vil under dette kapittelet ta for oss piloten og det som må være på plass for å få den gjennomført. Vi går her over nettverkstopologi, servere og programvare med mer.

Kapittel 8: Kilder
Inneholder kildehenvisninger til de ressursene som er brukt i dette dokumentet.

1.3 Definisjoner og forkortelser

Forkortelse	Begrep	Definisjon
	Metrics	Metrics er numeriske verdier som beskriver noen aspekter av et system på et bestemt tidspunkt. Dette kan være ting som hvor mye cpu, minne, lagring osv. blir brukt til enhver tid. Metrics samles med jevne mellomrom og identifiseres med tidsstempel, navn, verdi og en eller flere definerende etiketter.
API	**Application Programming Interface**	Applikasjoner som deler data med andre applikasjoner.
RDP	**Remote Desktop Protocol**	Fjernstyring av en annen datamaskin.
	Daemon	Et dataprogram som kjøres som en bakgrunnsprosess, i stedet for å være under direkte kontroll av en interaktiv bruker. Utfører en spesifisert operasjon på forhåndsdefinerte tider eller som svar på visse hendelser.

JSON	**JavaScript Object Notation**	Tekstbasert standard for å formatere dokumenter (meldinger) som brukes for datautveksling
YAML	**YAML Ain't Markup Language**	Programmeringsspråk. Ofte brukt for konfigurasjonsfiler.
	Cluster	I Kubernetes består et cluster av minst en master og flere arbeidermaskiner som kalles worker noder. Disse master nodene og worker nodene driver orkestrasjonssystemet til et Kubernetes cluster. Kubernetes objektene som representerer dine containeriserte applikasjoner, kjører alle på toppen av et cluster
	Master node	Håndterer automatisk scheduling av Pods over nodene i clusteret m.m. Del av et cluster.
	Worker node	Er en virtuell maskin eller en fysisk maskin, avhengig av clusteret. Del av et cluster.
	ReplicaSet	Et ReplicaSet har som mål å opprettholde et stabilt sett med replika pods som kjører til enhver tid.
	DaemonSet	DaemonSet sørger for alle (eller noen) av nodene kjører en kopi av en Pod
VM	**Virtuell maskin**	Betegnelsen på en emulering av et gitt operativsystem
VMSS	**Virtual Machine Scale Set**	Ett sett identiske, lastbalanserte VM-er

ARM	**Azure Resource Manager**	En tjeneste som brukes til å levere ressurser i et Azure abonnement. Ressurser det er snakk om kan være ting som VM-er, SQL databaser, lastbalanserere, nettverksgrensesnitt etc.

1.4 Prosjektets øvrige dokumenter

Prosjektet har følgende dokumenter:

- Forstudierapport
- Designrapport
- Driftsrapport
- Sluttrapport
- Prosjekthåndbok
- Presentasjonsmateriell
- Individuelt refleksjonsnotat

2. Bakgrunn og oversikt

Grunnlaget for dette dokumentet, er forstudierapporten (Roksvaag og Dønnem, 2019b) for dette bachelorprosjektet. Der har vi sett på dagens system, problemer, behov, og målene både NHN og prosjektgruppen ønsker å oppnå. Nå skal vi beskrive mulige løsninger for å dekke behovene og oppnå de målene som er satt.

Det miljøet vi har fått tilgang til er et Azure skymiljø via NTNU og flere av løsningene blir preget av dette, spesielt valget av AKS Engine, men vi kommer tilbake til det mer detaljert senere. Et viktig moment i den totale løsningen er valg av plattform, som i denne bacheloroppgaven falt på Azure. Andre viktige momenter er Docker, Kubernetes, AKS Engine og monitoreringsverktøyet Datadog. Vi kommer til å dekke generell teori om disse først før vi detaljert beskriver planlagt løsning. Til slutt vil vi beskrive design av en pilot der vi konverterer en nåværende Windows applikasjon (HelseID), som NHN drifter på tradisjonell måte, til å kunne kjøres som containere og rulle disse ut i et testmiljø.

2.1 Implementert vs planlagt driftsmiljø

Henviser til kapittel “2. Bakgrunn for prosjektet” i forstudierapporten (Roksvaag og Dønnem, 2019b), spesielt “2.2 Dagens system”, for å se beskrivelse av dagens implementerte system. Det planlagte testmiljøet for dette prosjektet blir satt opp i Azure via AKS Engine med 1 Linux master node for Kubernetes cluster, 2 Windows worker noder i Kubernetes clusteret, 2 Linux worker noder i Kubernetes clusteret og 1 Windows VM for nødvendige databaser..

2.2 Forutsetninger og avhengigheter

Det er en forutsetning at de verktøyene vi bruker ikke endres i vesentlig grad for beskrivelsene og løsningene som er skissert. Containerteknologi, og spesielt rundt Windows containere, er i hyppig utvikling og noe som gjelder i dag kan være foreldet i morgen/neste uke/neste måned. Det er for eksempel rykter om private previews av Microsoft som kan gjøre løsningen AKS Engine komplett unødvendig for enkelt oppsett av Kubernetes clusters med Windows workere i Azure. Teori og design beskrevet i denne rapporten kan derfor være mindre praktisk anvendbar og utdatert i forhold til når den ble skrevet.

Avhengigheter som må være på plass for å gjennomføre prosjektet:

- Fortsatt tilgang til Azure Subscription via NTNU.
- Nødvendige rettigheter blir tildelt på nevnte Subscription av NTNU.
- Azure-tjenester er tilgjengelige.

- Personell-ressurser blir frigjort ved behov for informasjon, hjelp og avklaringer av NHN.

3. HelseID

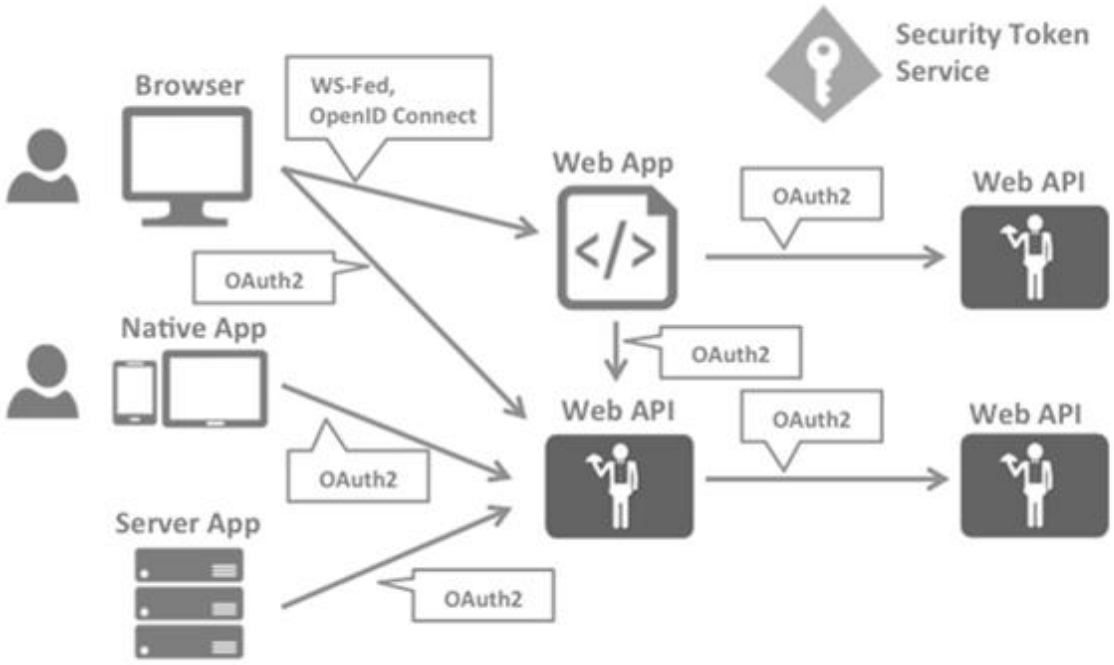
HelseID er en felles påloggingsløsning for helse- og omsorgssektoren. Den legger til rette for at helsepersonell kan få engangspålogging med én elektronisk ID (e-ID) i hele helsetjenesten, og for at sektoren lettere kan dele data og dokumenter (Norsk Helsenett SF, 2019a).

HelseID tilbyr flere ting, pålogging som en tjeneste er det den er kjent for men den tilbyr også autentisering av maskiner og andre enheter med mer. Det er tilrettelagt slik at nasjonale e-helseløsninger og kliniske fagsystemer kan overlate pålogging til HelseID. På denne måten får man en sentralisert løsning slik at helsepersonell får et mindre antall påloggingsmetoder og færre systemer å forholde seg til.

HelseID tar i bruk e-ID-er som allerede eksisterer i sektoren, den lager ikke nye. Det er integrasjon med ID-porten for pålogging med høyt sikkerhetsnivå samt at det er lagt til rette for at helsetjenester kan gjenbruke egne autentiseringsløsninger.

HelseID består av mange komponenter for å få det hele til å fungere, men ikke alle blir fokuset i dette prosjektet. En vi skal spesielt se nærmere på er det Norsk Helsenett selv kaller kjernekomponenten i HelseID, en såkalt STS.

STS står for Security Token Service og hovedoppgaven til en STS er å utstede tokens til applikasjoner og tjenester.
Hensikten med en STS er å outsource sikkerhetsfunksjonalitet til en sentralisert tjeneste, slik at man kan unngå duplisering av funksjonaliteten på tvers av tjenestegrensesnitt, web-, skrivebords- og mobilapplikasjoner. HelseID sin STS behersker protokoller som OpenID Connect 1.0 og OAuth 2.0.



Figur 4: HelseID STS flytdiagram (Norsk Helsenett SF, 2019b)

Modellen viser en forenklet flyt for autentisering og API-tilgang, etter hvilket mønster HelseID STS-en er designet. HelseIDs STS er en RP-STS (Relying Party) hvilket vil si at tjenesten ikke autentiserer klientene selv, men baserer seg på tokens basert på ovennevnte sikkerhetsprotokoller, utstedt fra IP-STS-er (Identity Provider) som er konfigurert som identitetstilbydere i HelseID STS (Norsk Helsenett SF, 2019b).

Da HelseID STS er sett på som kjernekomponenten til HelseID og mye av trafikken til HelseID går gjennom denne vil dette være noe som vi vil konvertere til å kjøre i containere på vårt testmiljø.

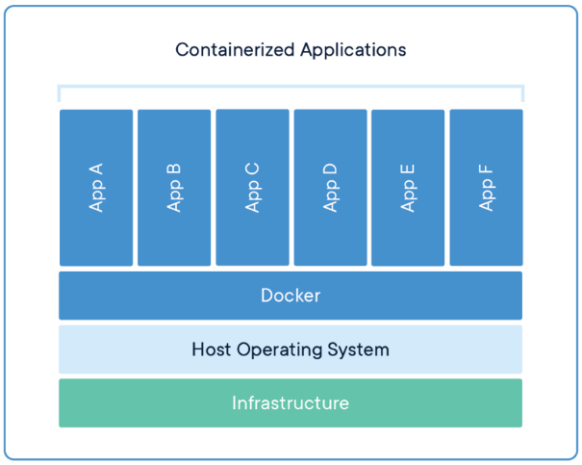
To andre komponenter vi kommer til å konvertere til containere henger tett sammen. Disse to er et web-grensesnitt for administrasjon av HelseID og et API som gjør data tilgjengelig for bruk av dette web-grensesnittet.

4. Teori om Docker

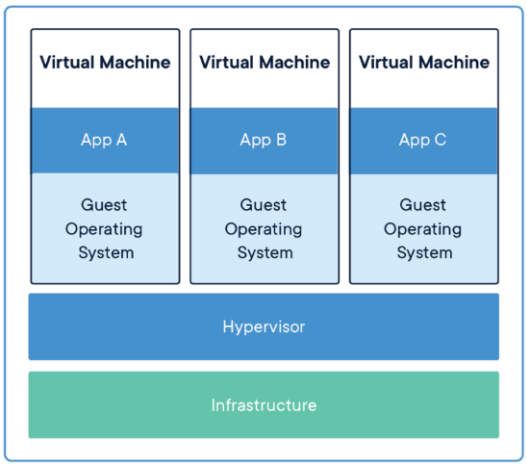
4.1 Hva er Docker

Docker er et verktøy som er designet for å gjøre det enklere å lage, distribuere og kjøre programmer ved hjelp av containere. Med containere kan man pakke sammen et program med alle delene det trenger, for eksempel biblioteker og andre avhengigheter, og sende alt ut som én pakke. På denne måten kan utvikleren av programmet være trygg på at applikasjonen kan kjøre på hvilken som helst annen maskin som bruker operativsystemet containeren er laget for, skal det være Linux, Windows eller Mac, uavhengig av eventuelle tilpassede innstillinger maskinen kan ha som avviker fra maskinen som brukes for å skrive og teste koden. Man kan se på containere som lettvektige virtualiserte maskiner, men i stedet for å virtualisere hardware er det operativsystemet som blir virtualisert. Og det er ikke hele operativsystemet som blir virtualisert, men kun de delene man trenger for å få den spesifikke applikasjonen til å fungere som ønsket. Det er dette som gjør applikasjoner kjørt i containere så lettvektige og mer effektive enn tradisjonelle applikasjoner.

Her er en visuell fremstilling av forskjellen mellom applikasjoner på en container-basert modell og en tradisjonell modell.



Figur 5: Containerized Applications (Docker, 2019b)



Figur 6: Virtual Machines (Docker, 2019b)

Docker er et open source prosjekt og er en av de største grunnene til at containerteknologien tok av som det gjorde for 5-6 år siden. Microsoft har et nært samarbeid med Docker og mye av deres utvikling av støtte for containere på Windows operativsystem er basert på dette samarbeidet.

4.2 Hva Docker ikke er

Når man snakker om containerteknologi så er Docker og Kubernetes de største navnene i dette området. Det kan lett bli vanskelig å skille disse fra hverandre, men det er viktig å forstå at de i grunn jobber på to forskjellige nivåer. Med Docker definerer du din applikasjon og lager et image av den, som så er klart til å bli rullet ut når som helst i den definerte/ønskede tilstanden. Det er Docker som sørger for at dette container image blir kjørt. Etter image er laget og containeren kjører så er jobben til Docker ferdig. Hvis du har flere mikrotjenester som skal kjøre over flere maskiner så må vi vite hvor og når hver container skal kjøres, de må kunne kommunisere med hverandre og det er her Kubernetes kommer inn i bildet. Kubernetes er et orkestreringsverktøy som bruker Docker engine for å kjøre opp containere og administrere dem. Detaljert informasjon om Kubernetes kommer senere i denne rapporten. Docker har også et orkestreringsverktøy kalt Docker Swarm som kan ta seg av noe av dette, men dette er en tilleggstjeneste og ikke en del av Docker i bunn og grunn.

4.3 Hvorfor Docker

Docker er den mest utbredte container engine-en i markedet og som nevnt tidligere har de et nært samarbeid med Microsoft for å forbedre støtten for containere på Windows operativsystem. Dette er en av hovedgrunnene til å bruke Docker i grunn når vi skal undersøke mulighetene rundt Windows applikasjoner og containerisering. Det finnes andre container engine-er som f.eks rkt, men de fleste av disse er

designet for Linux, ikke like gode som Docker og passer derfor ikke så godt for dette prosjektet.
For de som driver med drift av applikasjoner vil Docker gi fleksibilitet og potensielt redusert antall systemer som trengs på grunn av mindre fotavtrykk, bedre effektivitet og lavere overhead via å ta i bruk en container-basert driftsmodell.

Containere tilbyr en logisk pakkemekanisme der applikasjoner kan abstraheres fra det miljø de faktisk driver på. Denne avkoblingen tillater at brukerbaserte applikasjoner distribueres enkelt og konsekvent, uavhengig av om målmiljøet er et privat datasenter, den offentlige skyen, eller til og med en utviklers personlige bærbare datamaskin. Dette gir utviklere muligheten til å lage forutsigbare miljøer som er isolert fra resten av applikasjonene, og kan kjøres hvor som helst. Fra et driftsperspektv får man mer granulær kontroll over ressurser som gir infrastrukturen forbedret effektivitet, noe som kan resultere i bedre utnyttelse av ressurser.

Figur 7: Google trends for søkebegrepet Docker

På grunn av disse fordelene har Docker (og containere generelt) sett omfattende adopsjon. Bedrifter som Google, Facebook og Netflix utnytter containere for å gjøre store ingeniør team mer produktive og forbedre utnyttelsen av tildelte ressurser. Google kreditert faktisk containere for å eliminere behovet for et helt datasenter (Pantheon, 2019).

4.4 Dockers hovedkomponenter

4.4.1 Dockerd

Dockerd er en daemon Docker bruker for å styre Docker containere og håndtere Docker container objekter. Dockerd lytter etter forespørsler sendt via Docker Engine API. Klientprogrammet Docker gi deg et kommandolinjegrensesnitt som lar brukere

kommunisere med Docker deamonen dockerd. Default socket som brukes for å lytte etter forespørsler er en unix domain socket, men man kan endre dette ut ifra hvordan man ønsker å bruke daemonen ved å enable enten tcp eller fd socket. Tcp socket må enables om man ønsker å få tilgang til daemonen remotely for eksempel.

Med versjon 1.2.0 av dockerd er det implementert Kubernetes Container Runtime Interface, slik at dockerd kan brukes direkte av Kubernetes, samt Docker Engine.

Dockerd er et open source container runtime prosjekt som Docker donerte til Cloud Native Computing Foundation (CNCF) i 2017. Dockerd brukes i både Community Edition og Enterprise Edition av Docker.

4.4.2 Docker objekter

Docker objekter er forskjellige enheter som brukes til å bygge en applikasjon i Docker. Det er tre hovedtyper objekter; containere, images, og tjenester.

- **Containere**
  Docker containere er som nevnt tidligere et standardisert, innkapslet miljø som kjører applikasjoner. Docker containere blir behandlet ved hjelp av Docker API eller CLI.
- **Image**
  Et Docker image er en read-only mal som brukes for å bygge containere. Et image blir brukt for å enkelt kunne lagre og sende applikasjoner til riktig sted. Docker image kan lages selv, med bruk av en Dockerfile, eller kan hentes fra offentlige og private repository, mer om dette senere.
- **Tjenester**
  En Docker tjeneste gjør at man kan skalere containere på tvers av flere Docker daemons. Resultatet av dette kalles gjerne en swarm, et sett av daemons som samarbeider og kommuniserer via Docker API.

Containere og image-er har nær sammenheng, da en container blir bygd fra et image. Man kan ikke bygge en container uten å ha et docker image, men man trenger ikke lage et slikt image selv.

4.4.3 Repositories

Et Docker repository er et lager for Docker image-er. Docker klienten kan koble seg til et repository for å laste ned et image, for å bygge containere, eller laste opp et image som du selv har bygget. Slike repository-er kan være offentlige eller private. Vi har i dette prosjektet laget oss et privat repository for alle Docker image-ene vi lager for applikasjonen HelseID slik at det kun er prosjektgruppen som har tilgang til disse.

To store offentlige repository-er er Docker Hub og Docker Cloud, her kan man finne tusenvis av offentlige image-er man kan benytte seg av. Noen av de mest populære image-ene som kan finnes på Docker Hub er Microsofts offisielle Windows Server Core image, Nginx, MySQL, RabbitMQ osv. Med disse offentlige image-ene slipper man en del av arbeidet med å lage et base image som applikasjoner blir bygd på. Ønsker man en MySQL-database som Docker container kan man bare bygge og kjøre den containeren basert på et offentlige image på Docker Hub.

4.5 Docker verktøy

4.5.1 Docker Compose

Med verktøyet Docker Compose kan man definere og kjøre multi-container Docker applikasjoner. Verktøyet bruker YAML-filer til å konfigurere applikasjonens tjenester. Når YAML-filen er laget kan du, med en enkelt kommando, lage og starte alle tjenestene fra konfigurasjonen. Docker Compose CLI-en lar brukere kjøre kommandoer på flere containere samtidig, for eksempel å bygge et image, skalere containere, starte opp nye/stoppe containere og mer.

Bruk av Docker Compose består typisk av tre steg:

1. Definer applikasjonens miljø med en Dockerfile, slik at den kan reproduseres uten ekstra arbeid.
2. Definer tjenestene som utgjør applikasjonen i en docker-compose.yml fil slik at tjenestene kan kjøre sammen i et isolert miljø.
3. Kjør “docker-compose up”. Docker Compose starter da opp og kjører hele applikasjonen.

På denne måten unngår man en del ekstra arbeid hvis man har en applikasjon som trenger å kjøre flere containere på en enkelt host, man slipper å kjøre “docker run …" kommandoer hver for seg. Dette er spesielt nyttig i et testmiljø der du gjerne hyppig starter/stopper containerne og oppdaterer kildekode som containerne skal kjøre.

4.5.2 Docker Swarm

Docker Swam er Dockers native container orkestreringsverktøy og er innebygd i nyere Docker Engine versjoner. På lik linje med Kubernetes er det til for å kjøre, behandle og koble sammen containere på flere hoster. Det administrerer containere som kjører på flere verter og utfører ting som skalering, oppstart av en ny container når en krasjer, nettverk osv. Docker Swam gjør en gruppe Docker Engines spredt over flere hoster til en enkelt virtuell Docker Engine for å kunne utføre disse oppgavene.

Docker swam CLI-verktøyet lar brukere kjøre Swarm containere, lage tokens, liste ut noder i clusteret med mer.

Docker node CLI-verktøyet lar brukere administrere noder i clusteret, for eksempel oppdatering av noder, sletting av noder fra clusteret med mer.
Docker Swarm bruker Raft Consensus Algorithm for administrering, kort sagt så må flertallet av noder i clusteret være enige før noe kan oppdateres/endres.

4.6 Docker i praksis

For å vise prosessen når man skal bruke Docker i praksis går vi gjennom hva som må gjøres for å få en enkel Windows IIS applikasjon til å kjøre i en Docker container og hvordan de komponentene som vi nevnte tidligere ser ut i et slikt tilfelle.

4.6.1 Bygge Docker image– Steg 1

Det første man trenger er et Docker image containeren kan bygges fra. Det finnes ferdiglagde offisielle Windows IIS image-er på offentlige repository som Docker Hub, men siden dette er et eksempel vil vi vise hvordan et image kan bygges fra grunnen av. Man starter med å lage en Dockerfile med dette innholdet for å bygge et image som har Web-Server feature installert på en Windows server:

```
FROM microsoft/windowsservercore:1803
RUN powershell -Command `
    Add-WindowsFeature Web-Server; `
    Invoke-WebRequest -UseBasicParsing -Uri `
    "https://dotnetbinaries.blob.core.windows.net/servicemonitor/ `
      2.0.1.6/ServiceMonitor.exe" `
      -OutFile "C:\ServiceMonitor.exe"
EXPOSE 80
ENTRYPOINT ["C:\\ServiceMonitor.exe", "w3svc"]
```

Alle Dockerfiles starter med FROM etterfulgt av hvilket image man skal bruke som operativsystem i grunn. I dette tilfelle skal det kjøre opp en enkel IIS applikasjon så det velges Windows Server Core versjon 1803.
Deretter er det en RUN linje som sier det skal kjøre to kommandoer i Powershell for å legge til Web-Server feature da dette må være på plass for IIS-applikasjonen.
Port 80 eksponeres med EXPOSE 80.

ENTRYPOINT sier hvilken kommando og parametere som skal kjøres først når containeren er oppe. I dette tilfellet skal tjenesten w3svc starte for at IIS skal fungere som forventet og blir monitorert.

Nå som man har en Dockerfile for Windows server image med IIS installert kan det bygges på dette image slik at det kan brukes det for å kjøre en IIS-applikasjon. For å bygge et docker image fra en Dockerfile bruker man kommandoen under, der <sti> er stien til den Dockerfile som nettopp ble lagd og man bruker argumentet "-t" for å gi image-et et navn.

```
docker build -t IIS-server <sti>
```

Nå som et image er laget, kan det lastes opp til et offentlig eller privat repository slik at hvis noen andre har bruk for dette image kan de enkelt få tak i det. Opplastning av et image gjøres med en docker push kommando og henting av et image gjøres med en docker pull kommando.

4.6.2 Bygge Docker image– Steg 2

Man kan nå bruke det image-et som nettopp ble laget til å lage et nytt image spesifikt for en IIS-applikasjon og man må da igjen lage en ny Dockerfile for denne applikasjonen. Den har flere av de samme elementene som den Dockerfile-en som nettopp ble laget med noen forskjeller. Den kan se slik ut:

```
FROM IIS-server
RUN powershell -NoProfile -Command Remove-Item -Recurse C:\inetpub\wwwroot\*
WORKDIR /inetpub/wwwroot
COPY iis-applikasjon/ .
```

Her ser vi at det igjen brukes FROM for å si hvilket image man skal bygge fra, og i dette tilfellet blir det valgt image kalt 'IIS-Server' som ble laget tidligere. Med RUN ser vi at det igjen kjører en kommando i PowerShell. Denne gangen slettes innholdet i mappen C:\inetpub\wwwroot slik at den er tom og klar til å motta innholdet for en IIS-applikasjon. Videre brukes WORKDIR for å si hvilken mappe som det videre skal forholde seg til. Til slutt brukes COPY for å kopiere innholdet i IIS-applikasjonen til stående mappe som da er /inetpub/wwwroot. Siden det allerede er eksponert port 80 i image som dette nye image bygges fra og lagt inn Entrypoint der, trenger man ikke gjøre det igjen i denne Dockerfile-en.

Og igjen brukers build kommandoen for å bygge et nytt image, i dette eksempelet kalles det 'iis-applikasjon':

```
docker build -t iis-applikasjon <sti>
```

Det samme som med det første image-ett gjelder her, man kan nå laste opp dette opp til et repository slik at andre kan hente det og kjøre ut sin egen IIS-applikasjon med samme innholdet som dette har hvor og når som helst uten å måtte ha tak i noe som helst av konfigurasjon, kildekode eller lignende.

4.6.3 Bruke Docker image til å lage og kjøre container – Steg 3

Nå som det er laget et docker image for en IIS-applikasjon kan man bruke dette til å lage og kjøre det ut i en container. Det meste av arbeidet er allerede gjort, så nå trengs det kun å bruke en docker run kommando for å kjøre den ut i en container. Den kommandoen for en slik simpel IIS-applikasjon er ganske enkel siden det meste allerede er gjort alt i Dockerfilen. Kommandoen kan se slik ut:

```
docker run -d -p 8000:80 iis-applikasjon -name kjorende-iis-applikasjon
```

Argumentet "-d" sier at containeren skal kjøres i 'detached mode', det vil si at den kjører i bakgrunnen slik at du kan fortsette å bruke terminalen til andre ting og du slipper å åpne en ny terminal osv. Du kan finne den kjørende containeren med kommando 'docker ps' og bruke kommando 'docker exec' for å kjøre kommandoer på containeren når den er i detached mode.
Argumentet "-p" sier hvilke porter containeren skal eksponeres på, som i dette tilfellet er 8000:80 slik at container port 80 blir eksponert som host port 8000.
Argumentet "--name" er nokså selvforklarende, det sier bare hva navnet på containeren skal være.
Og til slutt i kommandoen sier man hvilket image som containeren skal bygges fra og i dette eksempelet er det derfor valgt 'iis-applikasjon' som er det siste image som ble laget for denne IIS-applikasjonen.

Nå som IIS-applikasjonen kjører kan man finne IP-adressen til containeren med å bruke kommandoen

```
docker inspect - f "{{ .NetworkSettings.Networks.nat.IPAddress }}" kjorende-iis-applikasjon
```

Man kan deretter kommer seg på nettsiden med IP adressen som finnes og den konfigurerte porten, for eksempel http://172.30.30.30:8000.

Og så enkelt kan det være. Dette er jo et nokså enkelt eksempel og jo større og mer kompleks applikasjonen er, jo vanskeligere er det å konvertere en applikasjon som ikke er utviklet for å kjøre på containere slik som HelseID. Men konseptene gjelder hele veien gjennom.

5. Teori om Kubernetes

5.1 Hva er kubernetes

Kubernetes (ofte kalt K8s) er en orkestreringsmotor for containerverktøy som for eksempel Docker og Rkt, som spesielt innenfor DevOps miljøer har vært i vinden siste par årene. Kubernetes brukes for automatisering av applikasjonsutrulling, skalering og styringskontroll. Kubernetes ble opprinnelig designet av Google og vedlikeholdes nå av Cloud Native Computing Foundation. Kubernetes er tilgjengelig i Microsoft Azure, Amazon Web Services og Google Cloud som en administrert tjeneste.

Kubernetes gir et container-sentrert styringsmiljø. Den orkestrerer databehandling, nettverk og lagringsinfrastruktur på vegne av brukerens arbeidsbelastning. Dette gir mye av enkelheten til plattform som en tjeneste (PaaS) med fleksibiliteten til infrastruktur som en tjeneste (IaaS), og muliggjør overførbarhet på tvers av infrastrukturleverandører.

Vi sier ofte at Docker er en container runtime: den gir funksjoner for pakking, frakt og kjøring av enkelt forekomster av en applikasjon på en standardisert måte, også kjent som en container. Men ettersom kompleksiteten øker opptrer nye behov; automatisert distribusjon, orkestrering av containere, høy tilgjengelighet, administrere et cluster av flere applikasjon forekomster, og så videre.

Kubernetes, kort sagt, er et system for orkestrering av containeriserte applikasjoner over et cluster av noder, inkludert nettverks- og lagringsinfrastruktur. Noen av de viktigste funksjonene er:

- Ressursplanlegging: Sikrer at Pods distribueres optimalt over alle tilgjengelige nodene.
- Automatisk skalering: Med økende belastning kan clusteret dynamisk tildele flere noder, og distribuere nye Pods på dem.
- Self healing: Clusteret overvåker containere og starter på nytt om nødvendig, basert på policy-ene man har satt.
- Service-discovery: Pods and Services er registrert og publisert via DNS .
- Rolling updates: Støtter rullende oppdateringer basert på sekvensiell omplassering av Pods og containere.
- Secrets: Støtter sikker håndtering av sensitive data som passord eller API-nøkler.
- Lagringorkestrasjon: Flere tredjeparts lagringsløsninger støttes, som kan brukes som eksterne volumer for å vedvare data.

5.2 Hva Kubernetes ikke er

Kubernetes er ikke et tradisjonelt, altomfattende PaaS-system (Platform as a Service). Siden Kubernetes opererer på containernivå i stedet for på maskinvarenivå, gir det noen vanlige anvendelige funksjoner som er felles for PaaS-tilbud, for eksempel distribusjon, skalering, lastbalansering, logging og overvåking.

I tillegg er Kubernetes ikke bare et orkestreringssystem. Faktisk eliminerer det behovet for orkestrasjon. Den tekniske definisjonen av orkestrering er utførelse av en definert arbeidsflyt: Først gjør A, deretter B, deretter C. Kubernetes består derimot av et sett med uavhengige, komposittе kontrollprosesser som kontinuerlig kjører nåværende tilstand mot ønsket tilstand. Det burde ikke være noe som betyr hvordan du kommer fra A til C. Det er ikke nødvendig med sentralisert kontroll. Dette resulterer i et system som er lettere å bruke og kraftigere, robustere, elastisk og utvidbart.

5.3 Hvorfor Kubernetes

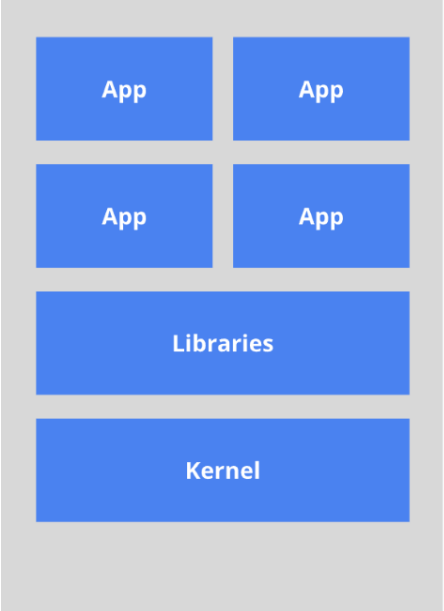
Kubernetes kan minske tiden man bruker på utviklingsprosessen ved å lage enkle, automatiske distribusjoner, oppdateringer (rullende oppdatering) og ved å administrere våre applikasjoner og tjenester med nesten null nedetid. Kubernetes har også innebygget self healing. Det vil si at Kubernetes kan oppdage og starte opp tjenester når en prosess kræsjer inne i containeren. I instruksjonene til kubernetes forteller man blant annet følgende:

- Hvilke tjenester man ønsker å kjøre.
- Hvor mye ressurser hver tjeneste maksimalt kan benytte.
- Hvor den skal hente og lagre sine variable data.
- Hvor mye last systemet skal kunne håndtere.
- Hva systemet skal gjøre dersom lasten blir for høy.

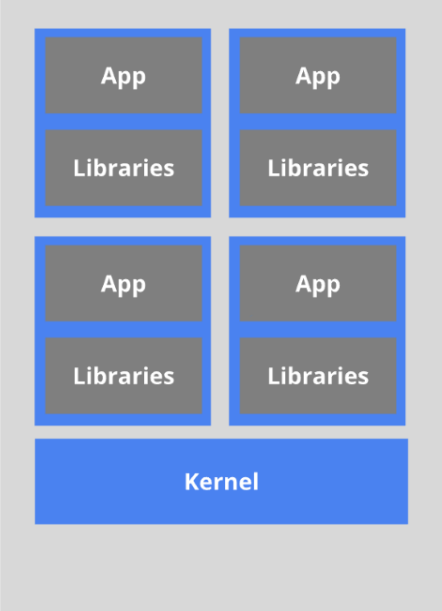
Basert på hvor mye systemet blir belastet eller hvor mange samtidige brukere som er tilkoblet kan kubernetes automatisk skalere tjenesten til å håndtere mer last. Dette gjøres ved å kjøre hver komponent av løsningen som en egen mikro-tjeneste. Dermed kan man fortelle hvor mange av hver mikrotjeneste som skal til for å håndtere en viss last. Dersom lasten eskalerer, kan systemet automatisk kompensere for dette ved å starte opp flere mikrotjenester som jobber sammen for å ta lasten.

Den gamle måten å distribuere programmer på, er å installere programmene på en host ved hjelp av OSet sin egen package manager. Dette har ulempen med å koble applikasjonens kjørbare filer sammen med konfigurasjon, biblioteker med hverandre og med verts-OSet.

Den nye måten er å distribuere containere basert på virtualisering på operativsystemnivå i stedet for maskinvarevirtualisering. Containerne er isolert fra hverandre, og fra hosten. Containere har egne filsystemer, de kan ikke se hverandres prosesser, og deres ressursbruk kan begrenses. Siden de er koblet fra den underliggende infrastrukturen og fra vertsfilsystemet, er de meget portable over både sky og OS-distribusjoner.



Figur 8: Gammel måte: Lite portabel, er avhengig av OS (Kubernetes, 2019c)



Figur 9: Nye måten: Små og portable, bruker OS-level virtualisering sin package manager (Kubernetes, 2019c)

Dette en-til-ett-forholdet mellom applikasjoner og image frigjør de fulle fordelene med containere. Generering av container image-er ved build / release gjør at et konsistent miljø kan overføres fra utvikling til produksjon. På samme måte er containere langt mer gjennomsiktige enn en VM, noe som letter monitorering og styring.

Sammendrag av container fordeler:

- Agil applikasjonsopprettelse og distribusjon: Økt brukervennlighet og effektivitet i skapelse av container image-er i forhold til VM-image-er.
- Miljømessig konsistens på tvers av utvikling, testing og produksjon: Kjører det samme på en bærbar datamaskin som den gjør i skyen.
- Løst koblet, distribuert, elastisk, frigjort mikrotjeneste: Programmene brytes inn i mindre, uavhengige pakker og kan distribueres og styres dynamisk - ikke en monolittisk stabel som kjører på en stor engangs maskin.
- Ressursisolering: Forutsigbar applikasjonsytelse.

- Ressursutnyttelse: Høy effektivitet og tetthet.

5.4 Kubernetes hovedkomponenter

Hovedkomponenter utfører globale beslutninger om clusteret, oppdager og reagerer på cluster hendelser. Som et eksempel starter den opp poder hvis antall replicas ikke er oppnådd på en deployment.

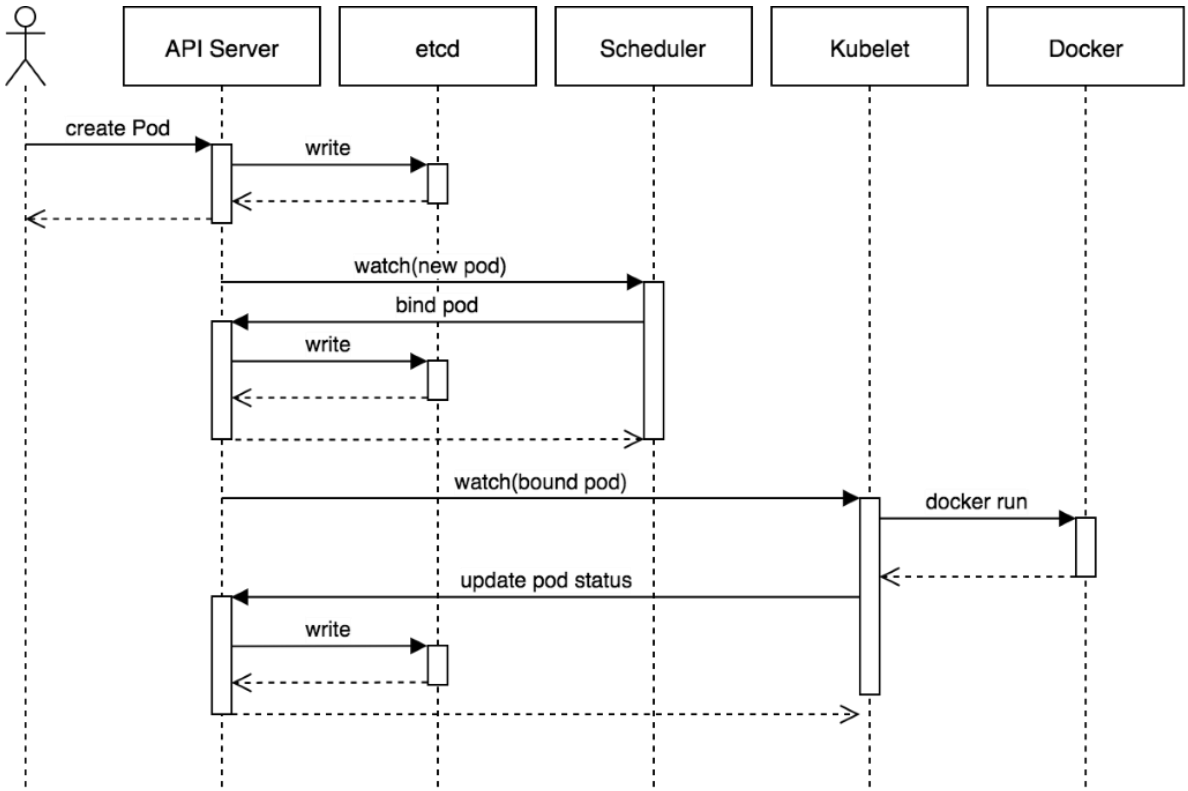
5.4.1 kube-apiserver

Kube-apiserver er en master komponentet som eksponerer Kubernetes sitt API. Det er frontenden til Kubernetes kontrollplan. Når du snakker med Kubernetes clusteret ditt ved hjelp av kubectl, kommuniserer du faktisk med master API Server-komponenten.

API-serveren er hovedpunktet til hele clusteret. Kort sagt behandler det REST-operasjoner, validerer dem og oppdaterer de tilsvarende objektene i etcd. API-serveren serverer API-en for Kubernetes og er ment å være en relativt enkel server, med de fleste forretningslogikkene implementert i separate komponenter eller i plugins.

API-serveren er også ansvarlig for autentisering. Alle API-klienter bør autentiseres for å kunne kommunisere med API-serveren.

Under følger et eksempel på hva som skjer når du oppretter en pod med Kubectl:



Figur 10: Opprette en pod med kubectl (Acetozi, J., 2019)

Her er det som skjer forklart i detalj:

- kubectl skriver til API-serveren.
- API-serveren validerer forespørselen og sender den videre til etcd.
- etcd validerer og gir beskjed tilbake til API-serveren.
- API-serveren påkaller Scheduler.
- Scheduler bestemmer hvor poden skal kjøres og returnerer svaret til API-serveren.
- API-serveren sender forespørsel videre til etcd.
- etcd validerer og gir beskjed tilbake til API-serveren
- API-serveren påkaller Kubelet i tilhørende cluster.
- Kubelet snakker til Docker-daemonen ved hjelp av API for å lage containeren.
- Kubelet oppdaterer podstatusen til API-serveren.
- API Server sender data om ny status til etcd.
- etcd validerer og gir svar tilbaek til API-serveren.
- API-serveren gir beskjed til Kubelet.

5.4.2 kube-controller-manager

Er en komponent på masteren som kjører kontrollere.
Logisk er hver kontroller en egen prosess, men for å redusere kompleksiteten blir de alle samlet inn i en enkelt binær og kjøres i en enkelt prosess.

Disse kontrollerene inkluderer:

- Node Controller: Ansvarlig for å merke og svare når noder går ned.
- Replication Controller: Ansvarlig for å opprettholde det riktige antallet pods for hvert replikasjonsstyringsobjekt i systemet.
- Endpoints Controller: Populerer Endpoints-objektet (det vil si, blir tildelt tjenester og pods).
- Service Account & Token Controllers: Oppretter standardkontoer og API-tilgangstokens for nye namespaces.

I tillegg til dette utfører Controller Manager oppretting av namespaces, innsamling av hendelser osv.

5.4.3 kube-scheduler

Scheduleren ser på unscheduled pods og binder dem til noder via API. Når poden har en node tildelt, utløses den vanlige oppførselen til Kubelet, og poden og dens containere er opprettet.

5.5 Resterende komponenter

Kubernetes har mange moduler, som gjør opp Kubernetes motoren, vi vil under dette punktet ta for oss disse.

5.5.1 REST API

Kubernetes API er et ressursbasert (RESTful) programmatisk grensesnitt levert via HTTP. Den støtter å hente, opprette, oppdatere og slette primære ressurser via standard HTTP-verbene (POST, PUT, PATCH, DELETE, GET). Det inneholder flere underressurser for mange objekter som tillater finkornet autorisasjon (for eksempel binde en pod til en container).

De fleste Kubernetes API ressurs typer er "objekter" - de representerer en konkret forekomst av et konsept på clusteret, som en pod eller namespace. Et mindre antall API-ressurstyper er "virtuelle" - de representerer ofte operasjoner i stedet for objekter. Alle objekter vil ha et unikt navn som tillater opprettelse og gjenfinning.

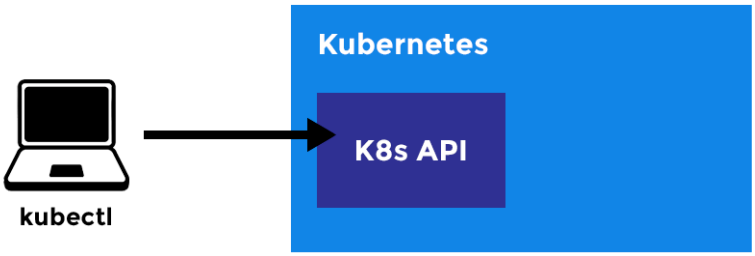
- En resource type er navnet som brukes i nettadressen (pods, namespaces, services).
- Alle resource types har en konkret representasjon i JSON.
- En liste over forekomster av en ressurstype er kjent som en collection.
- En enkelt forekomst av ressurstypen kalles en resource.

Alle ressurstyper er enten scoped av clusteret **(/apis/GROUP/VERSION/*)** eller til et namespace **(/apis/GROUP/VERSION/namespaces/NAMESPACE/ *)**. Et namespace resource vil bli slettet når et namespace er slettet, og tilgangen til ressurstypen styres av autorisasjonskontroller på namespace scopet.

Alle operasjoner og kommunikasjoner mellom komponenter og eksterne brukerkommandoer er REST API-kall som API-serveren håndterer. Derfor blir alt i Kubernetes-plattformen behandlet som et API-objekt og har en tilsvarende oppføring i API. De fleste operasjoner kan utføres via kommandolinjegrensesnittet kubectl eller andre kommandolinjeverktøy, for eksempel kubeadm, som i sin tur bruker API. Du kan imidlertid også få tilgang til API direkte ved hjelp av REST kall.

5.5.2 Kubectl

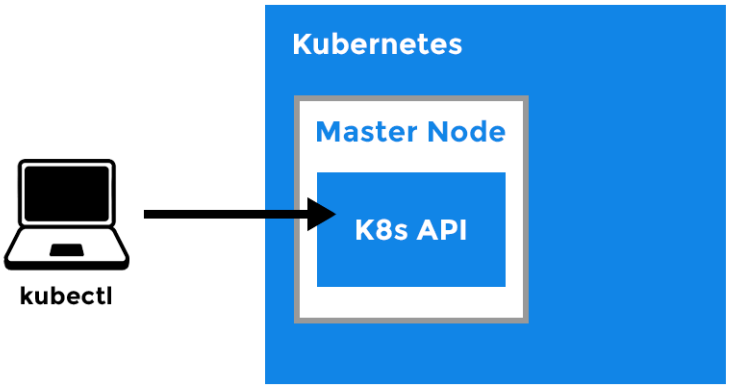
Kubectl er et kommandolinjegrensesnitt for å kjøre kommandoer mot Kubernetes clustere. For å jobbe med Kubernetes bruker du Kubernetes sitt REST API for å beskrive clusteret sin ønskede tilstand. Det vil si at du definerer hvilke programmer eller andre arbeidsbelastninger du vil kjøre, hvilke container images de bruker, antall replicas, hvilke nettverks- og diskressurser du vil gjøre tilgjengelig, osv. Du angir ønsket status ved å lage objekter ved hjelp av Kubernetes REST API, vanligvis via kommandolinjegrensesnittet, kubectl.



Figur 11: Kubectl (Ivancza, K., 2018)

5.5.3 Master node

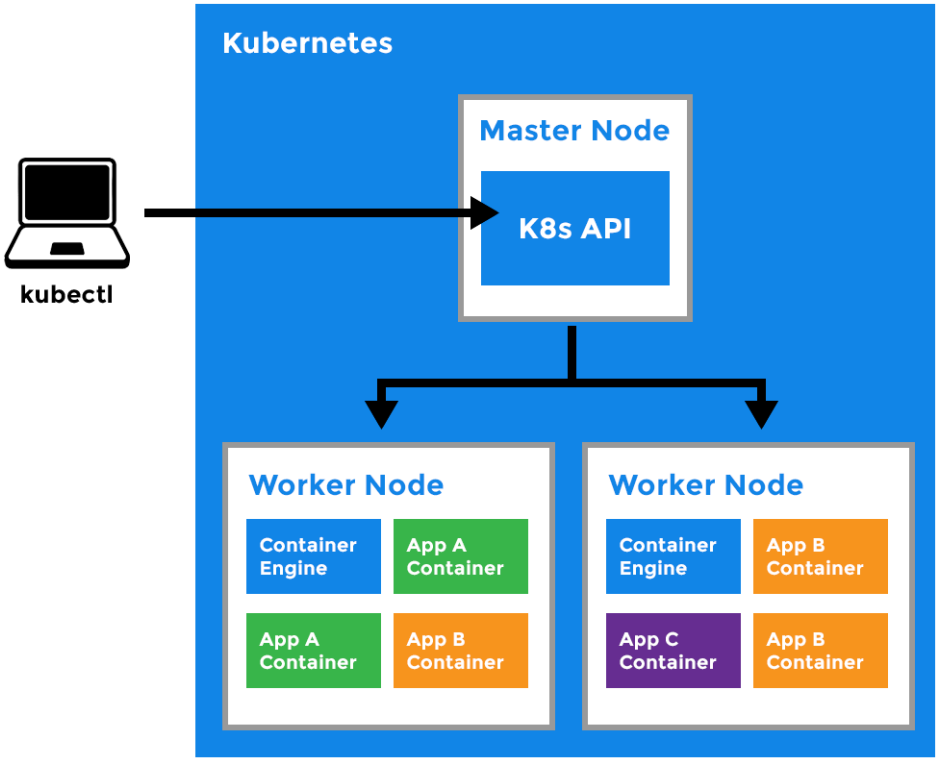
Hovedmaskinen som styrer nodene i clusteret. Hovedpunkt for alle administrative oppgaver. Den håndterer orkestrering av worker nodene.



Figur 12: Master node (Ivancza, K., 2018)

5.5.4 Worker node

Worker noden utfører de forespurte oppgavene. Hver worker node styres av master noden. Worker noden kjører containere inne pods. Det er inni worker noden at docker kjører, og tar for seg nedlastning av images og starting av containere.

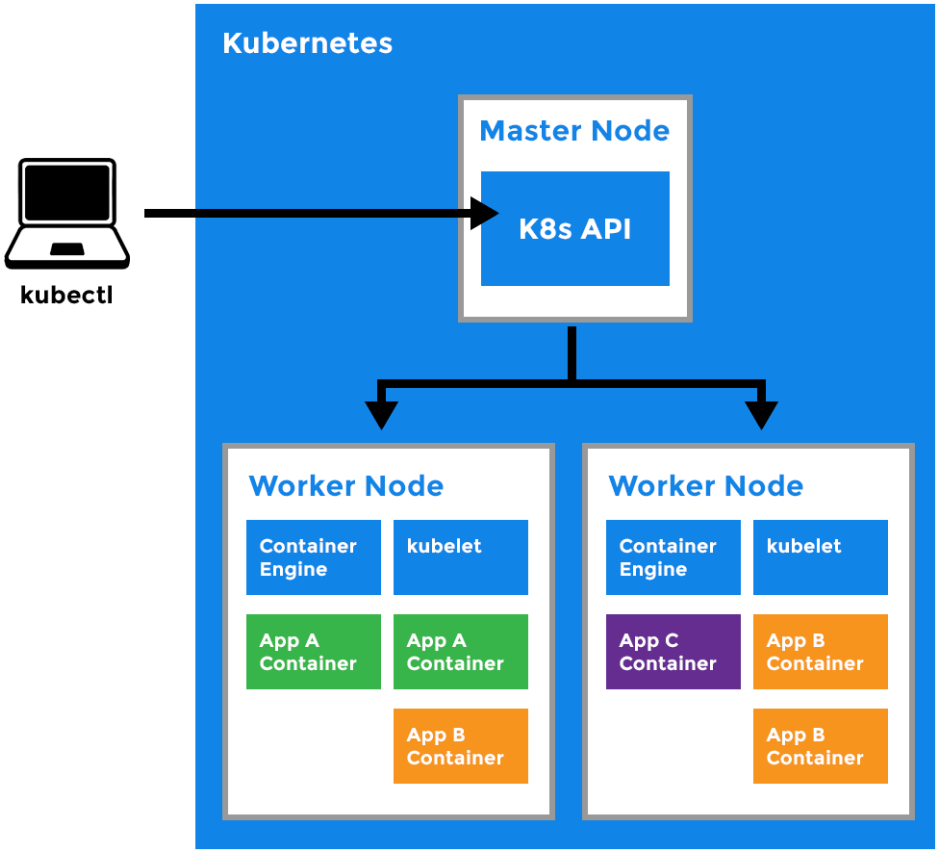


Figur 13: Worker node (Ivancza, K., 2018)

5.5.5 Kubelet

Kubelet er den primære "node agenten" som kjører på hver node. Kubelet fungerer i form av en PodSpec. En PodSpec er et YAML- eller JSON-objekt som beskriver en pod. Kubelet tar et sett med PodSpecs som tilbys gjennom ulike mekanismer (primært gjennom api-server) og sikrer at containerne beskrevet i disse PodSpecs kjører og er healthy. Kubelet administrerer ikke containere som ikke ble opprettet av

Kubernetes.



Figur 14: Kubelet (Ivancza, K., 2018)

5.5.6 Kubernetes Pod

En Pod er en gruppe med en eller flere containere (for eksempel Docker-containere), med delt lagring og nettverk. En Pod er en spesifikasjon for hvordan man kjører containerne i et cluster. Pod'ens innhold er alltid samlokalisert og samordnet, og kjører i en felles sammenheng.

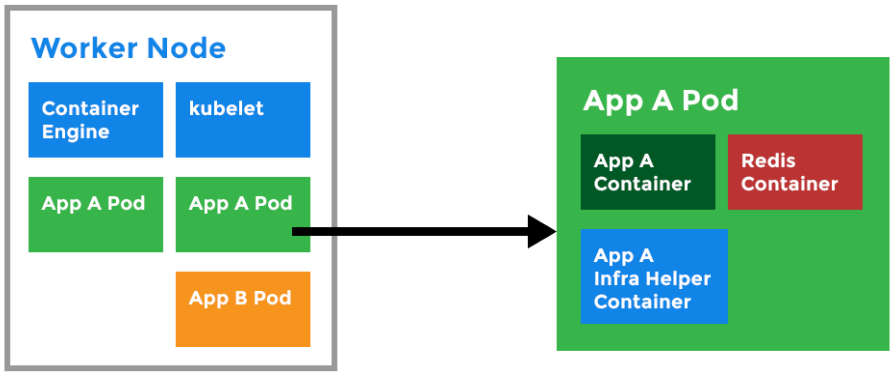
Containere i en Pod deler IP-adresse og port, og kan finne hverandre via localhost. De kan også kommunisere med hverandre ved hjelp av standard inter-prosesskommunikasjon (IPC) som SystemV eller POSIX-delt minne. Containere i forskjellige Pods har forskjellige IP-adresser og kan ikke kommunisere med IPC uten spesiell konfigurasjon. Disse containerne kommuniserer vanligvis med hverandre via pod IP-adresser.

Oppsummert kan vi si at

- En Pod kan være vert for flere containere og lagringsvolumer.
- Pods er forekomster av deployments.
- En deployment kan ha flere Pods.
- Med Horisontal Pod Autoscaling kan pods automatisk startes og stoppes

basert på CPU-bruk.

- Containere innenfor samme pod har tilgang til delte volumer.
- Hver Pod har sin unike IP-adresse i clusteret.
- Pods er oppe og kjører helt til noen sletter de.
- Eventuelle data lagret inne i Pod vil forsvinne.

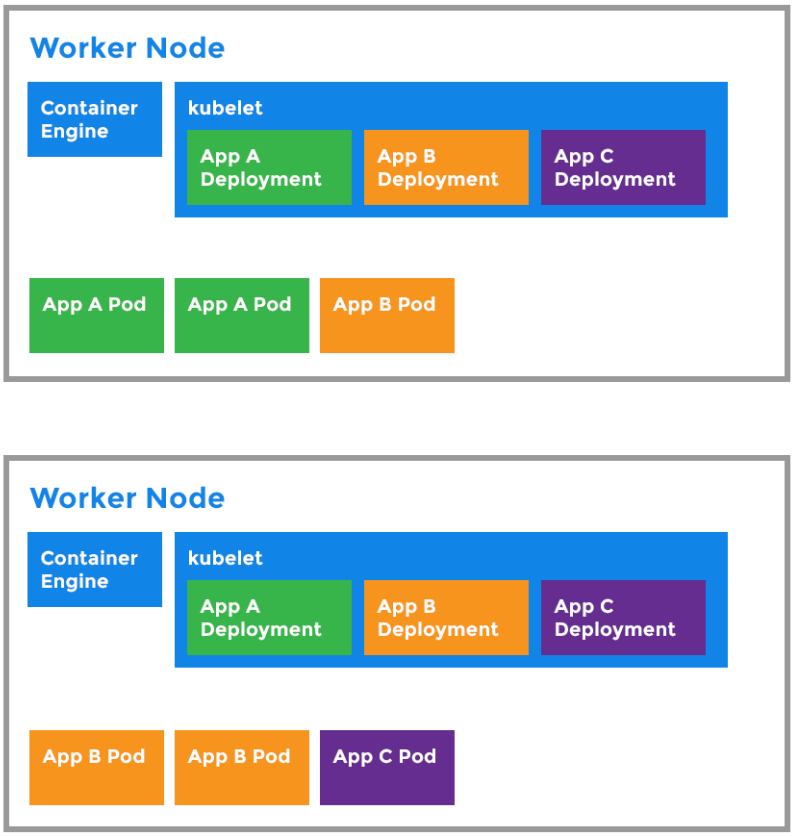


Figur 15: Pods (Ivancza, K., 2018)

5.4.7 Deployments

Deployments representerer et sett med flere, identiske Pods uten unike identiteter. En Deployment kjører flere replicas av applikasjonen din, og erstatter automatisk eventuelle Pods med feil. På denne måten kan Deployments sørge for at alltid en eller flere forekomster av applikasjonen din er tilgjengelige for for eksempel brukerforespørsler. Deployments administreres av Kubernetes Deployment-kontrolleren.

Deployments bruker en Pod-mal, som inneholder en spesifikasjon for Podene. Pod-spesifikasjonen bestemmer hvordan hver pod skal se ut: hvilke applikasjoner som skal kjøre inne i containeren, hvilke volumer podene skal ha etc. Når en Deployment blir endret, blir nye Pods automatisk opprettet en om gangen.



Figur 16: Deployment (Ivancza, K., 2018)

Deployments er velegnet for stateless applikasjoner som bruker ReadOnlyMany eller ReadWriteMany-volumer montert på flere replicas, men er ikke godt egnet for arbeidsbelastninger som bruker ReadWriteOnce-volumer. For stateful applikasjoner som bruker ReadWriteOnce volumer, bruker man StatefulSets. StatefulSets er utviklet for å distribuere stateful applikasjoner og clustered applikasjoner som lagrer data til vedvarende lagring. StatefulSets er egnet for å distribuere Kafka, MySQL, Redis, ZooKeeper og andre applikasjoner som trenger unike, vedvarende identiteter og stabile vertsnavn.

Man oppretter Deployments ved bruke kubectl run, kubectl apply eller kubectl create kommandoer. Når du har opprettet en deployment, sikrer den for at ønskede antallet pods kjører og er tilgjengelig til enhver tid. Distribusjonen erstatter automatisk Pods som mislykkes eller utsettes fra deres noder. Følgende er et eksempel på en Deployment i YAML-format:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
```

```
      app: nginx
  template:
      metadata:
      labels:
      app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
      - containerPort: 80
```

I dette eksempelet:

- En deployment kalt **nginx** blir opprettet, dette ser vi fra **metadata: name** feltet
- Deploymenten oppretter tre replicas (Pods), angitt av **replicas** feltet.
- Podmalen, eller **spec: template**, indikerer at Podene er merket **app: nginx**.
- Pod-malens spesifikasjon, eller **template: spec** feltet, indikerer at Pods kjører en container, **nginx**, som kjører **nginx** Docker Hub-imaget på versjon 1.7.9.
- Deploymenten åpner port 80 for bruk av Podsene.

For å oppsummere, så inneholder Pod templaten følgende instruksjoner for Pods opprettet av denne Deploymenten:

- Hver Pod er merket **app: nginx**.
- Opprett en container og gi den navnet **nginx**.
- Kjør **nginx**-image med versjon **1.7.9**.
- Åpne port **80** for å sende og motta trafikk.

5.4.8 Secrets

Objekter av typen secrets er ment å holde sensitiv informasjon, for eksempel passord, OAuth-tokens og SSH-nøkler. Å legge denne informasjonen i en secret er tryggere og mer fleksibel enn å sette den i en Deployment eller i et docker image.

Hvis man i stedet lagrer slik sensitiv informasjon i et secret objekt muliggjør det mer kontroll over hvordan den brukes, og reduserer risikoen for utilsiktet eksponering. Brukere kan skape secrets, og systemet skaper også noen egne secrets. For å bruke en secret må en pod referere til secreten som skal brukes. En secret kan brukes med en pod på to måter: Som filer i et volum montert på en eller flere av dets containere, eller brukes av kubelet når du puller images for en Pod.

Eksempel: Hvis noen Poder trenger tilgang til en database eller lignende, oppretter

man to tekstfiler som heter username.txt og password.txt. Kommandoen utføres på master noden og blir som følger:

```
$ echo -n 'Administrator' > ./username.txt
$ echo -n 'Passord1' > ./password.txt
```

Kommandoen lager her to filer med et username lik “Administrator” og et passord lik “Passord1”.

Vi bruker så kommandoen **kubectl create secret**. Denne kommandoen pakker disse filene inn i en Secret og lager dette objektet på API serveren, slik at det er tilgjengelig for Podene i clusteret.

```
$ kubectl create secret generic db-user-pass --from-file=./username.txt --from-
file=./password.txtsecret "db-bruker-tilgang" created
```

Etter man kjører kommandoen kan man se om secreten ble lagret ved å kjøre kommandoen **kubectl get secrets**.

```
$ kubectl get secrets
NAME                     TYPE         DATA         AGE
db-bruker-tilgang       Opaque         2           51s
```

Får her opp en secret med navn db-bruker-tilgang, som er lik den som ble laget med create secret. Man kan så kjøre kommandoen kubectl describe secrets/navn på secret for å finne flere detaljer.

```
$ kubectl describe secrets/db-bruker-tilgang
Name:        db-bruker-tilgangNamespace:       defaultLabels:
      <none>Annotations:  <none>Type:           OpaqueData
====password.txt:      8 bytes
username.txt:      13 bytes
```

En viktig detalj å merke seg er at ingen av **get** eller **describe** kommandoene viser innholdet i password.txt og username.txt. Dette er for å beskytte mot uheldig eksponering når noen for eksempel går igjennom terminal loggen.
Man kan decode secreten for å se passordet i klartekst, dette gjøres ved å kjøre **kubectl get secret** kommandoen igjen:

```
$ kubectl get secret db-bruker-tilgang -o yaml
apiVersion: v1
data:  username: QWRtaW5pc3RyYXRvcg==  password: UGFzc29yZDE=kind: Secret
```

Kan så bruke Base64 decoding for å se passordet:

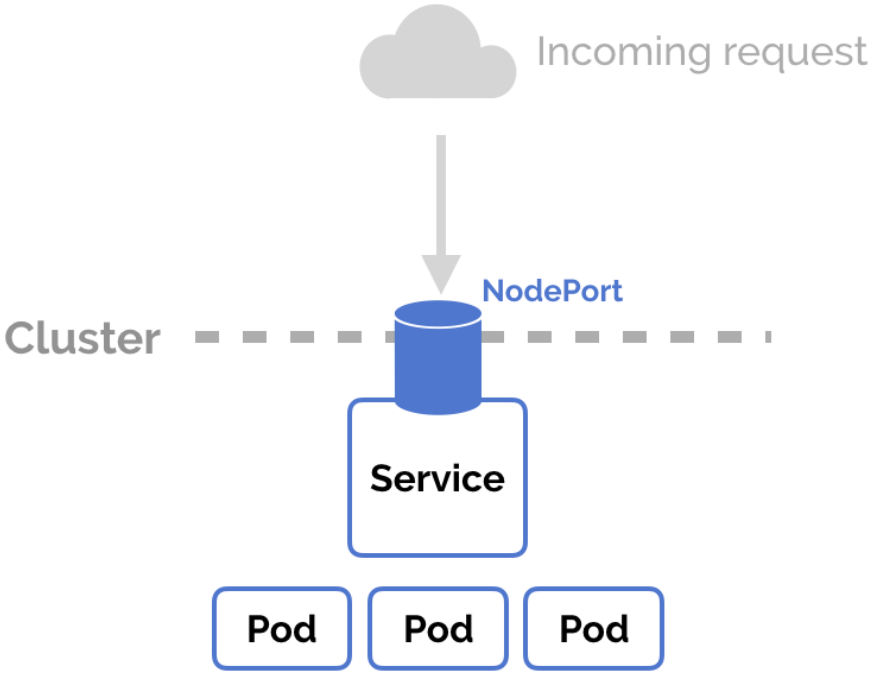
```
$ echo 'UGFzc29yZDE=' | base64 -decode
$ Passord1
```

5.4.9 Services

For enkelte deler av en applikasjon (for eksempel frontends) vil du kanskje eksponere en tjeneste på en ekstern IP-adresse utenom clusteret ditt. En service i Kubernetes er ansvarlig for å gjøre Podene våre synlige i nettverket eller eksponere dem til internett.
Kubernetes ServiceTypes lar deg spesifisere hvilken type service du vil bruke. Det er hovedsakelig fire typer, ClusterIP (som også er default), NodePort, LoadBalancer og nylig Ingress.

ClusterIP
Eksponerer tjenesten på en intern IP-adresse. Å velge denne ServiceTypen gjør tjenesten bare tilgjengelig fra inne i clusteret. Tjenesten får en intern ClusterIP tildelt.

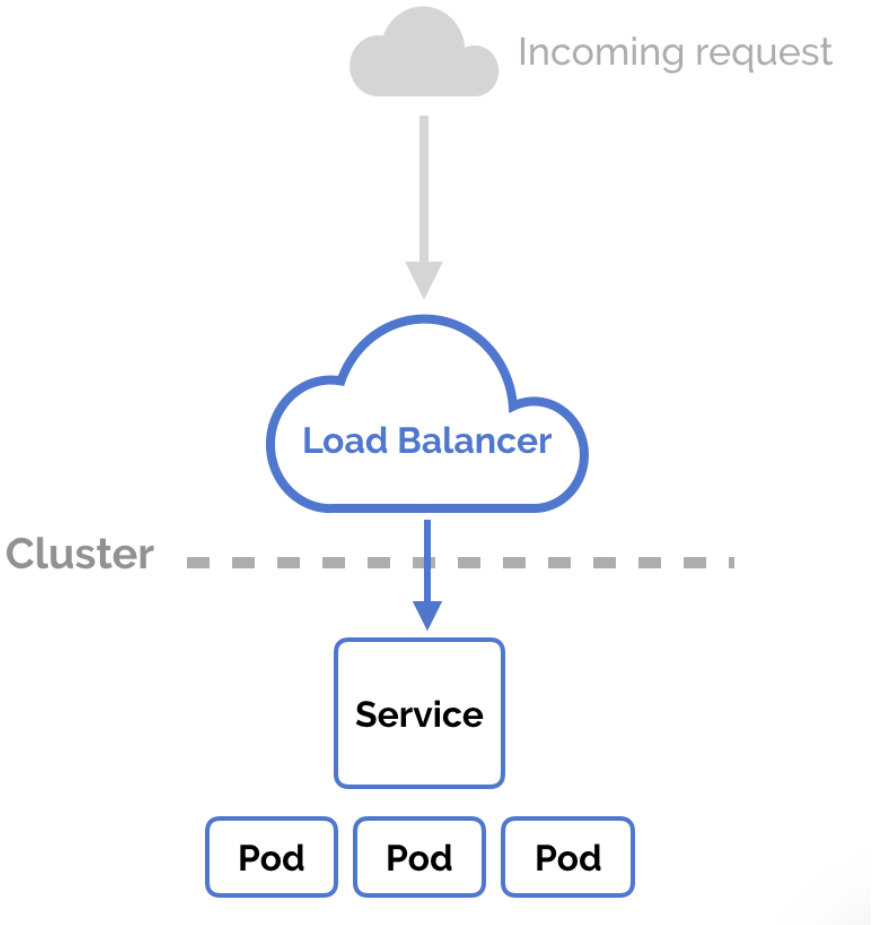
NodePort
NodePort er en konfigurasjonsinnstilling du setter i tjenestens YAML fil, under **type: NodePort**. Da vil Kubernetes allokere en bestemt port på hver node til den tjenesten, og enhver forespørsel til clusteret på den porten blir videresendt til tjenesten. Dette er enkelt, men ikke veldig robust. Du vet ikke hvilken port din tjeneste skal tildeles, og porten kan bli reallokert på et tidspunkt.. Du kan kontakte NodePort-tjenesten fra utenfor clusteret ved å be om **<NodeIP>: <NodePort>**.



Figur 17: NodePort (Palmer, M., 2018)

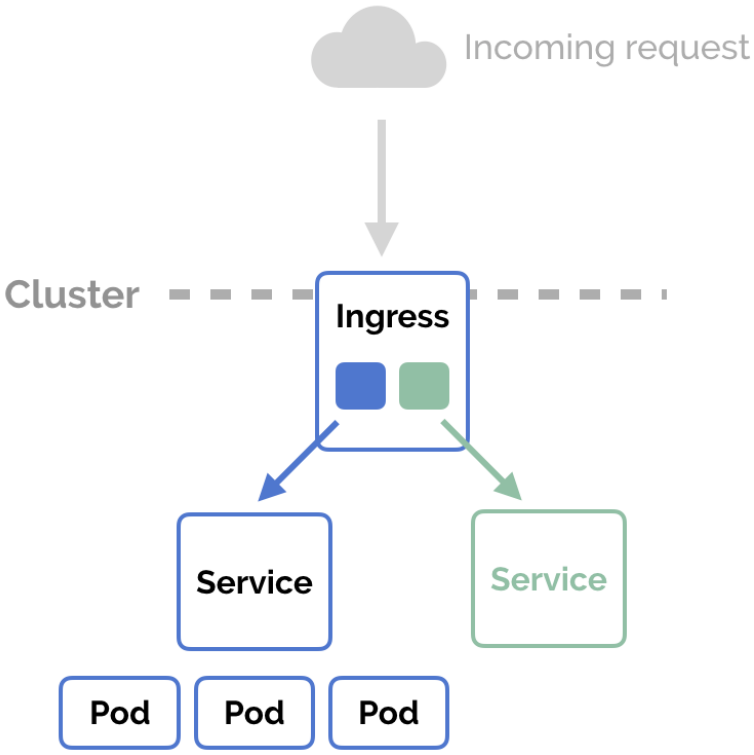
Load Balancer

Du kan angi en tjeneste av typen LoadBalancer på samme måte som du ville sette NodePort. Vi legger til type: LoadBalancer. Det må være en ekstern lastbalanseringsfunksjonalitet i clusteret, vanligvis implementert av en skyleverandør (Azure, AWS eller Google). Kubernetes Engine oppretter en LoadBalancer med en IP-adresse som du kan bruke for å få tilgang til tjenesten. Hver gang du vil eksponere en tjeneste til omverdenen, må du opprette en ny service av typen LoadBalancer. Trafikk fra den eksterne lastbalansereren vil bli rettet mot backend Pods, hvordan dette nøyaktig fungerer avhenger av skyleverandøren.



Figur 18: Load Balancer (Palmer, M., 2018)

Ingress
NodePort og LoadBalancer lar deg eksponere en tjeneste ved å spesifisere type tjeneste i service konfigurasjonsfilen. Ingress, derimot, er en helt uavhengig ressurs til din tjeneste. Ingress gir deg mulighet til å sende forespørsler til tjenester basert på host eller path, som igjen vil føre til sentralisering av en rekke tjenester til et enkelt inngangspunkt.



Figur 19: Ingress (Palmer, M., 2018)

5.4.10 Cluster Autoscaler

Kubernetes sin cluster autoscaler skalerer automatisk et cluster basert på kravene til arbeidsbelastningene du vil kjøre. Når autoscaler er aktivert, legger Kubernetes automatisk til en ny node i clusteret ditt hvis du har opprettet nye Pods som ikke har nok tilgjengelige ressurser til å kjøre. Omvendt, hvis en node i clusteret ditt er underutilisert, kan Kubernetes fjerne noden.Ved hjelp av Cluster Autoscaler kan du bare betale for ressurser som trengs på et gitt tidspunkt, og automatisk få ekstra ressurser når etterspørselen øker.
Det er viktig å huske på at når ressursene slettes eller flyttes i løpet av skalering av ditt cluster, kan tjenestene dine oppleve forstyrrelser. Hvis tjenesten for eksempel består av en kontroller med en enkelt kopi, kan replica Pods starte på nytt på en annen node hvis den nåværende noden blir slettet.

Hvordan cluster autoscaler fungerer
Cluster autoscaler fungerer på per-node basis. For hvert nodepool kontrollerer autoscaler periodisk om det finnes Pods som ikke er scheduled og venter på en node med tilgjengelige ressurser. Hvis slike Pods eksisterer, og autoscaler bestemmer at resizing av et node pool ville tillate at ventende Pods blir scheduled, så ekspanderer autoscaler nodepool-et.
Cluster autoscaler måler også bruken av hver node opp mot et nodepools totale kapasitetsbehov. Hvis en node ikke har hatt noen nye scheduled Pods på den i en bestemt tidsperiode, og alle Pods som kjører på den noden kan kjøre på andre noder i poolet, så flyttes podene og noden blir slettet. Hvis noden ikke kan bli drainet etter 10 minutter, blir noden tvunget til å avslutte.

Begrensninger
Cluster autoscaler har følgende begrensninger:

- Cluster autoscaler støtter opptil 1000 noder som kjører 30 Pods hver.
- Ved nedskalering er det en avslutningsperiode for en Pod på opptil 10 minutter. En pod blir alltid slettet etter maksimalt 10 minutter, selv om Poden er konfigurert med en høyere grace periode.

6. Teori om Azure

6.1 Hva er Azure

Når vi snakker om Azure snakker vi som oftest om “Cloud Computing”. “Cloud Computing” er tjenester som er tilgjengelige via internett og baserte på programvare brukeren ikke er i direkte forbindelse med. Azure kjører i store datasentre hos Microsoft. Det betyr at applikasjoner og data befinner seg i disse datasentrene. Fordelene ved dette er at hverken bruker eller leverandør behøver å forholde seg til drift eller skalering, men lar Microsoft håndtere infrastrukturen. Plattformen tilbyr blant annet integrerte tjenester for lagring, nettverk og applikasjoner. Kombinert med både IaaS (Infrastructure-as-a-service) og PaaS(Platform-as-a-service) gir stor fleksibilitet i å bygge og lansere ulike typer skreddersydde løsninger. Med datasenter over hele verden og flere i Europa gir Azure-brukeren ytelse der den er. Det ble i juni 2018 annonsert at Microsoft skal lansere to nye datasenter regioner i Norge (Microsoft, 2018).

Azure er en åpen plattform med støtte for alle operativsystemer, språk og verktøy med muligheten til å kjøre hybridløsninger der deler av infrastrukturen er i skyen og resten er on premise. Azure kan raskt skalere opp eller ned etter behov, slik at du bare betaler for det du bruker.

Skyplattformen Azure består av tre hoveddeler. Den første heter enkelt og greit Windows Azure, som er et Windows-basert miljø for applikasjoner og lagring, altså et operativsystem for nettverk. Den andre delen er SQL Azure og omfatter databasetjenester basert på SQL Server. Den tredje er .Net Services som består av forskjellige infrastrukturtjenester for både skybaserte og lokale applikasjoner.

Azure har tre forskjellige grunnkomponenter: En lagringsdel, en beregningsdel og det man på engelsk kaller "fabric" -- struktur, oppbygning, råbygg, rammeverk. Lagringsdelen tar seg av lagring, beregningsdelen eksekverer applikasjonen og fabric-delen byr på metoder for å administrere og overvåke programvaren som kjører.

Alle data i Windows Azure replikeres tre ganger. Det gir plass til en viss feiltoleranse og en tapt kopi er ikke umiddelbart en katastrofe. Windows Azure sørger også for å lagre en ekstra sikkerhetskopi i et annet datasenter i den samme delen av verden.

Når det gjelder lagring av kunders data, kan det være interessant å vite at Windows Azure er sertifisert i henhold til standardene ISO/IOC 27001:2005 og AIPCA SAS 70. Dette er standarder med høye krav til informasjonssikkerhet og skal sikre at privat informasjon ikke kan lekke ut.

6.1.1 Skalering etter behov

Et IT-system kan skaleres på flere måter. Ved innførelsen av virtuelle maskiner ble det populært å skalere i høyden. Skalering i høyden er å gi den enkelte datamaskinen eller ressursen mer kraft til å utføre oppgavene sine. Et eksempel på dette kan være å øke minnet og legge til flere CPUer. Denne skaleringsteknikken fungerer bare opp til et visst punkt, da den har fysiske begrensninger på hvor mye minne det er plass til, og hvor mange CPU-er det er mulig å sette inn i en datamaskin. Selv om enkelte ressurser på en maskin kan skaleres, vil det fortsatt være andre som ikke kan det. For eksempel nettverkstrafikk og diskkontrollere.

Skalering i bredden er det andre konseptet for å skalere et IT-system. Dette vil si å bygge ut med flere maskiner eller ressurser for å ta unna arbeidsmengden til systemet. Skalering i bredden har flere fordeler, blant annet gir det en naturlig «fail over», i tilfellet en maskin eller komponent feiler. Ved å skalere alle deler av et system i bredden unngås «single point of failure» og det blir mulig å operere med høy oppetid.

Når IT-systemer skaleres i bredden er det nødvendig med infrastruktur som støtter opp under dette. Microsoft Azure tilbyr lastbalansering som IaaS. Traffic Manager som lastbalansereren heter har tre forskjellige moduser:

- Round robin – forespørslene fordeles mellom nodene
- Ytelse – forespørslene sendes til den nærmeste noden i tid
- Fail over – forespørslene sendes til en node, men går til en eller flere backupnoder om hovednoden skulle falle ut.

6.1.2 Sikkerhet og ytelse

Data er en fysisk gjenstand som eksisterer og befinner seg på en eller flere lokasjoner til bestemte tidspunkt. I Microsoft Azure kan data lagres på flere forskjellige steder. I Norge er det et krav om at lagring av all person- og forretningsdata lagres i Dublin eller Amsterdam av følgende grunner:

- Data ligger innenfor EØS og følger norsk og europeisk lovgivning både med tanke på personvern og sikkerhet.
- Datasentrene har god kapasitet seg imellom og tjenester er naturlig speilet mellom dem.
- Rask responstid fra Norge.

Lagring av data i Microsoft Azure eller hos en tradisjonell driftsleverandør er et valg om hvor de fysiske dataene skal befinne seg. I prosessen med valg av Microsoft

Azure eller tradisjonell drift er det viktig å vurdere mekanismene rundt dataene. Hvilke sikkerhetsmekanismer kan applikasjonen implementere rundt dataene og hvordan er rutinene for driftsteknikere knyttet til data. Myndighetenes innsynsrett i data er også noe som må vurderes i valget om bruk av Azure eller ikke. Det er viktig å huske at selv åpne offentlige data må sikres, så ikke en tredjepart kan bruke systemet til å spre sitt budskap og sine data.

Ikke alle data kan lagres innenfor EØS, for brukere fra Asia, USA og Australia vil det ta lang tid å hente data i Europa. Microsoft Azure har da flere muligheter for å lagre data i andre kontinenter, alt avhengig av type data og lagringsteknologi. Det vil for eksempel være mulig å lage en lese-optimalisert relasjonsdatabase, filområde eller NoSQL i et annet kontinent, for så å la brukere der bruke sin lokale kopi for lesing. I tilfeller som dette er det viktig å vite at lokale regler for datalagring gjelder. Det mest vanlige scenarioet er nettsider som skal være raske og brukervennlige over hele verden. Til dette brukes CDN-tjenesten i Microsoft Azure. Statiske filer som bilder, CSS og JavaScript lastes opp til CDN og spres til hele verden. Microsofts CDN nettverk er spredt rundt i 29 forskjellige land, så brukerne vil alltid hente innholdet fra en server nært seg. Dette gir meget god ytelse uavhengig av lokasjon.

6.1.3 Kostnader ved bruk av Azure

Før var det kostbart å kjøpe datamaskiner, og det krevdes forholdsvis mye menneskelig interaksjon for å drifte dem. Nå er det mulig i store datasentre å automatisere driften mest mulig, og bare ha mennesker som agerer på avvik i driften. Selve hardwaren har også blitt mye billigere enn det den en gang var. Azure er bygget opp av veldig mange mid-end datamaskiner, dette er maskiner som ikke koster så mye i innkjøp, kraften ligger i måten de er satt sammen, for å skape et stort datasenter. Den drivende kosten ved et sånt datasenter er strømforbruket til å drive maskinparken og kjøling. Microsoft er blant de verdensledende i grønne datasentre og der igjen lavest mulig strømforbruk. For eksempel datasenteret i Dublin klarer seg hovedsakelig med luftkjøling fra den kalde havluften der. Energieffektive store datasentre gjør det mulig å tilby datakraft til meget konkurransedyktige priser.

Det beste med denne prismodellen er at det kun betales for bruk. For de fleste tjenestene betales det bare for faktisk diskplass, overført trafikk, eller per aktive minutt. Det kan derfor bli kostnadseffektivt å nedskalere applikasjoner om natten hvis det er mindre bruk da. Store kostnader i forbindelse med kjøp av infrastruktur er ikke nødvendig lenger, i Microsoft Azure er det mulig å begynne med lav kapasitet, og øke gradvis ettersom bruken av systemet øker.

6.2 Kubernetes i Azure

Hvis man skal rulle ut et Kubernetes cluster i Azure er det flere muligheter for

hvordan en kan gå frem med dette. Hvordan det gjøres kommer ann på hva man ønsker å oppnå med Kubernetes clusteret og hva det skal brukes til.

Azure har flere verktøy som forenkler implementasjon og administrasjon av Kubernetes i et Azure miljø. Ett av dem er innebygd som en tjeneste i Azure (AKS) og det andre er et open source prosjekt (AKS Engine) Microsoft bidrar til å vedlikeholde og som de selv bruker deler av i sin innebygde tjeneste. Vi skal nå beskrive disse to og hva forskjellen på dem er.

6.2.1 Azure Kubernetes Service (AKS)

Azure Kubernetes Service er en fullbyrdig gratis administrert tjeneste tilbudt til brukere av Azure. AKS gjør det enkelt å distribuere et styrt Kubernetes Cluster i Azure ved å ta seg av mye av administrasjonen og reduserer kompleksiteten av orkestrering for brukere. AKS konfigurerer alle Kubernetes-mastere og worker-noder under utrullingen av clusteret, samt at masteren blir styrt og vedlikeholdt av Azure. Brukere administrerer og vedlikeholder kun worker-noder. Du betaler også kun for worker-nodene i clusteret, ikke for master-noden. Selve administrasjonstjenesten er som nevnt gratis. Siden det er en administrert tjeneste, håndterer Azure alle Kubernetes-oppgraderinger for tjenesten, ettersom nye versjoner blir tilgjengelige. Brukere kan bestemme om og når oppgraderingen skal skje i sitt Kubernetes cluster.

Azure håndterer også oppgaver som Azure Active Directory integrasjon, tilkoblinger til overvåkningstjenester og konfigurering av avanserte nettverksfunksjoner.

Alle med et Azure abonnement kan opprette et Kubernetes cluster med AKS via Azure-portalen, Azure CLI eller mal-drevne alternativer som Resource Manager-maler og Terraform.

Alt dette gjør at AKS er et perfekt verktøy for noen som ikke er veldig erfarne med Kubernetes som orkestreringsverktøy. Men et av de største problemene med denne tjenesten, i hvert fall for dette prosjektet, er at det ikke støtter Windows worker-noder og dermed heller ikke Windows containere rett ut fra boksen. Det er kun Linux VM-er som blir rullet ut som worker-noder i et AKS Kubernetes cluster og siden man ikke kan kjøre Windows containere på en Linux OS host er det et stort problem for vårt prosjektet.

Det finnes et open-source prosjekt kalt Virtual Kubelet som brukes til å lage en type løsning for å få Windows containere til å fungere i et AKS Kubernetes cluster. Det ble annonsert i mai 2018 (Foulds, I., 2018) at det er planlagt støtte for Windows containere i AKS uten å måtte bruke Virtual Kubelet og det er en private preview for dette man kan melde seg på. Siste nytt på denne fronten, av 05.04.2019, er at støtte kommer men det er ikke noe offisielt nytt å komme med (Microsoft, 2019f), så dette

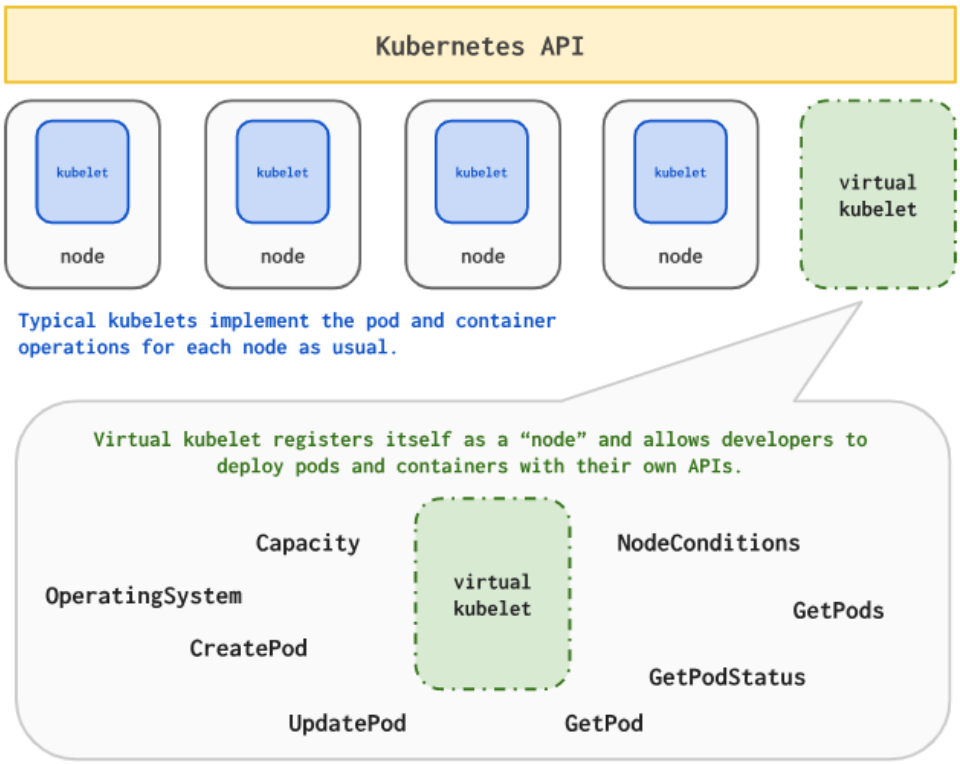
er ikke noe vi kan se på enda.

Men hva er egentlig Virtual Kubelet?

6.2.1.1 Virtual Kubelet og ACI

Til å starte med må vi si noe om hva ACI er, da Virtual Kubelet og ACI er nært knyttet sammen. ACI står for Azure Container Instances, det er en tjeneste som lar deg distribuere containere i den offentlige Azure skyen uten å måtte rulle ut noe som helst infrastruktur. Azure administrerer alt av nødvendig underliggende infrastruktur for å kjøre disse containerne. Tjenesten - som støtter både Linux- og Windows-containere – eliminerer derfor behovet for at må rulle ut virtuelle maskiner, eller implementere en container-orkestreringsplattform, som for eksempel Kubernetes, for å distribuere og kjøre containere. Med ACI får man en enkel måte å rulle ut containere på, men man mister mye av kontrollen på orkestreringslaget.

Virtual Kubelet utnytter dette ved å maskere seg som en kubelet i et Kubernetes cluster for å koble Kubernetes til andre APIer, som ACI. Diagrammet under viser hvordan en Virtual Kubelet later som den er en node i et cluster for å koble Kubernetes mot ACI.

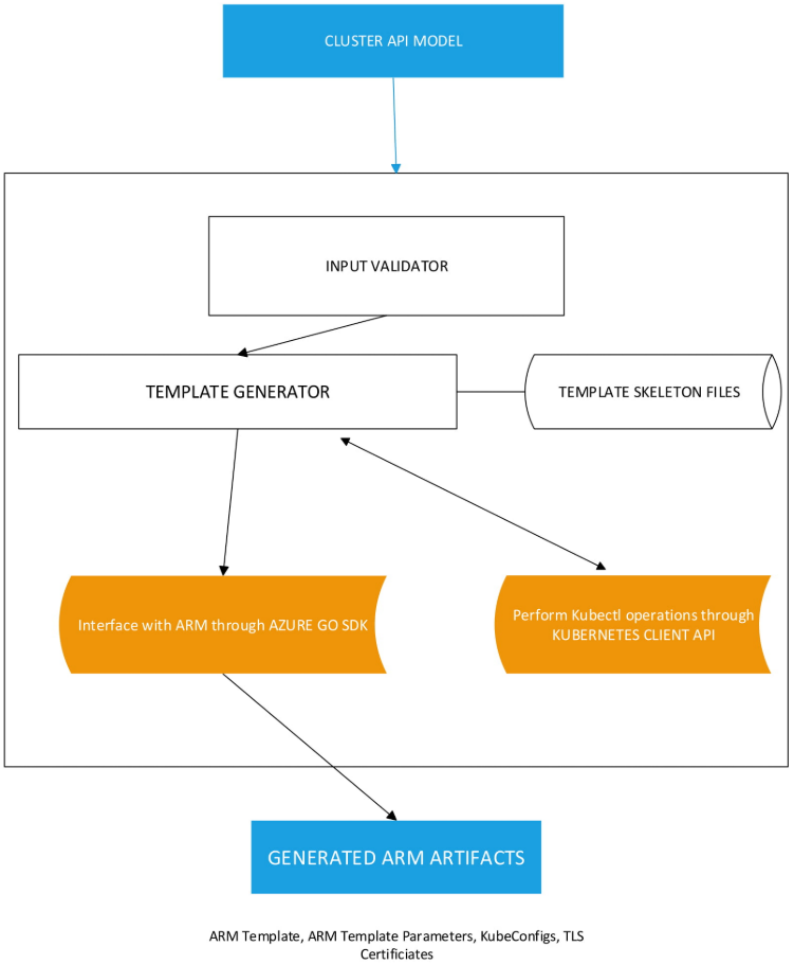


Figur 20: Virtual Kubelet (Github, 2019c)

På denne måten får man noe som oppstår som en "Virtual Kubelet" worker node i Kubernetes clusteret som man kan kjøre containere på, både Windows og Linux containere støttes av Virtual Kubelet.

6.2.2 AKS Engine

Det andre verktøyet for å forenkle implementasjon av et Kubernetes cluster i Azure er noe som ligner veldig på AKS, og det er AKS Engine.
AKS Engine leverer praktiske verktøy for enkel og rask oppstart av Kubernetes clustere i Azure. Det er et Azure open source-prosjekt. Ved å utnytte ARM (Azure Resource Manager) hjelper AKS Engine med å opprette, slette og vedlikeholde Kubernetes clusters med grunnleggende IaaS-ressurser i Azure. Under ser dere et diagram over arkitekturen til AKS Engine.



Figur 21: AKS Engine arkitektur (Github, 2019e)

Vi kan raskt gå gjennom stegene i prosessen og forklare hva de enkelte delene faktisk gjør.

Cluster api model
Dette er en JSON fil du forer til AKS Engine kommandolinjeverktøyet. Denne filen inneholder konfigurasjonsinformasjon om ting som:

- Master og worker noder – Skal det være linux worker noder, windows worker noder eller begge? Hvilken versjon av OS skal installeres? Hvordan type VM (cpu/minne/lagring etc) skal rulles ut?
- Kubernetes versjon
- Autentiserings parametere for Azure plattformen

Input Validator
Dette er en sjekk etter feil eller manglende input i JSON api modellen som bruker forsøker å bruke. Hvis det oppdages problemer, stoppes prosessen og valideringsfeilmelding blir rapporter tilbake til bruker.

Template Generator
Etter validering er gjennomført, kalles mal generatoren som konverterer JSON api modellen til et format som er forståelig for ARM. Mal generatoren bruker verdiene som er tilstede i cluster api modell filen til å fylle inn eksisterende maler med et format som er forståelig for ARM. Disse eksisterende malene inneholder plassholdere som blir byttet ut med verdiene i cluster api modell json filen. Malene nøster også andre mal finner inne i seg for å få det hele til å fungere. Disse malene er kalt Template Skeleton Files i diagrammet, figur 21.
Mal generatoren lager dermed følgende artefakter:

- ARM maler (utrulling og parameter JSON filer) - Disse malene brukes av ARM for å gjennomføre den faktiske utrullingen av Kubernetes clusteret i Azure.
- KubeConfigs – Dette er Kubernetes config filer som kan brukes av brukeren eller Kubernetes API klientene til å utføre kubectl-kommandoer mot Kubernetes clusteret direkte. Disse filene er nødvendige for å kunne administrere clusteret fra en arbeidsstasjon.
- TLS-sertifikater – For å skape sikre forbindelser med Azure VM-er som blir rullet ut.

ARM Interface
AKS Engine snakker med ARM gjennom Azure Go SDK. Go SDK gir et grensesnitt for å utføre funksjoner som; utrulling fra maler, validering osv.

Kubernetes Client API
AKS Engine utfører også Kubernetes cluster administreringskommandoer (kubectl) gjennom importerte Kubernetes API-biblioteker. Klient API kall gjøres under skalering og oppgradering kommandoene til AKS Engine.

Dette kan kanskje se litt komplisert ut, men det eneste en vanlig bruker av AKS Engine har kontroll over er Cluster api modell JSON filen man forer inn til AKS Engine. Resten blir gjort automatisk av AKS Engine, og så får du noen ARM maler som du kan bruke til å rulle ut clusteret i Azure og noen kubeconfig filer du kan bruke til å administrere clusteret når det er rullet ut. Api modell JSON filene kan være veldig enkle til å starte med. Når du har litt erfaring med dette kan du utnytte fleksibiliteten til AKS Engine for å lage spesielt tilpassede Kubernetes clusters ut ifra dine behov.

AKS Engine er også biblioteket som brukes av AKS (Azure Kubernetes Service) for å utføre lignende operasjoner for å levere administrerte tjenester. AKS bruker AKS Engine internt for å gjennomføre flere av oppgavene sine.
Med AKS Engine kan du lage dine egne Kubernetes clusters med tilpassede krav, som for eksempel at det må være worker-noder i clusteret som kjører Windows OS, slik at vi kan rulle ut Windows applikasjoner i containere på vårt cluster.

Du mister noen av fordelene med å ha et cluster som er fullt administrert av Azure, slik som AKS. Det blir en del mer arbeid med administrasjon av master-noden med mer, men du oppnår mer fleksibilitet. Og selv om det er løsninger der du kan bruke AKS i sammenheng med Virtual Kubelet for å kunne kjøre Windows containere mangler dette som vi så på tidligere mye funksjonalitet for øyeblikket. Monitorering og automatisk skalering blir veldig mye mer komplisert og i flere tilfeller ikke praktisk med AKS og Virtual Kubelet i forhold til mulighetene man får med å ha fullverdige worker noder med Windows OS med AKS Engine.

6.2.3 “On premise” i Azure

En mulig løsning som kan nevnes er å rulle ut VM-er i Azure, late som at de er fysiske servere, sette opp eget nettverk i Azure og dermed simulere en on-premise løsning. For deretter å installere og konfigurere et Kubernetes cluster fra bunn uten å bruke verken av verktøyene nevnt tidligere. Dette føler prosjektgruppen blir unødvendig ekstraarbeid, i form av installasjon, konfigurasjon og administrasjon, og kommer til å ta mer tid enn ønsket så vi ser ikke mye nærmere på dette.

7. Teori om monitoreringsverktøy

IT bransjen har lenge vært avhengig av microservice-basert arkitektur for å levere programvare raskere og tryggere. Som igjen har banet veien for containerteknologi. Containerteknologi har fått oss til å tenke på hvordan vi bygger og distribuerer våre applikasjoner. Bruk av Docker eksploderte i 2013, og for bedrifter som fokuserer på å modernisere infrastrukturen og skymigrasjonen, er Docker en avgjørende applikasjon for å distribuere applikasjoner raskt, og i høy skala.

Men når man skal distribuere applikasjoner i en slik høy skala, kommer en del utfordringer. Containere innfører et ikke-trivielt nivå av kompleksitet når det gjelder orkestrasjon. Det er nettopp her Kubernetes kommer inn i bildet.

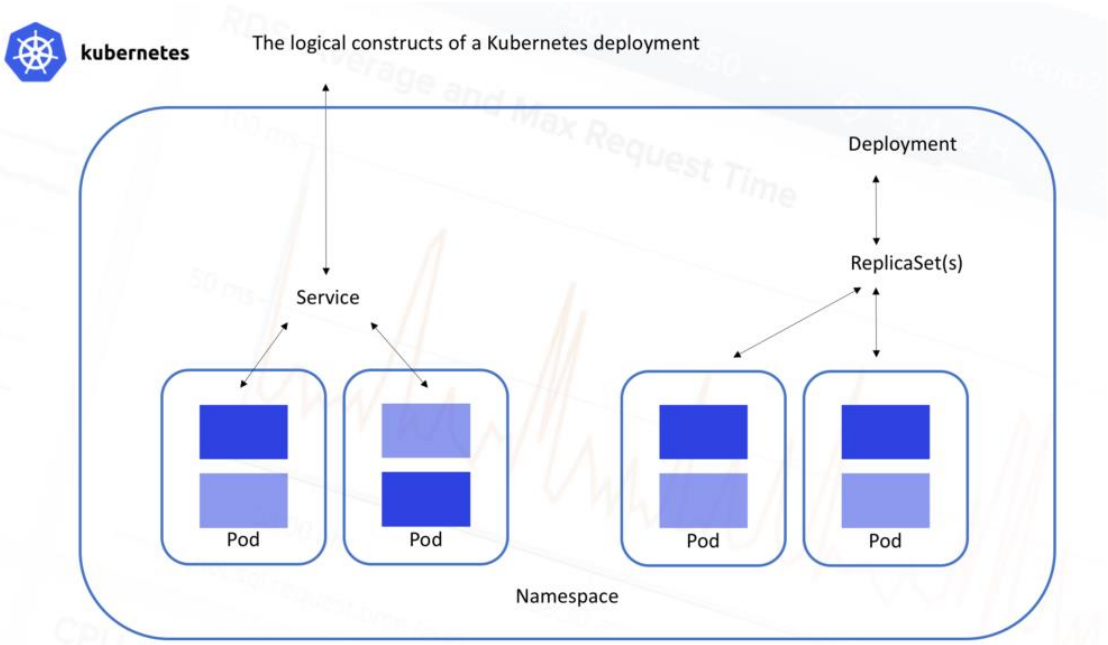
Kubernetes fungerer som hjernen for distribuert containerutplassering. Den er designet for å administrere serviceorienterte applikasjoner ved hjelp av containere fordelt over flere noder. Kubernetes gir mekanismer for applikasjonsutplassering, serviceoppdagelse, planlegging, oppdatering, vedlikehold og skalering. Men hva med å overvåke Kubernetes?

Mens Kubernetes har potensial til å dramatisk forenkle oppgaven med å distribuere applikasjonen din i containere, legger den også til et nytt sett av kompleksiteter for de daglige oppgavene dine for å administrere applikasjonsytelse, å få synlighet i tjenester og den typiske overvåking -> varsling -> feilsøking arbeidsflyten.

I tillegg til økt infrastrukturkompleksitet, blir applikasjoner nå utviklet for mikroservices, hvor signifikant flere komponenter kommuniserer med hverandre. Hver tjeneste kan distribueres over flere forekomster; Docker containere beveger seg over infrastrukturen etter behov. Mens før vi visste hvor mange forekomster vi hadde av hver tjenestekomponent, og vi visste hvor de befinner seg, er det ikke lenger tilfelle. Hvordan påvirker dette Kubernetes overvåkingsmetodikk og verktøy?

7.1 Forståelse av Kubernetes og dens kompleksiteter

Fra et infrastrukturmessig synspunkt består et Kubernetes cluster av et sett med noder som overvåkes av en master node. Masterens oppgaver inkluderer orkestrering av containere over noder, holde oversikt over tilstanden og eksponering av clusteret gjennom et REST API og et brukergrensesnitt. Bildet under viser noe forenklet hvordan en applikasjon er bygget opp i Kubernetes.



Figur 22: En applikasjon i Kubernetes (Davé, A., 2018)

Alle containere kjører inne i pods. En pod er et sett med containere som lever sammen. De er alltid samlokalisert og samordnet, og kjører i en felles sammenheng med delt lagring. Containerne i podene er garantert å være samlokalisert på samme maskin og kan dele ressursene. Podene sitter vanligvis bak tjenester, som tar seg av å balansere trafikken, og eksponerer også podene opp mot en enkelt IP-adresse. Tjenestene skaleres horisontalt av et ReplicaSet som lager eller fjerner pods for hver tjeneste etter behov. ReplicaSets styres videre av deployments som gjør at du kan deklarere tilstand for en rekke løpende ReplicaSets og pods. Namespaces er virtuelle clustere som kan inkludere en eller flere tjenester. Metadata tillater bruk av etiketter og koder for å markere containere basert på distribusjonsegenskapene.

Så for å gjenta, så kan flere tjenester og til og med flere namespaces spres over samme fysiske infrastruktur. Hver av disse tjenestene består av pods, som igjen kan bestå av et antall containere. Dette kan føre til noe ganske imponerende kompleksitet når det gjelder å overvåke selv en beskjeden Kubernetes-distribusjon.

7.2 Utfordringer med monitorering av containere

Kubernetes gjør det mye enklere for bedrifter å håndtere containere. En av fordelene med Kubernetes er at det kan distribuere programvare uavhengig av hvor de kjører, enten det er AWS, GCP, Azure eller bare metal. Uansett størrelsen på distribusjonen vår, trenger vi fortsatt å vite hvor mange tilgjengelige ressurser du har i den distribusjonen, samt å vite helsen til distribuerte applikasjoner og containere. Akkurat som microservices førte oss til å tenke på hvordan vi bygger våre applikasjoner, krever Kubernetes at vi forandrer vår tradisjonelle tilnærming til monitorering. Den dynamiske naturen til containerorkestrasjonen insisterer på en mer dynamisk

tilnærming til monitorering.

Her er noen utfordringer med å se monitorere Kubernetes og containere:

- Kubernetes er ekstremt dynamisk. Det vil si at applikasjoner kan leve på forskjellige hoster, og flyttes fram og tilbake. Nye servere kan slåes på og av i clusteret avhengig av last. Dette fører til at vanlige monitoreringsverktøy fungerer dårlig.
- I likhet med flyttingen fra monolitt til microservice-arkitektur betyr bruk av Kubernetes at det er mange, mindre deler å monitorere.

La oss først ta en titt på alternativene Kubernetes selv gir oss til å løse disse utfordringene.

7.3 Hvordan samle inn Kubernetes data

I hovedsak samler overvåkingsverktøyene Kubernetes data fra fire kilder:

1. Kubernetes hostene som kjører Kubelet. Kubernetes hostene har begrensede ressurser, så det er spesielt viktig å overvåke dem. Det finnes mange måter å få data ut fra disse hostene, som for eksempel Prometheus, Splunk, Datadog, Azure etc.

2. Kubernetes prosessen, også kalt Kubelet metrics, som inkluderer beregninger for api-server, kube-scheduler og kube-controller-manager. Disse gir deg informasjon om en Kubernetes node og jobbene den kjører.

3. Kubelet innebygde cAdvisor. Kubelet blir levert med innebygd støtte for cAdvisor, som samler, prosesserer og eksporterer metrics for våre containere. I motsetning til de fleste elementene i Kubernetes som opererer på Pod-nivået, fungerer cAdvisor per node. Den oppdager automatisk alle beholdere i den oppgitte noden og samler CPU, minne, og nettverksbrukestatistikk. cAdvisor gir også den generelle maskinbruken ved å analysere "root" containeren på maskinen. Det er viktig å være klar over begrensningene ved bruk av cAdvisor:
    - Det samler kun grunnleggende ressursutnyttelse - cAdvisor kan ikke fortelle deg hvordan programmene faktisk utfører, bare hvis en container har X% CPU-utnyttelse (for eksempel).
    - cAdvisor selv tilbyr ikke noen langsiktige lagrings-, trending- eller analysemuligheter.For å gå videre med disse dataene må vi legge til Heapster. Heapster overvåker data på tvers av alle noder i Kubernetes clusteret. Heapster kjører som en pod i clusteret, akkurat som enhver applikasjon.

- Heapster tillater deg heller ikke å lagre, trend eller varsle på data. Det gjør det bare enklere for deg å samle cAdvisor data over hele clusteret ditt. Disse dataene sendes deretter til en konfigurerbar backend for lagring og visualisering. Eksempler på støttende backends inkluderer InfluxDB, Google Cloud Monitoring og noen få andre. Du må legge til et visualiseringslag som Grafana for å presentere dataene dine.

4. kube-state-metrics gir oss informasjon på cluster nivå. Dette vil gi oss et større bilde av hva som skjer i Kubernetes clusteret vårt, for eksempel alle podene du har konfigurert og deres nåværende tilstand. Kube-state-metrics treffer alle Kubernetes-tjenester og samler inn informasjon om deres nåværende tilstand, for eksempel hvor mange containere som kjører, hvor mange er i en bestemt tilstand, om noen indikerer at de er usunne eller at vi nærmer oss full kapasitet, etc.

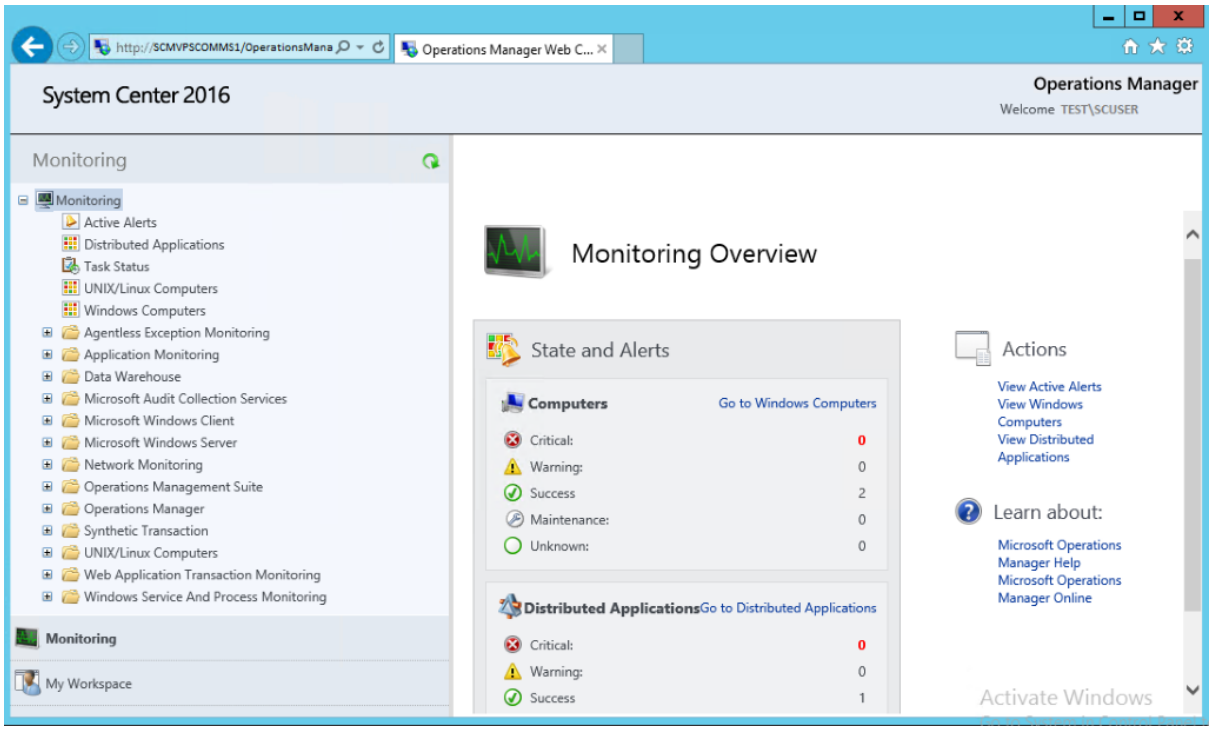
7.4 Valg av monitoreringsverktøy

Siden vi ikke har mye kunnskap rundt bruk av Kubernetes og kompleksiteten rundt monitoreringen av et cluster velger vi å se på flere forskjellige verktøy. Verktøyene vi skal ta en titt på er følgende.

- SCOM
- Splunk
- Prometheus m/influxDB og Grafana
- Azure Monitor
- DataDog

Under vil vi gå nærmere inn på de forskjellige alternativene. I driftsrapporten vil det så beskrevet hvilken løsning vi endte opp med. Derfor vil vi ikke på nåværende tidspunkt gå i detalj i hver enkelt løsning, men nevne kort hva de gjør. Det er noen løsninger vi allerede på nåværende tidspunkt ikke velger å gå videre med, noen på bakgrunn av at vi ikke tror det er gode løsninger for å monitorere containere, men også verktøy vi har erfaring med fra før.

7.4.1 System Center Operations Manager



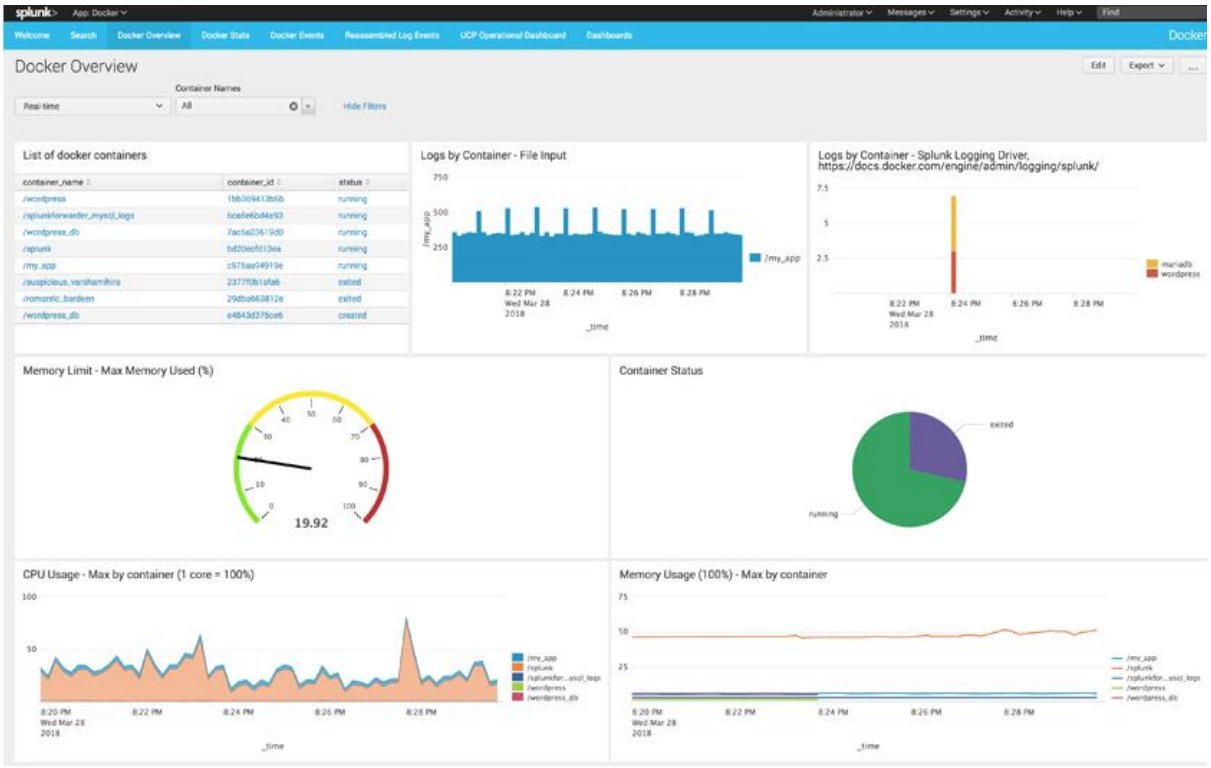
Figur 23: SCOM

System Center Operations Manager (SCOM) er et verktøy for overvåking av IT-miljøets infrastruktur. Med SCOM kan man sikre en fleksibel og effektiv infrastruktur, som i sin tur vil styrke organisasjonen. SCOM muliggjør overvåking av ytelse og statusinformasjon for programvare. Verktøyet har også funksjoner for å motta varsler, for eksempel hvis systemet identifiserer et sikkerhetsproblem.

SCOM er ett av flere verktøy som NHN bruker i sitt daglige virke. SCOM har sin styrke i infrastruktur monitorering, og er ikke like godt til å monitorere helse på en applikasjon. SCOM er heller ikke bygget for å monitorere containere, og det finnes få management packs for å monitorere slike miljøer. Det finnes unntak, men disse management packs-ene fungerer bare on premise, og har som oftest helt basis funksjonalitet. Siden vi ønsker å monitorere Windows containere, er ikke SCOM noe vi ønsker å gå videre med.

Vi ser det også som et problem med alarmering på noder og pods som slettes og startes vilkårlig. Det vil føre til mye alarmering og støy for, og mange alarmer.

7.4.2 Splunk

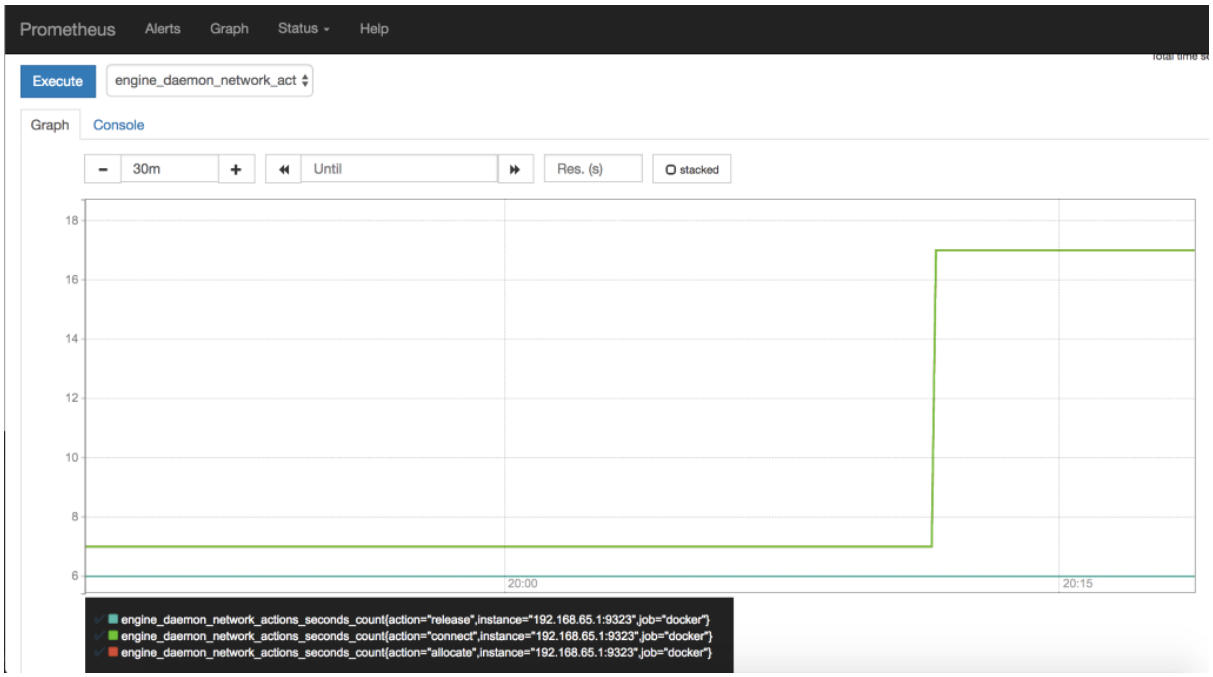


Figur 24: Splunk (Splunk 2019b)

Splunk er en web-basert tjeneste som analyserer maskingenererte data. Splunk blir ofte kalt “Google for logfiles”. Du kan laste inn loggfiler og lignende rådata fra hvilken som helst kilde, og så får du hjelp til å hente ut verdifull informasjon som ligger lagret i disse kildene. Splunk henter inn, lagrer og indekserer enorme mengder data for søk og analyse av informasjonen i sanntid. Dette innebærer at man for første gang kan bruke den enorme mengden logger som genereres i IT-systemer til storskala overvåkning, og samtidig hente ut verdifull informasjon ved hjelp av Big Data-funksjonalitet. Denne type data kan for eksempel være loggfiler fra anti-virusprogrammer, antall spørringer på en webside, og se etter kjente feilkoder på en tjeneste og koble dette opp mot eksterne aktører.

Splunk er et verktøy som brukes av NHN til å monitorere flere av deres mest kritiske tjenester. Siden noen av prosjektgruppens medlemmer har erfaring med å jobbe med Splunk er det ønskelig at man ser etter at andre alternativer for å tilegne seg mer kunnskap. Prosjektgruppen vil allikevel anbefale NHN å se på Splunk til å monitorere et container-basert miljø. Vi tror at det vil passe utmerket til dette, men vi ønsker som sagt å bruke bachelorperioden på å lære oss nye verktøy.

7.4.3 Prometheus m/influxDB og Grafana

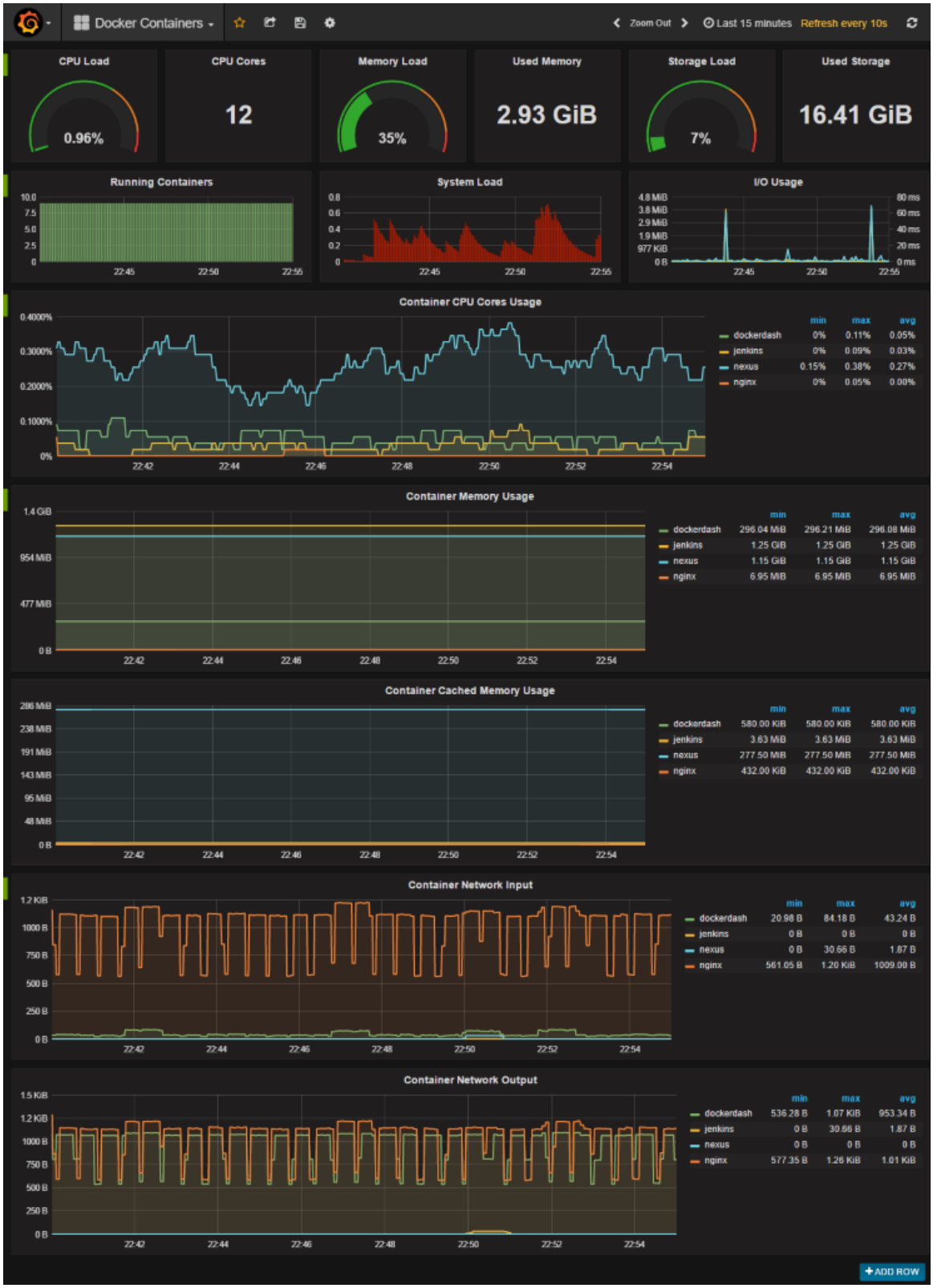


Figur 26: Prometheus (Docker 2019c)

Prometheus med bruk av influxDB og Grafana for å visualisere dataene er en annen løsning prosjektgruppen har sett på. Prometheus er et open source-program skrevet i Go som brukes til å registrere sanntids-beregninger i en tidsseriedatabase (som gir høy dimensjonalitet) bygget ved hjelp av en HTTP-pull metode, med fleksible spørringer og sanntidsvarsling.

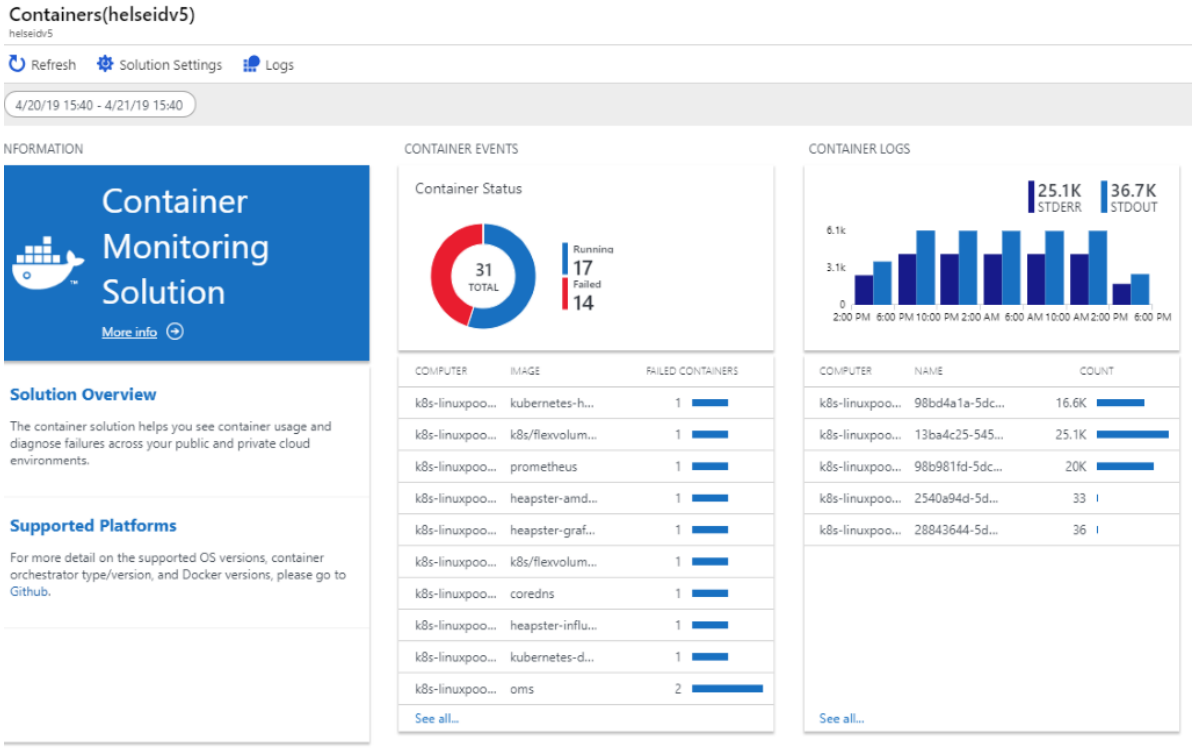
Prometheus virker interessant og selv om mange på NHN kjenner til verktøyet er det i dag ikke tatt i bruk. Allikevel føler vi at det er for mange deler til å få dette til å fungere som vi ønsker. Hadde vi hatt mer tid, eller at fokuset på oppgaven var mer knyttet opp til hvordan man monitorerer et kubernetes cluster på en god måte. Så kunne vi gått videre med dette alternativet. Men etter å ha testet Prometheus et par dager, ser vi at det vil ta for lang tid å få til å fungere slik vi ønsker.

Som vist på bildet over, så er Prometheus ganske spartansk hvis man ikke bruker andre programmer som for eksempel Grafana. Ved bruk av Grafana får man laget mer ryddige og forståelige dashboards, mens som nevnt tidligere, er ikke hovedfokuset i denne oppgaven å se på og utforske alle mulighetene med monitorering av et Kubernetes cluster. Så vi velger å ikke gå videre med dette verktøyet basert på testing vi har gjort.



Figur 27: Grafana (Github, 2019h)

7.4.4 Azure Monitor



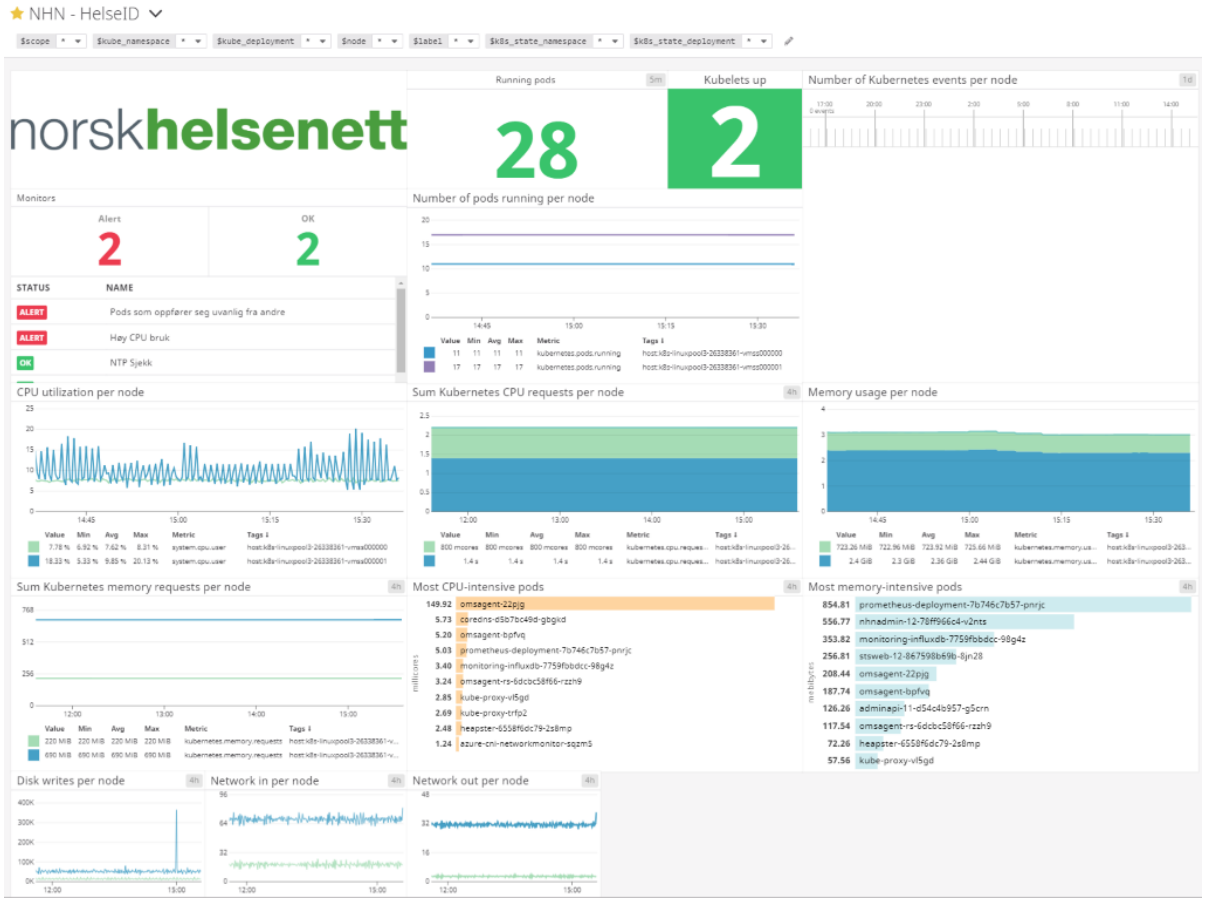
Figur 28: Azure Monitor

Azure Monitor er et annet verktøy som vi kan ta i bruk, spesielt med tanke på at vårt Kubernetes miljø kjører i Azure. Azure Monitor er en omfattende løsning for innsamling, analyse og handling av data. Det hjelper deg å forstå hvordan programmene dine utfører og proaktivt identifiserer problemer som påvirker dem og ressursene de er avhengige av.

Alle data samlet av Azure Monitor passer inn i en av to grunnleggende typer, metrics og logging. Metrics er numeriske verdier som beskriver noe aspekt av et system på et bestemt tidspunkt. Logger inneholder forskjellige typer data som er organisert i ulike sett med egenskaper for hver type.

Prosjektgruppen brukte vel en uke å se på forskjellige varianter av Azure sin container-baserte monitoreringsløsning med Log Analytics. Etter å brukt mye tid, kommer vi frem til konklusjonen at pods som henter inn metrics for OMS tar mye CPU kraft fra clusteret vårt, og vi sliter samtidig med å få inn metrics som har nytteverdi. Vi får heller ikke inn Windows container metrics. Applikasjonen er også lite intuitiv i bruk, og det er vanskelig å lage gode skreddersydde dashboards sammenlignet med Splunk og Datadog.

7.4.5 Datadog



Figur 29: Datadog dashboard eksempel

Datadog er kanskje for mange et nytt monitoreringsverktøy. Datadog er ment for å monitorere skybaserte tjeneste som for eksempel ligger i Azure eller AWS. Datadog er et overvåkings- og analyseverktøy som kan brukes til å bestemme ytelsesstatistikk, samt hendelsesovervåking for infrastruktur og skytjenester. Programvaren kan overvåke tjenester som servere, databaser og verktøy.

Datadog bruker en Go-basert agent og backend er laget av Apache Cassandra, PostgreSQL og Kafka. Et REST API brukes til å tillate Datadog å integrere med mange tjenester, verktøy og programmeringsspråk. Datadog har i dag over 250 integrasjoner mot forskjellige tjenester og services. Deriblant Kubernetes. Datadog har blitt omtalt som en moderne vri på monitorering, der det er enkelt å komme i gang, uten alt for mye jobb.

Brukergrensesnittet inneholder dashboards som kan vise grafer som består av flere datakilder i sanntid. Datadog kan også sende brukerne varsler om ytelsesproblemer og lignende. Brukerne blir varslet via hjelpemidler som e-post eller Slack for å nevne noen.

Datadog er det verktøyet vi bestemte oss for å ta med videre i testingen. Vi er

imponert over hvor skalerbart det er, og hvor enkelt det er å lage gode dashboards. Datadog er en betalingsapp, men prosjektgruppen lager prøvekontoer som varer ut bachelorperioden. Lisenskostnadene til Datadog er ikke tatt med i det totale kostnadsbildet. Dette da vi tror NHN kan bruke et av de verktøyene de allerede bruker i dag (og som de har lisens på). Datadog har en meget god integrasjon mot Kubernetes, som gjør at vi kan hente ut kube state-metrics uten for mye trøbbel. Dette vil igjen føre til at vil få flere metrics enn standard metrics som går på CPU, minnebruk etc.

8. Design av pilot

Vår pilot vil bestå av å konvertere en Windows applikasjon, kalt HelseID, som per dags dato driftes i produksjon av NHN med en tradisjonell driftsmodell over til å kunne kjøres i containere på et Kubernetes cluster. Piloten vil implementere et Kubernetes cluster i et testmiljø i Azure for å kjøre HelseID i containere. Vi vil i tillegg i dette miljøet teste skalering (både manuell og automatisk), lastbalansering, utrulling av oppdateringer, monitorering og andre relevante saker. En database blir også opprettet for bruk av HelseID.

HelseID er en applikasjon som NHN antar kommer til å ha en enorm vekst fremover, vi vil derfor legge stor vekt på mulighetene rundt skalering. Spesielt automatisk skalering og hvilke fordeler dette kommer til å ha i forhold til måten de drifter applikasjonen på i dag.

Programvare og verktøy teoretisk beskrevet i denne rapporten vil bli benyttet for å gjennomføre piloten.
Sammenligninger med hvordan det er i produksjon for HelseID i dag, vil gjøres der det er muligheter for det. For eksempel nedetid ved oppdatering, tid før nødvendig skalering er gjennomført med mer.

8.1 Testmiljø

8.1.1 Valg av testmiljø

Det er flere typer infrastruktur man kan bruke for å distribuere tjenester. De faller inn i fire grunnmodeller, som kan blandes og tilpasses etter behov;

1. **On-premise**
   Dette har vært den dominerende modellen i mange år. Organisasjoner opprettholder eget utstyr og programvare i datasenter der de har full kontroll. Dette kan være nødvendig for bedrifter med stor grad av regulering, da behovet for tett kontroll av data og dokumenter er tydelig. Det kan også være lovpålagt at data ikke kan lagres utenfor bedriftens systemer. Denne løsningen kommer med betydelige kostnader og kapitalutgift da det må kjøpes fysisk maskinvare som også må vedlikeholdes lokalt.

2. **Public Cloud / Offentlig Sky**
   Med offentlig sky har man tilgang til ressurser som prosessorkraft, minne, operativsystem, applikasjoner og lagring over det offentlige internett. Det er mange fordeler med å bruke en offentlig skyløsning, som skalerbarhet, fleksibilitet og enkel administrasjon. Fakturering er vanligvis basert på bruk, slik at man kun betaler for de ressursene som man bruker. Dette i forhold til

en on-premise løsning der du betaler for cpu/minne/lagring når du kjøper det fysiske utstyret.

3. **Private Cloud / Privat sky**
   Noen bedrifter, for eksempel NHN, har behov for å holde ressurser lokalt for kontroll eller på grunn av regulasjoner. En privat sky gir samme fleksibilitet, skalerbarhet og muligheter for enkel administrasjon som en offentlig sky samt mer kontroll og data kan bedre fysisk sikres enn i en offentlig sky. Private skyer kan bygges ut ifra eksisterende arkitektur eller man kan bruke lisensiert arkitektur fra offentlige skyleverandører, det blir en versjon av deres eksisterende tjenester bare i et sikkert miljø.

4. **Hybrid Cloud / Hybrid sky**
   Offentlig og privat sky kan kombineres til en hybrid sky. Med denne arkitekturen får man en god kombinasjon av kontroll og fleksibilitet som kan tilpasses bedriftens situasjon. De applikasjonene og data man må ha tett kontroll på eller er regulert at ikke kan opp i en offentlig sky kan fortsette å bruke lokale ressurser, mens andre applikasjoner og data som ikke er så kraftig regulert kan bruke såkalt “cloud bursting”, der overbelastning går til skyen der en duplikat eller kompatibel sky-applikasjon tar over etter behov. Med dette kan man unngå behovet for on-premise infrastruktur som ikke ofte brukes og bare kjører på tomgang og venter på at de ekstra ressursene trengs.

Når vi starter prosjektet fikk vi to valgmuligheter for arkitektur.

1. Et par bare metal servere on-premise fra NHN - On-premise
2. Azure subscription fra NTNU - Offentlig sky

Når vi skal vurdere hvilken arkitektur som passer dette prosjektet best, med tanke på at vi vet at vi skulle bruke Kubernetes som container orkestreringsverktøy, er det som veier mest i fordel for Azure (offentlig sky) den forenklede administrasjonen. Fokuset for dette prosjektet er å undersøke de nye mulighetene for å kjøre Windows applikasjoner i containere med de oppdateringene som kom med Windows Server 2019, samt hvordan det fungerer å konvertere en eksisterende NHN Windows applikasjon til å kjøre i containere. Vi vil derfor ikke bruke mer tid enn nødvendig på installasjon og konfigurasjon av miljøet vi bruker for å teste ut disse tingene. Når vi undersøkte mulighetene for å kjøre Kubernetes i Azure fant vi AKS og AKS Engine, lovende verktøy som kunne lette arbeidet med installasjon og administrasjon. Med Azure blir også ting som nettverksadministrasjon, lastbalansering osv. tatt litt ut av våre hender og vi får mer tid til å fokusere på målet med prosjektet.

I tillegg er skalering en viktig del av dette for NHN og det fungerer best i en skyløsning, der man kan kjøre opp nye virtuelle maskiner og tildele ressurser enklere

og bedre enn med en on-premise løsning. Det er en generell akseptert sannhet at du får mest ut av fordelene med å kjøre applikasjoner, i containere, i skyen. Det er derfor naturlig at vi valgte å bruke Azure som miljø for utrulling av Kubernetes.

Testmiljø blir opprettet i Azure ved bruk at et Pay-As-You-Go abonnement prosjektgruppen har fått tilgang til av NTNU.
Følgende ressurser vil bli opprettet og konfigurert delvis av verktøyet AKS Engine og delvis av prosjektgruppen:

- Virtual machine scale sets – Ett for Windows maskiner, ett for Linux
- Virtuelle maskiner med Windows og Linux
- Load balancer
- Virtuelt nettverk, nettverksgrensesnitt og nettverkssikkerhet gruppe

8.1.2 Hvorfor vi valgte AKS Engine

Det mest åpenbare valget for implementasjon av et Kubernetes cluster når man bruker Azure kan tenkes å være AKS. Men det er flere grunner til at dette ikke er tilfellet for dette prosjektet.

Når vi testet Virtual Kubelet på et AKS cluster i forstudiet møtte vi på en del problemer med å få hentet ut metrics fra Windows containere som kjørte på en slik Virtual Kubelet. Og siden det ikke faktisk er en host, men at den bare later som den er det, er det ikke enkelt å hente “host” metrics fra en slik Virtual Kubelet. Uten gode muligheter for å hente metrics blir monitorering vanskelig. Skalering blir også et problem, da dette er basert på metrics som CPU-bruk.
Vi spurte utviklerne om nettopp dette med monitorering, og fikk følgende som svar:

Figur 30: Svar på Github ticket (Github, 2019g)

Det må også nevnes at det blir presisert i dokumentasjonen til Virtual Kubelet at dette er eksperimentell programvare som ikke burde brukes i noe som i det hele tatt ligner på et produksjonsmiljø.

Det er også annonsert at Microsoft slutter å støtte ACI fra og med 31. Januar 2020 (Foley, M. J. 2019). De ønsker at veien fremover for container støtte i Azure er AKS. Men siden AKS per dags dato kun støtter Linux containere er ikke dette noe vi ville bruke i dette prosjektet.

Ettersom vi ønsker at prosjektet skal være så fremtidsrettet som mulig blir Virtual Kubelet og dermed AKS ikke noe vi ser mye nærmere på i utover det vi har gjort allerede.

AKS Engine blir dermed et naturlig valg, da vi har flere konfigurasjonsmuligheter og kan implementere et hybrid cluster med både Windows og Linux worker noder. Vi får fortsatt mange av fordelene med mindre arbeid å implementere og mindre arbeid med administrasjon som man får med AKS.

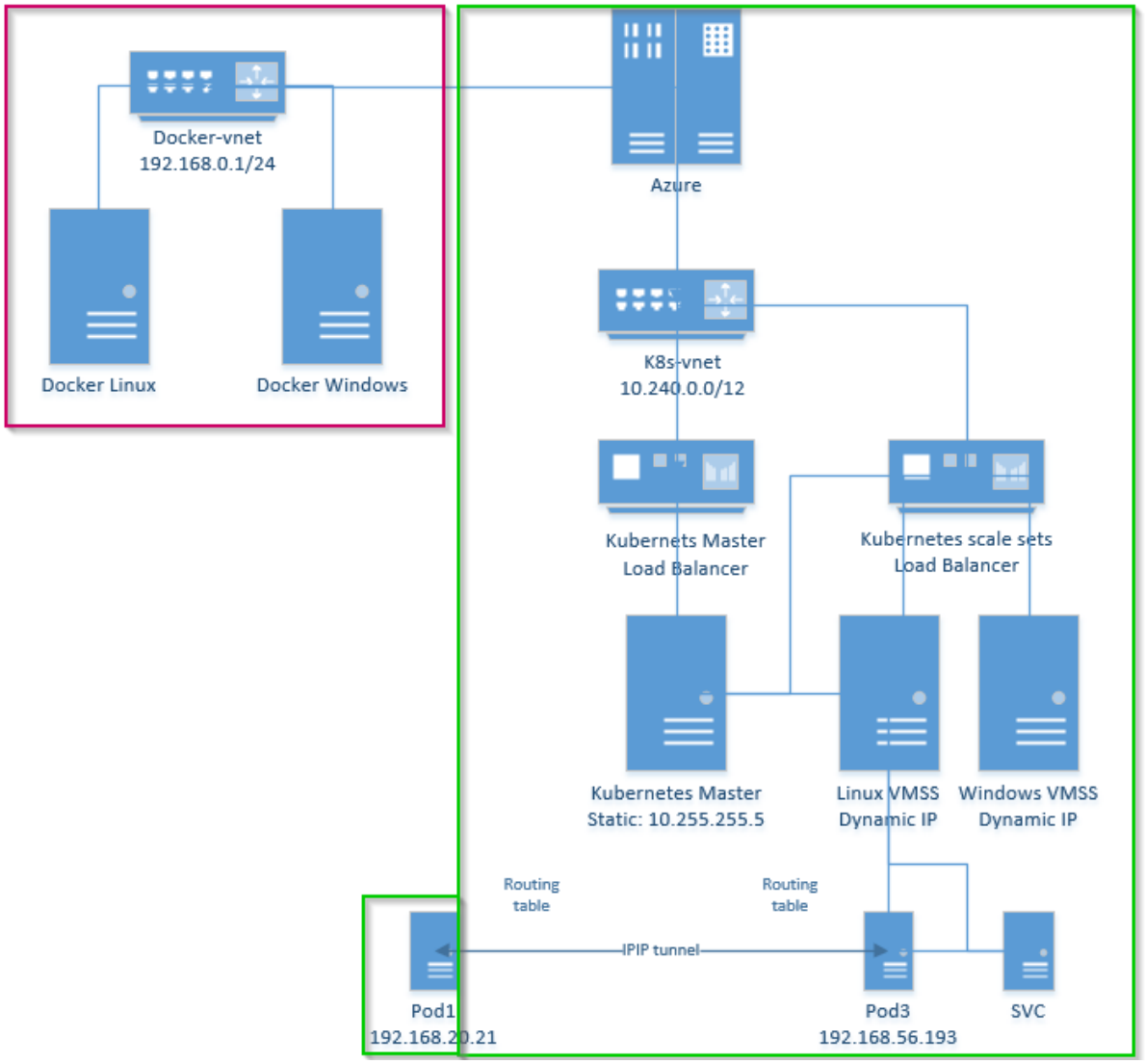
8.1.3 Nettverkstopologi

På bildet under viser vi hvordan nettverkstopologien vil bli til vår pilot. Til venstre med rød ramme rundt seg, har vi våre Docker maskiner som skal brukes til å bygge å konvertere HelseID.

Disse er på eget vNet med subnet IP 192.168.0.1./24.

Til høyre i bildet med grønn ramme rundt seg har vi vårt Kubernetes cluster. Her har vi et subnet på 10.240.0.0/12, vi har også her to lastbalanserer der den ene snakker direkte med Kubernetes masteren på statisk IP 10.255.255.5. Vi har også en lastbalanserer for VMSS-ene våre. Denne lastbalansereren kommuniserer også med Kubernetes for å sette opp regler.

Videre har vi Pods som kjører på VMSS-ene. Pods på tvers av noder kommuniserer gjennom en IPIP tunnel med et sett routing regler.



Figur 31: Nettverkstopologi for vårt Azure testmiljø

8.2 Docker

Docker vil i hovedsak bli brukt til å konvertere HelseID til kjørbare Docker image-er og laste opp disse til et privat repository i Docker Hub for bruk i testmiljøet.
To virtuelle maskiner provisjoneres av prosjektgruppen for bruk av Docker:

- 1 stk Ubuntu Server 18.04 LTS
- 1 stk Windows Server 2019, versjon 1809

Det er mulig det må lages Docker image-er av både typen Windows og Linux for å få HelseID oppe å gå og vi bruker derfor for enkelhets skyld begge typer maskiner og installerer siste stabile versjon av Docker på dem.
Servere blir oppdatert til siste versjon av all installert programvare og operativsystem.

Begge maskiner ment for Docker-bruk blir konfigurert med mulighet for fjernstyring. På Ubuntu-maskin blir inngående port 22 åpnet for SSH og på Windows-maskin blir port 3389 åpnet for RDP.

8.3 Microsoft SQL Express 2017 database

HelseID har noen avhengigheter som må på plass, en av disse er fire databaser. Vi vil derfor provisjonere følgende virtuelle maskin:

- 1 stk Windows Server 2019, versjon 1809

På denne maskinen installerer vi siste stabile versjon av Microsoft SQL Express 2017. SQL Server Management Studio blir også installert for administrasjon av databasene og det blir brukt program mottatt av NHN til å importere de nødvendige databasene. Seed-script for å fylle databasene med nødvendig data blir beskrevet i driftsrapporten (Dønnem og Roksvaag, 2019a) da dette er en del av prosessen med å konvertere HelseID.

8.4 AKS Engine

Verktøyet AKS Engine blir brukt til å implementere et Kubernetes cluster i vårt Azure miljø. Siste stabile versjon av AKS Engine blir lastet ned, installert og konfigurert lokalt på de personlige maskinene til prosjektgruppen. Azure CLI blir også installert på samme måte, da det er nødvendig for vellykket bruk av AKS Engine. Maler for konfigurasjon av Kubernetes cluster-utrulling blir lastet ned lokalt på maskinene. Nødvendig konfigurasjon blir gjort lokalt på maskinene, dette inkluderer enkle ting som filmappe-oppsett og mer kompliserte ting som utfylling av mal for konfigurasjon av Kubernetes clusteret.

8.5 Kubernetes

Kubernetes blir brukt som orkestreringsverktøy for de containerne vi kommer til å bruke i denne piloten. Dette inkluderer containere som inneholder applikasjonen HelseID samt containere for cluster autoskalering, monitorering, generell Kubernetes administrasjon osv. Kubernetes vil også bli brukt til å lage regler for automatisk skalering på pod/container-nivå, eksponere containere til internett via en lastbalanserer med mer.

Gjennom konfigurasjonsfilene som vi forer til AKS Engine vil det bli opprettet følgende virtuelle maskiner som del av Kubernetes clusteret:

- 1 stk Ubuntu Server
- 2 stk Ubuntu Server plassert i Linux virtual machine scale set
- 2 stk Windows Server plassert i Windows virtual machine scale set

Alle disse maskinene blir provisjonert av AKS Engine på vegne av prosjektgruppen, ut ifra hvordan vi fyller ut konfigurasjonsfilene.

Den Ubuntu maskinen som blir laget utenfor scale set er Kubernetes master noden. En master node i et Kubernetes cluster må per dags dato være en Linux-maskin, det kan ikke være en Windows maskin.

Master noden har 2 vCPU (virtuelle CPU-er) og 8GB ram. Masteren bruker som regel ikke å behandle veldig mye last, så det er nok for den.
Hver av worker nodene har 4 vCPU og 16 GB ram. Worker nodene kan få en del mer å arbeide med enn master noden, så vi gir dem derfor litt mer kraft.

Grunnen til at vi forteller AKS Engine at det skal lage virtual machine scale sets er fordi vi trenger det til å teste skalering av noder i Kubernetes clusteret på måten som er planlagt. Addonen vi bruker for cluster skalering på node-nivå krever at worker nodene ligger i scale sets. Grunnen til at vi har to maskiner per scale set er for å kunne teste lastbalansering på containernivå.

8.6 Datadog

Datadog vil bli brukt til monitorering. Datadog blir implementert som kjørende containere i vårt Kubernetes cluster.

9. Kilder

Acetozi, J. (2017): *Kubernetes Master Components: Etcd, API Server, Controller Manager, and Scheduler*
Tilgjengelig fra: https://medium.com/jorgeacetozi/kubernetes-master-components-etcd-api-server-controller-manager-and-scheduler-3a0179fc8186
(Hentet 18.03.2019)

Cogan, S. (2019): *Windows Containers and Azure*
Tilgjengelig fra: https://samcogan.com/windows-containers-and-azure/
(Hentet 01.04.2019)

Davé, A. (2018): *Monitoring Kubernetes in Production: How To Guide (Part 1 og 5)*
Tilgjengelig fra: https://dzone.com/articles/monitoring-kubernetes-in-production-how-to-guide-p
(Hentet 01.04.2019)

Docker (2019a): *Docker Engine*
Tilgjengelig fra: https://www.docker.com/products/docker-engine
(Hentet 01.04.2019)

Docker (2019b): *What is a container?*
Tilgjengelig fra: https://www.docker.com/resources/what-container
(Hentet 01.04.2019)

Docker (2019c): *Collect Docker metrics with Prometheus*
Tilgjengelig fra: https://docs.docker.com/config/thirdparty/prometheus/
(Hentet 10.04.2019)

Dønnem og Roksvaag (2019a): *Driftsrapport*

Dønnem og Roksvaag (2019b): *Forstudierapport*

Dønnem og Roksvaag (2019c): *Sluttrapport*

Foley, M. J (2019): *Microsoft to stop supporting its Azure Container Service in January 2020*
Tilgjengelig fra: https://www.zdnet.com/article/microsoft-to-stop-supporting-its-azure-container-service-in-january-2020/
(Hentet 20.04.2019)

Foulds, I. (2018): *Azure-docs Github Issue #11616*
Tilgjengelig fra: https://github.com/MicrosoftDocs/azure-docs/issues/11616
(Hentet 20.04.2019)

Github (2019a): *Frequently asked questions about Azure Kubernetes Service (AKS)*
Tilgjengelig fra: https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/faq.md

(Hentet 01.04.2019)

Github (2019b): *Use Virtual Kubelet with Azure Kubernetes Service (AKS)*
Tilgjengelig fra: https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/virtual-kubelet.md
(Hentet 18.03.2019)

Github (2019c): *Virtual Kubelet; How it works*
Tilgjengelig fra: https://github.com/virtual-kubelet/virtual-kubelet#how-it-works
(Hentet 18.03.2019)

Github (2019d): *Deploy a Kubernetes Cluster*
Tilgjengelig fra: https://github.com/Azure/aks-engine/blob/master/docs/tutorials/deploy.md
(Hentet 18.03.2019)

Github (2019e): *AKS Engine; Architecture*
Tilgjengelig fra: https://github.com/Azure/aks-engine/blob/master/docs/topics/architecture.md
(Hentet 18.03.2019)

Github (2019f): *Cluster Autoscaler on Azure*
Tilgjengelig fra: https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/azure/README.md
(Hentet 25.03.2019)

Github (2019g): *No Metrics from Windows Containers*
Tilgjengelig fra: https://github.com/virtual-kubelet/virtual-kubelet/issues/522
(Hentet 01.04.2019)

Github (2019h): *Dockprom*
Tilgjengelig fra: https://github.com/stefanprodan/dockprom
(Hentet 01.04.2019)

Ivancza, K. (2018): *What is Kubernetes & How to Get Started With It*
Tilgjengelig fra: https://blog.risingstack.com/what-is-kubernetes-how-to-get-started/
(Hentet 11.03.2019)

Kubernetes (2019a): *Picking the Right Solution*
Tilgjengelig fra: https://kubernetes.io/docs/setup/pick-right-solution/
(Hentet 22.04.2019)

Kubernetes (2019b): *Pull an Image from a Private Registry*
Tilgjengelig fra: https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/
(Hentet 22.04.2019)

Kubernetes (2019c): *What is Kubernetes*
Tilgjengelig fra: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
(Hentet 08.03.2019)

Microsoft (2018): *Microsoft announces two new datacentre regions in Norway*
Tilgjengelig fra: https://news.microsoft.com/europe/2018/06/20/microsoft-announces-two-new-datacentre-regions-in-norway/
(Hentet 22.04.2019)

Microsoft (2019a): *Azure Monitor overview*
Tilgjengelig fra: https://docs.microsoft.com/en-us/azure/azure-monitor/overview
(Hentet 18.03.2019)

Microsoft (2019b): *Azure Monitor for containers overview*
Tilgjengelig fra: https://docs.microsoft.com/en-us/azure/azure-monitor/insights/container-insights-overview
(Hentet 18.03.2019)

Microsoft (2019c): *Azure Kubernetes Service (AKS)*
Tilgjengelig fra: https://azure.microsoft.com/en-us/services/kubernetes-service/
(Hentet 11.03.2019)

Microsoft (2019d): *Intro to Azure Kubernetes Service (AKS)*
Tilgjengelig fra: https://docs.microsoft.com/en-us/azure/aks/intro-kubernetes
(Hentet 11.03.2019)

Microsoft (2019e): *Azure Kubernetes Service -- Windows Containers Preview*
Tilgjengelig fra: https://aka.ms/aks-windows-preview
(Hentet 24.04.2019)

Microsoft (2019f): *Windows containers now supported in Kubernetes*
Tilgjengelig fra: https://cloudblogs.microsoft.com/opensource/2019/03/25/windows-server-containers-now-supported-kubernetes/
(Hentet 24.04.2019)

Norsk Helsenett SF (2019a): *Norsk Helsenett*
Tilgjengelig fra: https://nhn.no/helseid
(Hentet 01.03 2019)

Norsk Helsenett SF (2019b): *HelseID; Grunnleggende kunnskap*
Tilgjengelig fra: https://nhn.no/helseid/grunnleggende-kunnskap/
(Hentet 01.03.2019)

Norsk Helsenett SF (2019c): *HelseID; Teknisk Dokumentasjon*
Tilgjengelig fra: https://nhn.no/helseid/teknisk-dokumentasjon/
(Hentet 01.03.2019)

Norsk Helsenett SF (2019d): *HelseID; Arkitektur*
Tilgjengelig fra: https://nhn.no/helseid/grunnleggende-kunnskap/arkitektur/
(Hentet 01.03.2019)

Palmer, M. (2018): *Kubernetes Ingress with Nginx Example*
Tilgjengelig fra: https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html
(Hentet 10.04.2019)

Pantheon (2019): *Why Containers?*
Tilgjengelig fra: https://pantheon.io/platform/why-containers
(Hentet 18.03.2019)

Rainey, R. (2016): *An Introduction to the Azure Resource Manager (ARM)*
Tilgjengelig fra: http://rickrainey.com/2016/01/19/an-introduction-to-the-azure-resource-manager-arm/
(Hentet 26.04.2019)

Scott, J. (2018): *Cloud vs. On-Premises – What Are the Best Options for Deploying Microservices with Containers?*
Tilgjengelig fra: https://mapr.com/blog/cloud-vs-on-premises-what-are-the-best-options-for-deploying-microservices-with-containers/
(Hentet 11.03.2019)

Splunk (2019a): *A Beginner's Guide to Kubernetes Monitoring*
Tilgjengelig fra: https://www.splunk.com/en_us/form/a-beginners-guide-to-kubernetes-monitoring.html#
(Hentet 15.03.2019)

Splunk (2019b): *Container Monitoring*
Tilgjengelig fra: https://www.splunk.com/en_us/it-operations/container-monitoring.html
(Hentet 15.03.2019)

Bruk av containere i NHN

Driftsrapport

Av: Henrik Roksvaag og Ole Valla Dønnem

NTNU
Kunnskap for en bedre verden

Revisjonshistorie

Dato	Versjon	Beskrivelse	Forfattere
30.04.19	1.0	Første utkast	Henrik Roksvaag & Ole Valla Dønnem
05.05.19	1.1	Lagt til kapittel 11. Sikkerhet i Kubernetes	Henrik Roksvaag & Ole Valla Dønnem
10.05.19	1.2	Implementert endringer basert på tilbakemelding fra veiledere	Henrik Roksvaag & Ole Valla Dønnem
19.05.19	1.3	Endelig versjon Ferdigstilling basert på endelig tilbakemelding fra veiledere	Henrik Roksvaag & Ole Valla Dønnem

Innholdsfortegnelse - Driftsrapport

1. Introduksjon

1.1 Hensikten med dokumentet

Hensikten med dette dokumentet er å gjennomføre piloten som står beskrevet i kapittel “8. Design av pilot” i Designrapporten (Dønnem og Roksvaag, 2019a). Vi vil beskrive i detalj hvordan vi har løst de forskjellige oppgavene.

1.1 Oversikt over innholdet

Kapittel 2 tar for seg piloten. Vi vil her beskrive hva vi ønsker å oppnå og hvordan vi skal klare det. Vi vil nevne kort hva som skal installeres av verktøy og programvare, dette blir et sammendrag av det som står i kapittel “8. Design av pilot” i Designrapporten (Dønnem og Roksvaag, 2019a).

Kapittel 3 tar for seg installasjon av Docker på Windows Server 2019. Vi vil vise hvordan vi installerer Docker, til bruk av bygging av Docker images, og vi viser hvordan vi oppretter repositories på Docker Hub.

Kapittel 4 inneholder installasjon og konfigurasjon av SQL Express 2017. Vi viser hvordan vi oppretter databasene vi trenger å ha for at applikasjonen skal fungere tilfredsstillende og hvordan vi fyller disse databasene med data samt hvordan vi genererer en test bruker.

Kapittel 5 inneholder hvordan vi tar i bruk AKS Engine for å lage vårt Kubernetes cluster i Azure. Vi går her i dybden og ser på hvordan vi oppretter SSH nøkler, hvordan vi konfigurerer maler, og hvordan vi lager service principals, velger maskiner etc.

Kapittel 6 er et forholdsvis kort kapittel der vi får på plass noen ting vi må gjøre i Azure og Kubernetes etter at AKS Engine har gjort seg ferdig med å opprette vårt cluster.

Kapittel 7 tar for seg bygging og konvertering av selve applikasjon HelseID. Vi ser på kildekoden og hvordan vi går frem for å bygge et docker image av de delene av applikasjonene vi behøver.

Kapittel 8 tar for seg hvordan de tre delene av applikasjonene rulles ut og konfigureres i Kubernetes, via YAML-filer.

Kapittel 9 tar for seg hvordan vi skalerer en pod eller node basert på last. Vi ser både på manuell skalering og automatisk skalering av hele clusteret.

Kapittel 10 inneholder informasjon om hvordan vi går frem for å monitorere vårt Kubernetes cluster. Vi ser her på metrics data, loggdata og alarmering. Vi ser også på design av dashboards.

Kapittel 11 inneholder tiltak vi har tatt i bruk for å sikre vårt Kubernetes cluster mot angrep. Vi har med noen eksempler på hvordan vi har sikret vårt cluster, og vi redegjør for de kapitlene i normen som vi mener er fulgt.

Kapittel 12 inneholder kildehenvisninger til de ressursene som er brukt i dette dokumentet.

2. Overordnet beskrivelse av pilot

Det vil i dette dokumentet være fokus på hvordan vi går frem for å implementere testmiljø og utføre tester som er beskrevet i Designrapportens kapittel “8. Design av pilot” (Dønnem og Roksvaag, 2019a). Vi vil også gjennomføre konvertering av HelseID og beskrive hvordan vi har satt opp monitorering.

For detaljert informasjon om hva de aktuelle ressursene og tjenestene som beskrives og implementeres i piloten er, samt hvorfor disse er valgt, henviser vi til Designrapporten (Dønnem og Roksvaag, 2019a).

Overordnet skal vi i piloten:

- Implementere et Kubernetes cluster i Azure med AKS Engine.
- Konvertere HelseID til å kjøre som containere.
- Rulle ut HelseID i vårt Kubernetes cluster som containere.
- Teste skalering av HelseID i Kubernetes.
- Sette opp monitorering av vårt Kubernetes cluster med Datadog.

For å få til dette vil vi gjøre følgende:

- **Installere og konfigurere Docker**
  Docker skal brukes til å konvertere HelseID.
- **Installere og konfigurere SQL Express**
  Trenger databaser til HelseID.
- **Installere og konfigurere AKS Engine**
  AKS Engine skal brukes til å implementere et Kubernetes cluster i Azure.
- **Konfigurasjon av Kubernetes**
  Kubernetes cluster må konfigureres slik at alle nødvendige verktøy som skal brukes videre i piloten, til for eksempel testing av skalering, fungerer som ønsket.
- **Implementasjon og konfigurasjon av Datadog**
  Datadog skal brukes til monitorering.

2.1 Ressursbehov

For å gjennomføre piloten trenger vi følgende ressurser:

- Tilgang til et Azure subscription
- Tilgang til kildekoden til HelseID

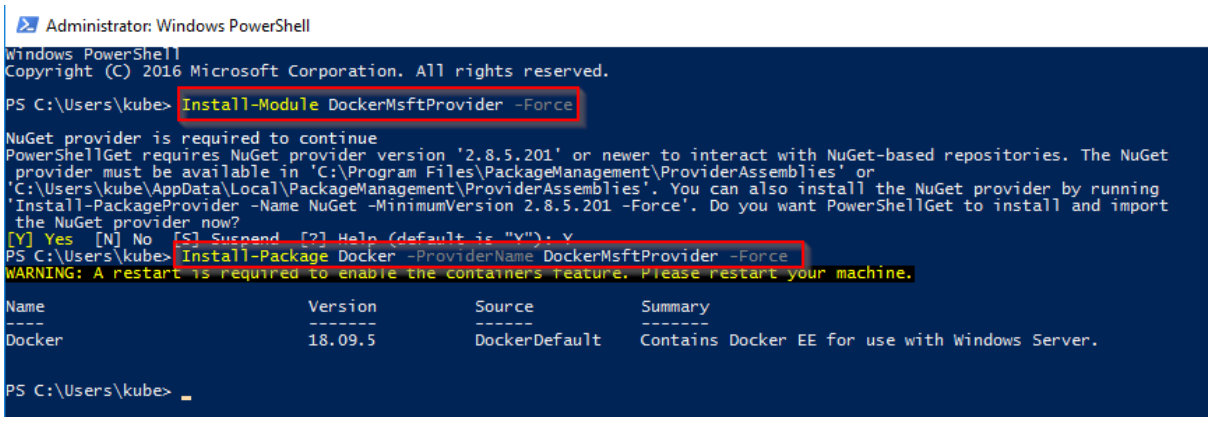
3. Installasjon av Docker

Etter å ha satt opp en Linux VM og en Windows VM i Azure til bruk for konvertering av HelseID, går vi videre med installasjon av Docker. Docker er det programmet vi skal ta i bruk for å konvertere HelseID over til noe som kan kjøre i Kubernetes.
Vi viser her fremgangsmåten for å installere Docker på Windows, fremgangsmåten er temmelig lik på Linux, så vi viser bare en av fremgangsmåtene.

Vi starter med å åpne Powershell med administratorrettigheter og kjører kommandoene:

```
Install-Module DockerMsftProvider -Force
```

```
Install-Package Docker -ProviderName DockerMsftProvider -Force
```



Figur 32: Installasjon Docker

Etter endt installasjon restarter vi maskinen.

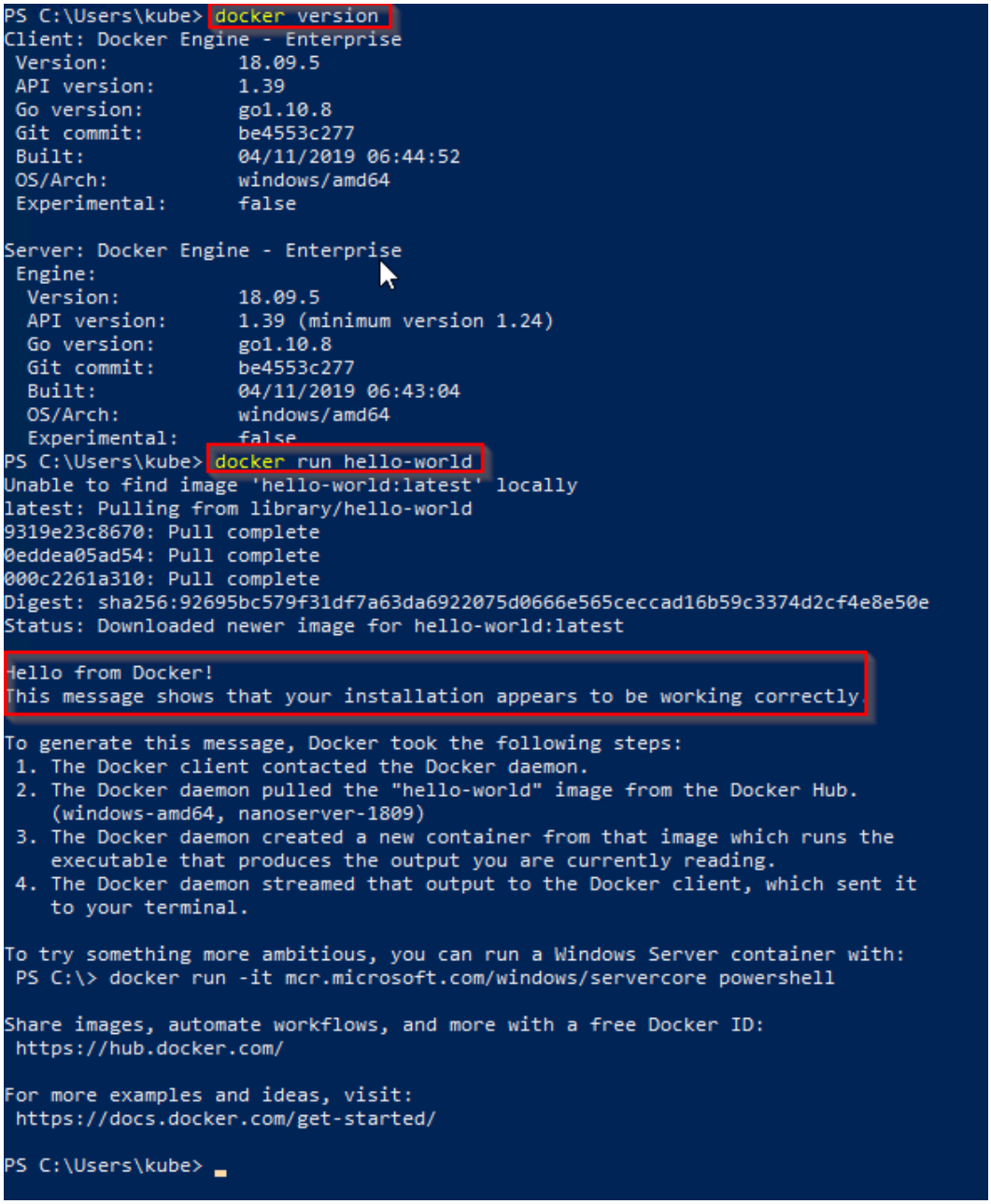
Etter at maskinen er restartet åpner vi Powershell igjen og kjører kommandoen `docker version` for å se at det finnes en versjon av docker på vår maskin, og at denne servicen faktisk kjører. Vi sjekker videre om Docker fungerer ved å kjøre ut et test image. Vi kjører kommandoen:

```
docker run hello-world
```

Et image kalt “hello-world” blir hentet fra Docker Hub. En container blir laget fra “hello-world”, denne kjører en executable som produserer teksten:

“Hello from Docker!
This message shows that your installation appears to be working correctly.”

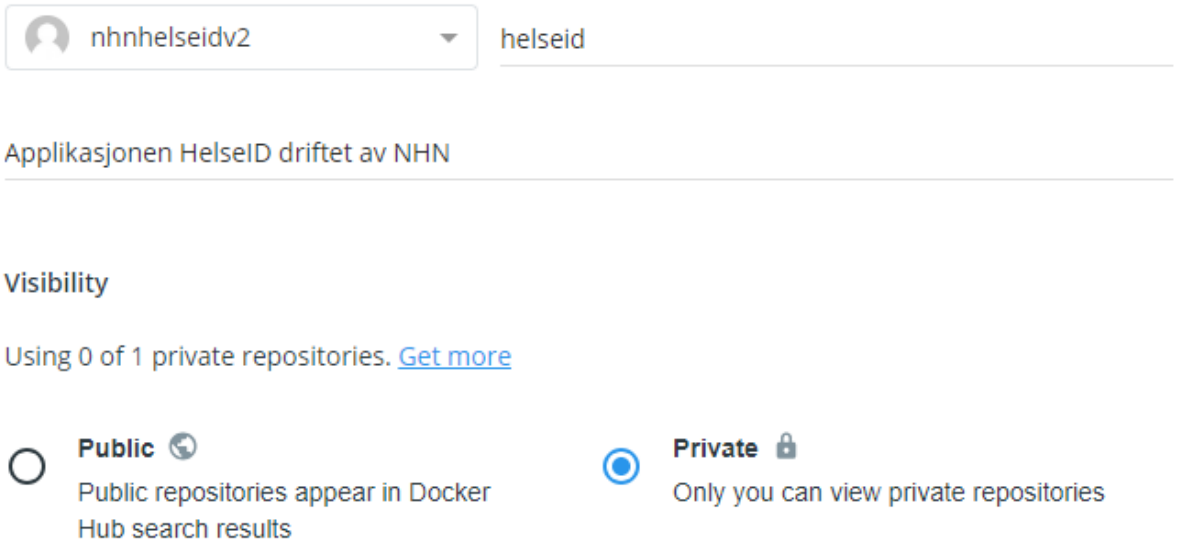
Docker er installert og fungerer. Outputen fra kommandoene vi kjørte vises i bildet under.



Figur 33: Docker hello world

3.1 Opprette repository på Docker Hub

Vi vil så opprette et private repository på Docker Hub. Vi oppretter en konto kalt nhnhelseidv2 og kaller repository-et vårt for Helseid. Med et private repository er det ingen andre som kan se innholdet vi lager.



Figur 34: Privat repository

Etter å ha laget dette bruker vi vår Docker maskin i Azure, og logger oss inn ved å bruke kommandoen

```
docker login
```

Som vi skrev fra installasjon av Docker hentet vi ned et hello-world docker image fra Dockerhub. Nå skal vi teste om vi kan få pushet dette til vårt repository.

Vi starter først med å tagge imaget ved å bruke følgende kommando. Her velger vi å tagge det med navnet “mintestapplikasjon”.

```
docker tag hello-world:nanoserver nhnhelseidv2/helseid:mintestappplikasjon
```

Etter å ha tagget det, kan det pushes til vårt repository. Vi bruker da kommandoen

```
docker push nhnhelseidv2/helseid:mintestappplikasjon
```

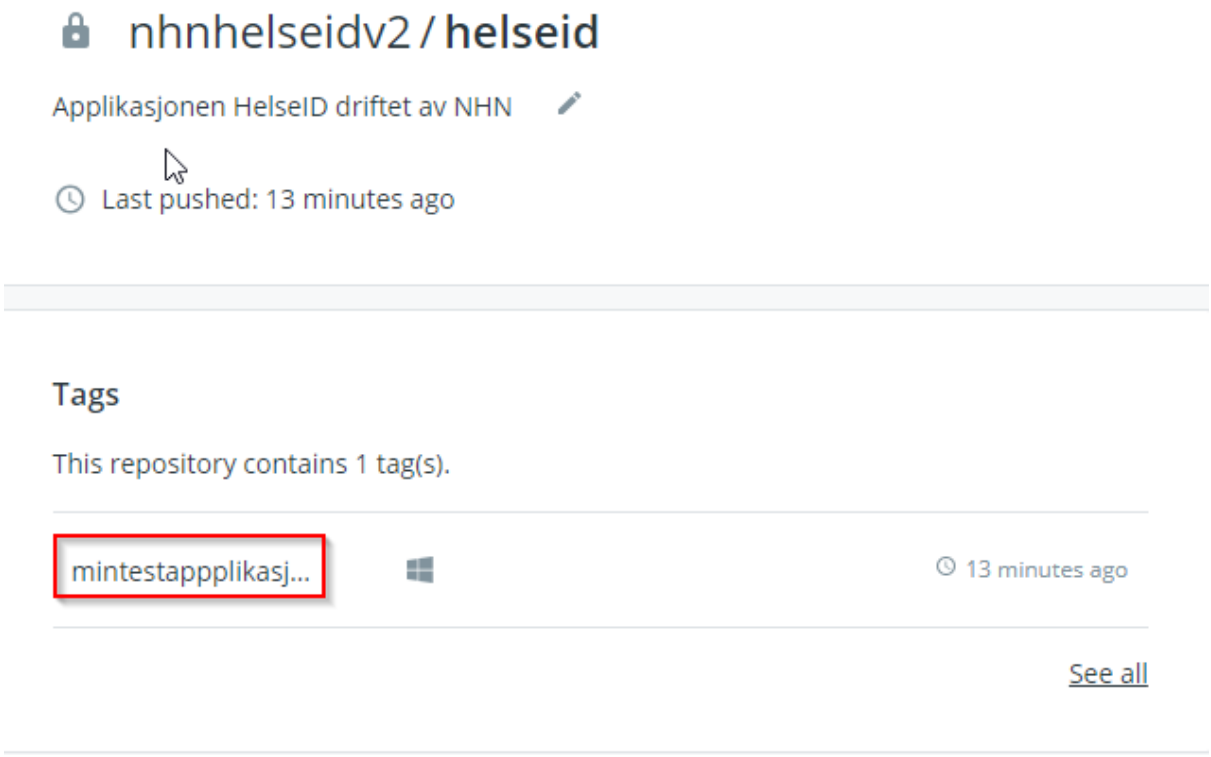
Outputen fra Powershell kan vi se under.

```
PS C:\Users\kube> docker push nhnhelseidv2/helseid:mintestappplikasjon
The push refers to repository [docker.io/nhnhelseidv2/helseid]
a473df95cad9: Mounted from library/hello-world
357cbc056c45: Mounted from library/hello-world
75ddd2c5f09c: Skipped foreign layer
37c182b75172: Skipped foreign layer
mintestappplikasjon: digest: sha256:dd9b7482975b66309507df1365b17ea38ecd5e79005eaa8a57136a60ef5e3cf5 size: 1358
PS C:\Users\kube>
```

Figur 35: Laste Docker image opp til privat repository

Til slutt kan vi se på Docker Hub hjemmesiden og se om vårt image ligger der. Fra bildet under ser vi at det gjør det, og vi er nå klar til å starte videre på videre

konfigurasjon av miljøet vårt. Vi skal nå installere Microsoft SQL Server 2017 Express.



Figur 36: Image i Docker repository

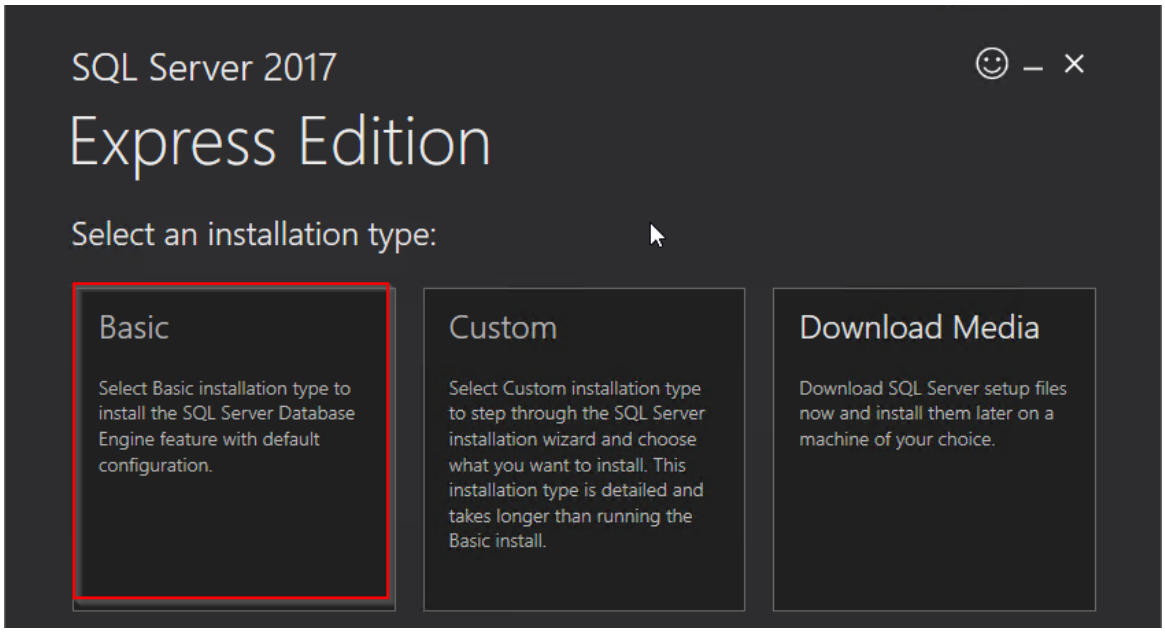
4. Installasjon av SQL

Vi valgte å bruke Sql Express 2017 som vår SQL database.

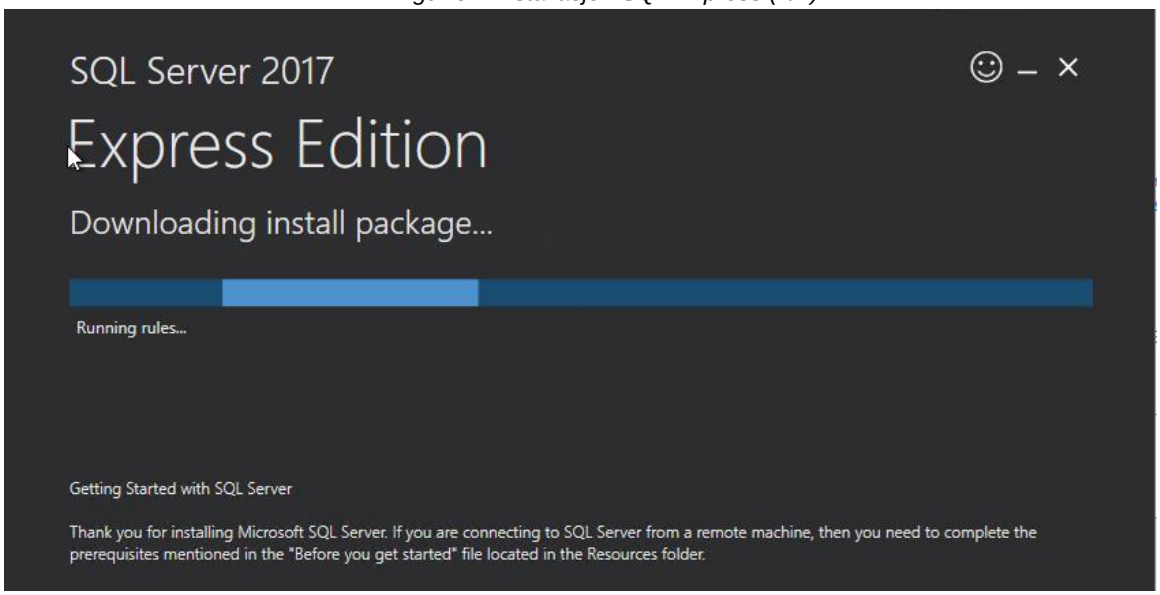
Vi velger å bruke samme VM som vi bruker til å konvertere HelseID applikasjonen, og ikke en egen separat VM. Databasene som trengs er ikke spesielt store og Docker bruker heller ikke noe spesielt av ressurser etter vi er ferdige med å konvertere HelseID.

4.1 Installasjon SQL Express 2017

Vi starter med å laste ned SQL Express 2017, og velger en basic installasjon. Etter et par minutter er installasjon ferdig.



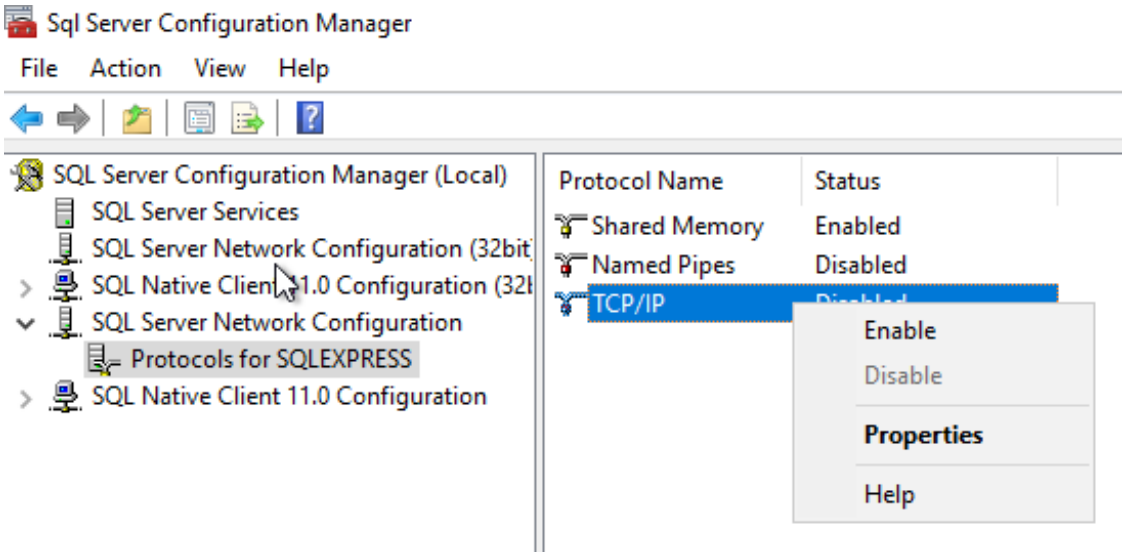
Figur 37: Installasjon SQL Express (1/2)



Figur 38: Installasjon SQL Express (2/2)

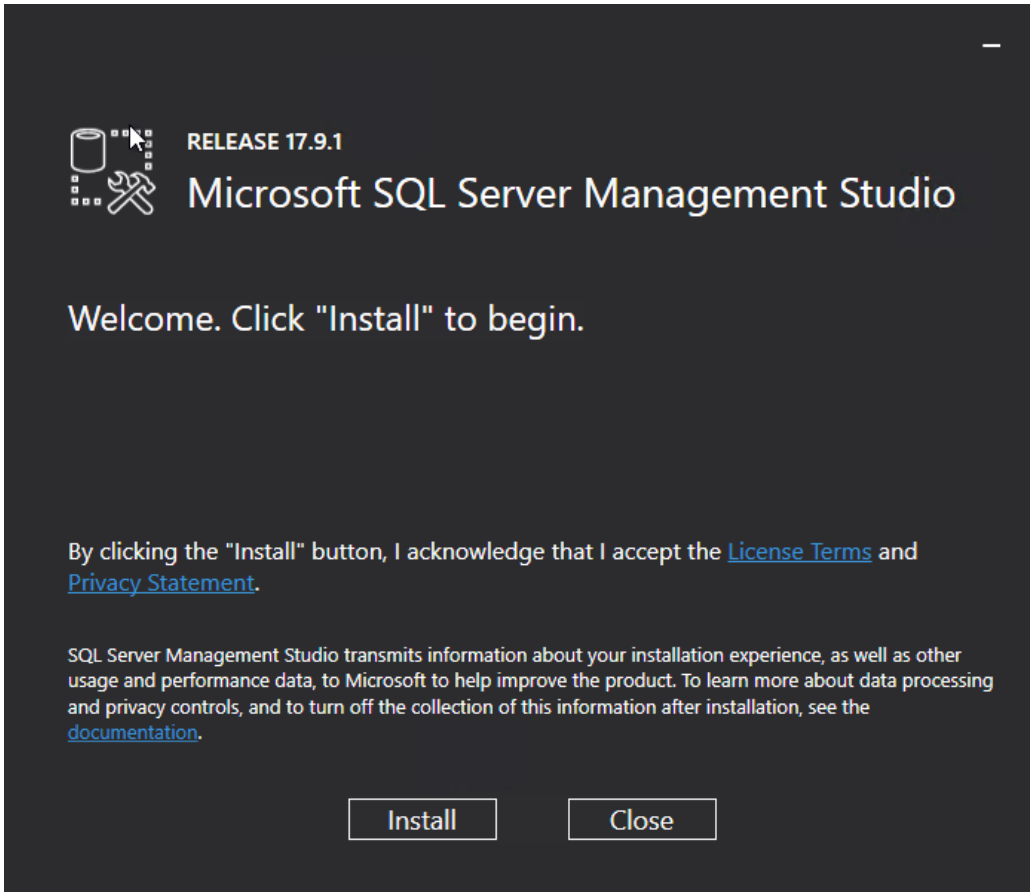
4.2 Konfigurasjon av SQL Express 2017

Vi går videre med å enable “TCP/IP” protokollen og “Named Pipes”. Dette trenger vi senere når vi skal koble til SQL instansen vår.



Figur 39: Konfigurasjon SQL Express

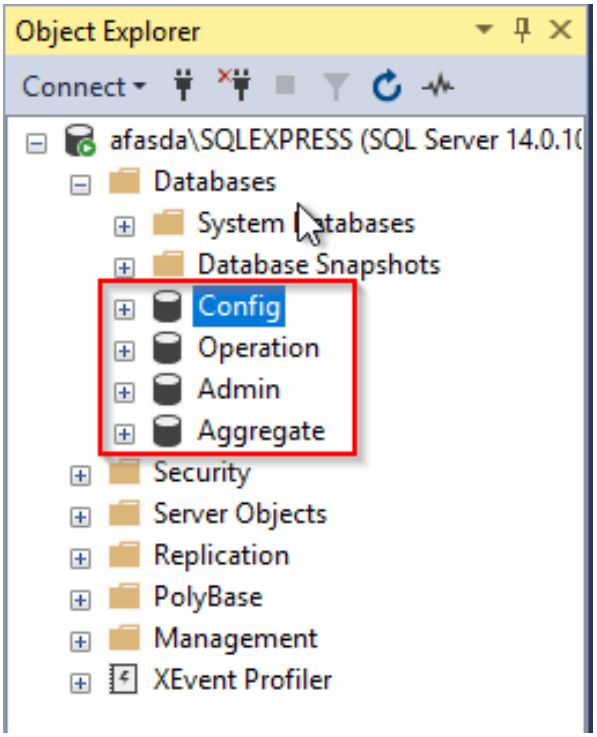
SQL Express 2017 kommer ikke med noe GUI, så vi velger å installere dette ved å laste ned gratis programmet Microsoft SQL Server Management Studio. Dette gjør det mulig for oss å enklere opprette databaser og gjøre endringer.



Figur 40: Installasjon SQL Server Management Studio

4.3 Opprettelse av databaser

Fra NHN sine interne sider og prat med utviklerne, fant vi ut at HelseID er avhengig av fire databaser. Disse heter Config, Operation, Admin og Aggregate. Vi oppretter databasene manuelt i SQL Express 2017 som vist på bildet under.



Figur 41: Databaser

Vi fikk av NHN en USB penn med noen database scripts, som vi bruker for å fylle databasene våre med riktige tabeller og indekser. For databasene Config, Operation og Aggregate generes det med følgende kommando:

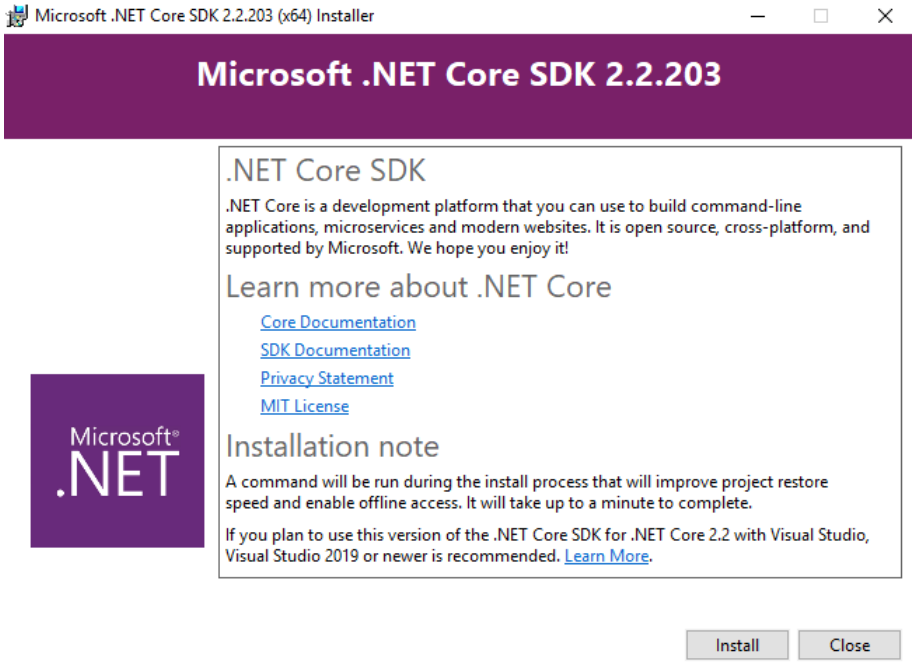
```
.\NHN.FIA.Web.STS.DataMigrator.exe "<Sql-ConnectionString for Config-db>" "<Sql-ConnectionString for Operation-db>" "<Sql-ConnectionString for Aggregate-db>"
```

Outputet fra Powershell vises i bildet under. Vi bruker her bare `Trusted_Connection=True`, fordi vi kjører kommandoen fra samme maskin som SQL Express 2017 er installert.

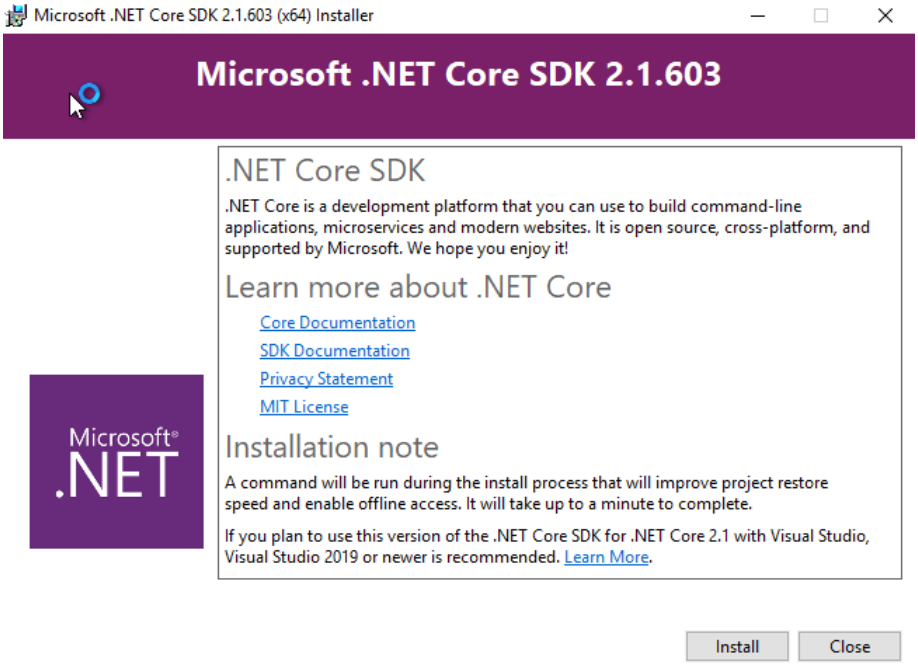
```
PS C:\Users\kube\Downloads\F\NHN.FIA.Web.STS.DataMigrator.0.1.20190313.2> .\NHN.FIA.Web.STS.DataMigrator.exe "Server=
afasda\SQLEXPRESS;Database=Config;Trusted_Connection=True" "Server=afasda\SQLEXPRESS;Database=Operation;Trusted_Conne
ction=True" "Server=afasda\SQLEXPRESS;Database=Aggregate;Trusted_Connection=True"
Master ConnectionString => Data Source=afasda\SQLEXPRESS;Initial Catalog=master;Integrated Security=True;Password=
Master ConnectionString => Data Source=afasda\SQLEXPRESS;Initial Catalog=master;Integrated Security=True;Password=
Master ConnectionString => Data Source=afasda\SQLEXPRESS;Initial Catalog=master;Integrated Security=True;Password=
PS C:\Users\kube\Downloads\F\NHN.FIA.Web.STS.DataMigrator.0.1.20190313.2>
```

Figur 42: Generere tabeller i databasene Operation, Config og Aggregate

For å importere innholdet som trengs i Admin databasen er vi nødt til å installere .NET Core SDK 2.2.203 og 2.1.603 som vist i bildene under.



Figur 43: Installasjon .NET Core SDK 2.2.203



Figur 44: Installasjon .NET Core SDK 2.1.603

Vi kan så genere tabellene for Admin databasen ved å kjøre følgende kommano:

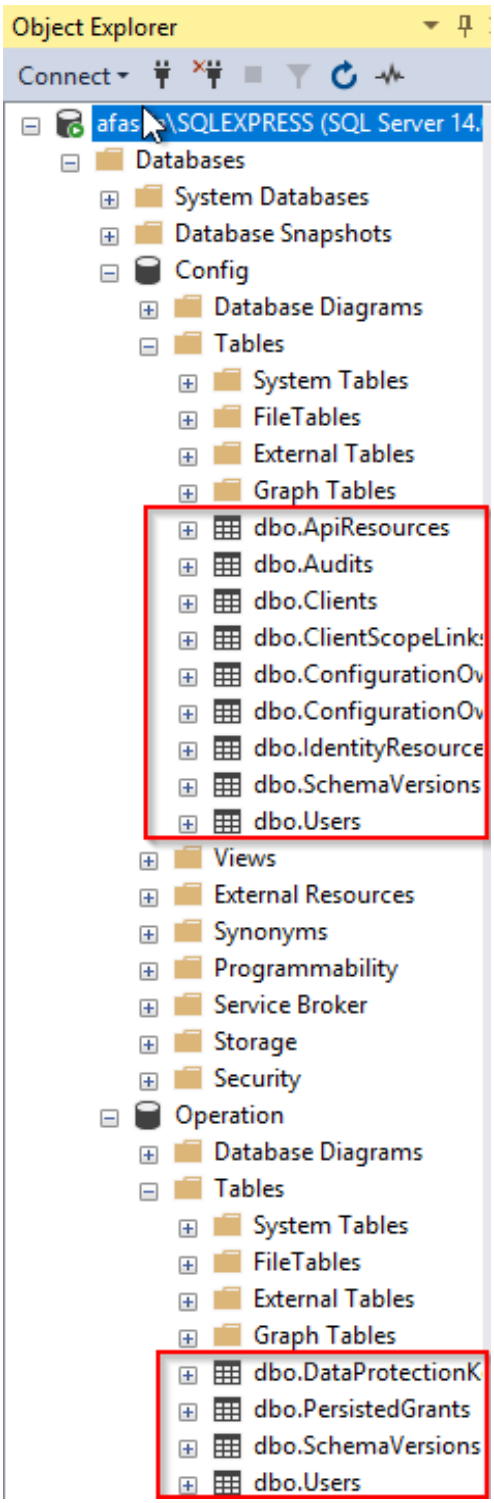
```
dotnet .\NHN.FIA.Web.Admin.DataMigrator.dll "<Sql-ConnectionString for Admin-db>"
```

Output fra kommandoen:

```
PS C:\Users\kube\Downloads\F\NHN.FIA.Web.Admin.DataMigrator.0.1.20190314.1> dotnet .\NHN.FIA.Web.Admin.DataMigrator.dll
"Server=afasda\SQLEXPRESS;Database=Admin;Trusted_Connection=True"
Master ConnectionString => Data Source=afasda\SQLEXPRESS;Initial Catalog=master;Integrated Security=True;Password=
PS C:\Users\kube\Downloads\F\NHN.FIA.Web.Admin.DataMigrator.0.1.20190314.1>
```

Figur 45: Generere tabeller i databasen Admin

Vi har nå generert tabellene for alle de fire databasene, vi kan sjekke dette ved å gå tilbake til SQL Express 2017 vinduet og se om det ligger tabeller der. Vi ser fra bildet under at det gjør det, og vi er nå klare til å gå videre med neste sted, som er å legge til data i disse nyopprettede tabellene.



Figur 46: Tabeller i databasene Config og Operation

4.4 Seed scripts for SQL databasen

Nå som alle databasene og tabellene er på plass, trenger vi å fylle de med data og til det bruker vi seed scripts. I samarbeid med utvikling og Drift 2 ved NHN har vi fått tilgang til disse scriptene.

Vi har gjort endringer på disse scriptene slik at det passer til vårt miljø.

Vi bruker følgende SQL script til dette:

```
USE [Config]
GO

-- ConfigurationOwners
INSERT [dbo].[ConfigurationOwners] (Id, Name, Prefix, IsEnabled, Created, Updated,
Description) VALUES(N'54ef7d1b-74e6-4ccb-959c-4a0ee605a7ea', N'Norsk Helsenett SF',
N'nhn', 1, getdate(), getdate(), '' )
GO

-- IdentityResources
INSERT [dbo].[IdentityResources] ([Id], [Name], [Created], [Updated],
[ConfigurationOwnerId], [Mapping]) VALUES (N'190d3208-9d07-4b03-a2fc-73be5fb51127',
N'helseid://scopes/hpr/hpr_number', getdate(), getdate(), N'54ef7d1b-74e6-4ccb-959c-
4a0ee605a7ea',
N'{"Enabled":true,"Name":"helseid://scopes/hpr/hpr_number","DisplayName":"helseid://scope
s/hpr/hpr_number","Description":"","Required":false,"Emphasize":false,"ShowInDiscoveryDoc
ument":false,"UserClaims":["helseid://claims/hpr/hpr_number"]}')
GO

INSERT [dbo].[IdentityResources] ([Id], [Name], [Created], [Updated],
[ConfigurationOwnerId], [Mapping]) VALUES (N'1a9a84fc-f870-4f3d-8f1a-4bbee83fbee9',
N'helseid://scopes/identity/security_level', getdate(), getdate(), N'54ef7d1b-74e6-4ccb-
959c-4a0ee605a7ea',
N'{"Enabled":true,"Name":"helseid://scopes/identity/security_level","DisplayName":"helsei
d://scopes/identity/security_level","Description":"","Required":false,"Emphasize":false,"
ShowInDiscoveryDocument":false,"UserClaims":["helseid://claims/identity/security_level"]}
')

GO
INSERT [dbo].[IdentityResources] ([Id], [Name], [Created], [Updated],
[ConfigurationOwnerId], [Mapping]) VALUES (N'836d1162-271f-472b-ab41-eade7b3d7d23',
N'helseid://scopes/identity/pid', getdate(), getdate(), N'54ef7d1b-74e6-4ccb-959c-
4a0ee605a7ea',
N'{"Enabled":true,"Name":"helseid://scopes/identity/pid","DisplayName":"helseid://scopes/
identity/pid","Description":null,"Required":false,"Emphasize":false,"ShowInDiscoveryDocum
ent":false,"UserClaims":["helseid://claims/identity/pid"]}')
GO

INSERT [dbo].[IdentityResources] ([Id], [Name], [Created], [Updated],
[ConfigurationOwnerId], [Mapping]) VALUES (N'a43d9bd6-abda-4085-8e04-d3944a871e40',
N'helseid://scopes/client/organization_number', getdate(), getdate(), N'54ef7d1b-74e6-
4ccb-959c-4a0ee605a7ea',
N'{"Enabled":true,"Name":"helseid://scopes/client/organization_number","DisplayName":"hel
seid://scopes/client/organization_number","Description":"","Required":false,"Emphasize":f
alse,"ShowInDiscoveryDocument":false,"UserClaims":["helseid://claims/client/organization_
```

```
number"]}')
GO

INSERT [dbo].[IdentityResources] ([Id], [Name], [Created], [Updated],
[ConfigurationOwnerId], [Mapping]) VALUES (N'c782b237-33c7-4e11-8cf3-32e82458716b',
N'helseid://scopes/identity/assurance_level', getdate(), getdate(), N'54ef7d1b-74e6-4ccb-
959c-4a0ee605a7ea',
N'{"Enabled":true,"Name":"helseid://scopes/identity/assurance_level","DisplayName":"helse
id://scopes/identity/assurance_level","Description":"","Required":false,"Emphasize":false
,"ShowInDiscoveryDocument":false,"UserClaims":["helseid://claims/identity/assurance_level
"]}')
GO

INSERT [dbo].[IdentityResources] ([Id], [Name], [Created], [Updated],
[ConfigurationOwnerId], [Mapping]) VALUES (N'e2c24aab-1227-4749-842c-d9b77385fc86',
N'helseid://scopes/identity/pid_pseudonym', getdate(), getdate(), N'54ef7d1b-74e6-4ccb-
959c-4a0ee605a7ea',
N'{"Enabled":true,"Name":"helseid://scopes/identity/pid_pseudonym","DisplayName":"PID
pseudonym","Description":null,"Required":false,"Emphasize":false,"ShowInDiscoveryDocument
":false,"UserClaims":["helseid://claims/identity/pid_pseudonym"]}')
GO

-- ApiResources
INSERT [dbo].[ApiResources] ([Id], [Name], [ConfigurationOwnerId], [Mapping], [Created],
[Updated]) VALUES (N'6585eaba-dac2-4099-bd78-69ef9949ff67', N'api.helseid.nhn.no',
N'54ef7d1b-74e6-4ccb-959c-4a0ee605a7ea',
N'{"Enabled":true,"Name":"api.helseid.nhn.no","DisplayName":"HelseID admin
API","Description":"HelseID admin
API","ApiSecrets":[],"UserClaims":[],"Scopes":[{"Name":"helseid://scopes/client/sts_confi
guration_admin","DisplayName":"Tilgang til admin
api","Description":null,"Required":false,"Emphasize":false,"ShowInDiscoveryDocument":fals
e,"UserClaims":["helseid://claims/identity/pid_pseudonym","helseid://claims/identity/secu
rity_level"]},{"Name":"helseid://scopes/client/dcr","DisplayName":"helseid://scopes/clien
t/dcr","Description":"","Required":false,"Emphasize":false,"ShowInDiscoveryDocument":fals
e,"UserClaims":[]}]}', getdate(), getdate())
GO

-- Clients
INSERT [dbo].[Clients] ([Id], [ClientId], [Name], [ConfigurationOwnerId], [Mapping],
[Metadata], [ParentId], [IsDcrClient], [Created], [Updated]) VALUES (N'8a90ba3a-157b-
43a0-a4db-e591d62a44dc', N'admin.helseid.nhn.no', N'HelseID Internt Admingrensesnitt',
N'54ef7d1b-74e6-4ccb-959c-4a0ee605a7ea',
N'{"LogoutSessionRequired":true,"AlwaysIncludeUserClaimsInIdToken":true,"IdentityTokenLif
etime":300,"AccessTokenLifetime":3600,"AuthorizationCodeLifetime":300,"AbsoluteRefreshTok
enLifetime":2592000,"SlidingRefreshTokenLifetime":1296000,"RefreshTokenUsage":1,"UpdateAc
cessTokenClaimsOnRefresh":false,"RefreshTokenExpiration":1,"AccessTokenType":0,"EnableLoc
alLogin":false,"IdentityProviderRestrictions":[],"IncludeJwtId":false,"Claims":[],"Always
SendClientClaims":false,"AllowedScopes":["openid","profile","helseid://scopes/client/sts_
configuration_admin"],"AllowOfflineAccess":false,"AllowedCorsOrigins":["http://adminhelse
id3.westeurope.cloudapp.azure.com"],"LogoutUri":null,"Enabled":true,"ClientId":"admin.hel
seid.nhn.no","ProtocolType":"oidc","ClientSecrets":[],"RequireClientSecret":false,"Client
Name":"admin UI. DO NOT
MODIFY!","ClientUri":null,"PrefixClientClaims":true,"LogoUri":null,"AllowRememberConsent"
:true,"AllowedGrantTypes":["implicit"],"RequirePkce":false,"AllowPlainTextPkce":false,"Al
lowAccessTokensViaBrowser":true,"RedirectUris":["http://adminhelseid3.westeurope.cloudapp
.azure.com/signin-
```

```
oidc"],"PostLogoutRedirectUris":["http://adminhelseid3.westeurope.cloudapp.azure.com"],"R
equireConsent":false}', N'{}', NULL, 0, getdate(), getdate())
GO

-- ClientScopeLinks
INSERT INTO [dbo].[ClientScopeLinks] (Scope, ClientClientId, ApiResourceId,
IdentityResourceId) VALUES (N'helseid://scopes/client/sts_configuration_admin',
N'admin.helseid.nhn.no', '6585eaba-dac2-4099-bd78-69ef9949ff67', null)
GO
```

Etter å ha kjørt scriptet kan vi se fra SQL Server Management Studio at det har blitt generert data.

Vi bruker følgende spørring som et eksempel:

```
SELECT TOP (1000) [Id]
      ,[Name]
      ,[Created]
      ,[Updated]
      ,[ConfigurationOwnerId]
      ,[Mapping]
  FROM [Config].[dbo].[IdentityResources]
```

Vi ser at vi får ut innholdet som vist under.

Results | Messages

	Id	Name	Created	Updated
1	190d3208-9d07-4b03-a2fc-73be5fb51127	helseid://scopes/hpr/hpr_number	2019-04-25 15:08:16.1566667	2019-04-25 15:08:16.1566667
2	1a9a84fc-f870-4f3d-8f1a-4bbee83fbee9	helseid://scopes/identity/security_level	2019-04-25 15:08:16.1566667	2019-04-25 15:08:16.1566667
3	836d1162-271f-472b-ab41-eade7b3d7d23	helseid://scopes/identity/pid	2019-04-25 15:08:16.1700000	2019-04-25 15:08:16.1700000
4	a43d9bd6-abda-4085-8e04-d3944a871e40	helseid://scopes/client/organization_number	2019-04-25 15:08:16.1866667	2019-04-25 15:08:16.1866667
5	c782b237-33c7-4e11-8cf3-32e82458716b	helseid://scopes/identity/assurance_level	2019-04-25 15:08:16.2033333	2019-04-25 15:08:16.2033333
6	e2c24aab-1227-4749-842c-d9b77385fc86	helseid://scopes/identity/pid_pseudonym	2019-04-25 15:08:16.2033333	2019-04-25 15:08:16.2033333

Figur 47: Innhold i tabellen IdentityResources

SQL databasen skal nå være fylt med noe data, men vi mangler fortsatt noe for at HelseID applikasjonen skal fungere tilfredsstillende. Vi trenger en testbruker til Admin pålogging.

4.5 Generering av admin bruker

For å få bruke HelseID applikasjonen trenger vi å generere en Admin bruker i databasen vår. Det gjør vi med dette sql scriptet:

```
USE [Admin]
GO

INSERT INTO [dbo].[AdminUsers]
            ([Id]
            ,[UserId]
            ,[Description]
            ,[Created]
```

```
            ,[Updated])
    VALUES(
            NEWID()
               ,'#HIDDEN'
            ,'Testbruker1'
            ,getdate()
            ,getdate());

GO
```

4.6 DataProtectionKeys

Det viste seg at det var noe feil med tabellen DataProtectionKeys i databasen Admin når vi la inn disse tabellene med metoden beskrevet i kapittel 4.3 i dette dokumentet. Når vi testet HelseID fikk vi alltid feilmeldinger når applikasjonen skulle finne nøkler her. Webapplikasjoner må ofte lagre sikkerhetssensitiv data. ASP.NET bruker et kryptografisk API til å beskytte slik data og det brukes DataProtectionKeys for å styre tilgangen til kryptert data. Denne tabellen i databasen er et sted hvor slike genererte nøkler kan lagres og gi sentralisert tilgang til dem.

Vi slettet derfor de tabellene vi hadde opprettet tidligere og fikk et oppdatert script av ressurser i NHN for å legge inn nye tabeller. Scriptet under ble brukt og etter å ha gjort det fikk vi ikke flere slike feilmeldinger.

```
CREATE TABLE [dbo].[DataProtectionKeys](
       [FriendlyName] [varchar](255) NOT NULL,
       [XmlData] [varchar](max) NOT NULL,
CONSTRAINT [PK_DataProtectionKeys] PRIMARY KEY CLUSTERED
(
       [FriendlyName] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

/****** Object:  Table [dbo].[PersistedGrants]     Script Date: 25.04.2019 14:40:08
******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[PersistedGrants](
       [GrantKey] [varchar](200) NOT NULL,
       [Type] [varchar](50) NOT NULL,
       [ClientId] [varchar](200) NOT NULL,
       [CreationTime] [datetime2](7) NOT NULL,
       [Data] [varchar](max) NOT NULL,
       [Expiration] [datetime2](7) NULL,
```

```
	[SubjectId] [varchar](200) NULL,
CONSTRAINT [PK_PersistedGrants] PRIMARY KEY CLUSTERED
(
	[GrantKey] ASC,
	[Type] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

/****** Object:  Table [dbo].[SchemaVersions]      Script Date: 25.04.2019 14:40:08
******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[SchemaVersions](
	[Id] [int] IDENTITY(1,1) NOT NULL,
	[ScriptName] [nvarchar](255) NOT NULL,
	[Applied] [datetime] NOT NULL,
CONSTRAINT [PK_SchemaVersions_Id] PRIMARY KEY CLUSTERED
(
	[Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
GO

/****** Object:  Table [dbo].[Users] Script Date: 25.04.2019 14:40:08 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Users](
	[SubjectId] [varchar](255) NOT NULL,
	[ClaimsAsJson] [varchar](max) NOT NULL,
	[Created] [datetime2](7) NOT NULL,
	[Updated] [datetime2](7) NOT NULL,
CONSTRAINT [PK_Users] PRIMARY KEY CLUSTERED
(
	[SubjectId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF,ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
SET ANSI_PADDING ON
GO

/****** Object:  Index [IX_PersistedGrants_SubjectId]     Script Date: 25.04.2019
14:40:08 ******/
CREATE NONCLUSTERED INDEX [IX_PersistedGrants_SubjectId] ON [dbo].[PersistedGrants]
(
	[SubjectId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF,DROP_EXISTING
= OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)ON [PRIMARY]
GO
```

```
SET ANSI_PADDING ON
GO

/****** Object:  Index [IX_PersistedGrants_SubjectId_ClientId]    Script Date: 25.04.2019
14:40:08 ******/
CREATE NONCLUSTERED INDEX [IX_PersistedGrants_SubjectId_ClientId] ON
[dbo].[PersistedGrants]
(
       [SubjectId] ASC,
       [ClientId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF,DROP_EXISTING
= OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)ON [PRIMARY]
GO
SET ANSI_PADDING ON
GO

/****** Object:  Index [IX_PersistedGrants_SubjectId_ClientId_Type]      Script Date:
25.04.2019 14:40:08 ******/
CREATE NONCLUSTERED INDEX [IX_PersistedGrants_SubjectId_ClientId_Type] ON
[dbo].[PersistedGrants]
(
       [SubjectId] ASC,
       [ClientId] ASC,
       [Type] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF,DROP_EXISTING
= OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)ON [PRIMARY]
GO
ALTER TABLE [dbo].[Users] ADD  CONSTRAINT [DF_Users_Created]  DEFAULT(getdate()) FOR
[Created]
GO
ALTER TABLE [dbo].[Users] ADD  CONSTRAINT [DF_Users_Updated]  DEFAULT(getdate()) FOR
[Updated]
GO
```

5. AKS Engine

For å implementere et Kubernetes cluster bruker vi verktøyet AKS Engine. For å kunne bruke dette verktøyet må noen ting først på plass. Microsoft Azure CLI må installeres og AKS Engine må lastes ned og konfigureres. Vi oppretter også en nøkkel for pålogging til master node med SSH. Alt dette gjøres lokalt på prosjektgruppens personlige maskiner.

5.1 Installasjon Microsoft Azure CLI

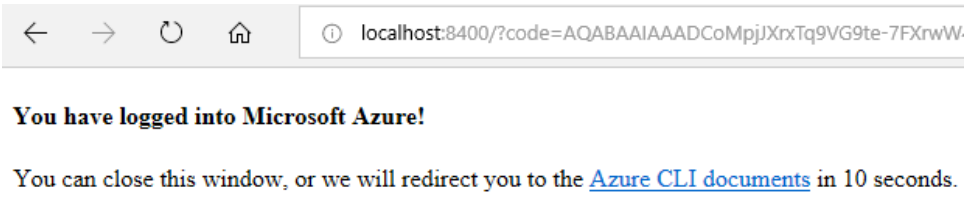
Vi laster ned siste versjon av Azure CLI og starter installasjonen, ikke noe konfigurering å gjøre her. Innen et par minutter er installasjonen ferdig

Figur 48: Installasjon Azure CLI

Vi sjekker om Azure CLI fungerer som forventet med å kjøre følgende kommando i Powershell for å logge oss inn i Azure gjennom CLI:

```
az login
```

En nettside blir da automatisk åpnet der du kan logge inn. Når du har gjort det, blir du sendt videre denne siden:



Figur 49: Azure innlogging (1/2)

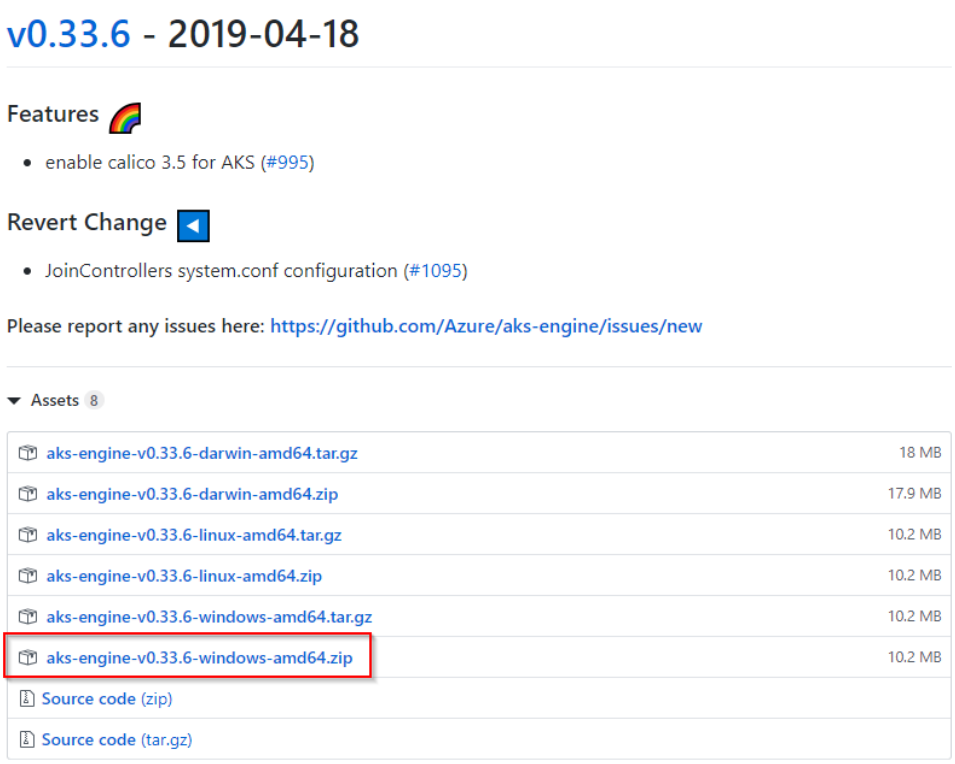
Dette er output man ser i Powershell ved en vellykket pålogging. Du får informasjon om tilgjengelige Azure abonnement til brukeren du logget på med.

```
PS C:\Users\Valla> az login
Note, we have launched a browser for you to login. For old experience with device code,
You have logged in. Now let us find all the subscriptions to which you have access...
[
  {
    "cloudName": "AzureCloud",
    "id": "42126d86-483a-4f47-8594-dd295f0a8a4e",
    "isDefault": false,
    "name": "Azure for Students",
    "state": "Disabled",
    "tenantId": "09a10672-822f-4467-a5ba-5bb375967c05",
    "user": {
      "name": "olejord@ntnu.no",
      "type": "user"
    }
  },
  {
    "cloudName": "AzureCloud",
    "id": "bc15a4e7-34c4-4645-933f-a3b8dc5d01d8",
    "isDefault": true,
    "name": "Pay-As-You-Go(Converted to EA)",
    "state": "Enabled",
    "tenantId": "09a10672-822f-4467-a5ba-5bb375967c05",
    "user": {
      "name": "olejord@ntnu.no",
      "type": "user"
    }
  }
]
PS C:\Users\Valla>
```

Figur 50: Azure innlogging (2/2)

5.2 Installasjon av AKS Engine

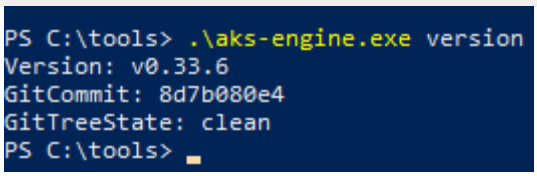
Nå som Azure CLI er på plass må vi ordne AKS Engine verktøyet. Vi laster ned siste versjon av AKS Engine og pakker den ut i mappen C:\tools for videre bruk.



Figur 51:: Installasjon AKS Engine (1/3)

Sjekker at vi har riktig versjon og at verktøyet kan brukes med kommando:

```
aks-engine.exe version
```



Figur 52: Installasjon AKS Engine (1/3)

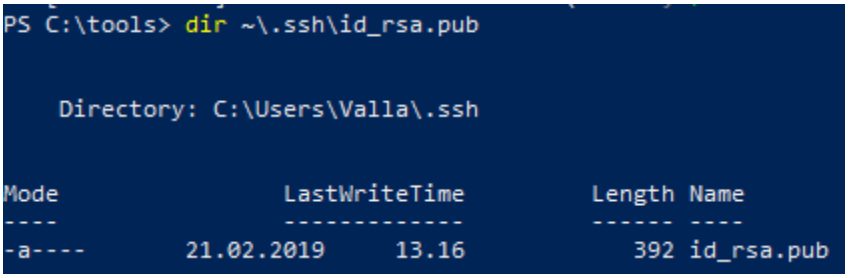
Legger stien til AKS Engine inn i miljøvariabelen $PATH for enkel videre bruk slik at vi slipper å stå i denne mappen hver gang verktøyet skal brukes.

Figur 53::Installasjon AKS Engine (1/3)

5.3 Opprette SSH nøkkel

Det første vi gjør er å sjekke om vi allerede har en SSH nøkkel liggende på maskinen med kommando:

```
dir ~\.ssh\id_rsa.pub
```

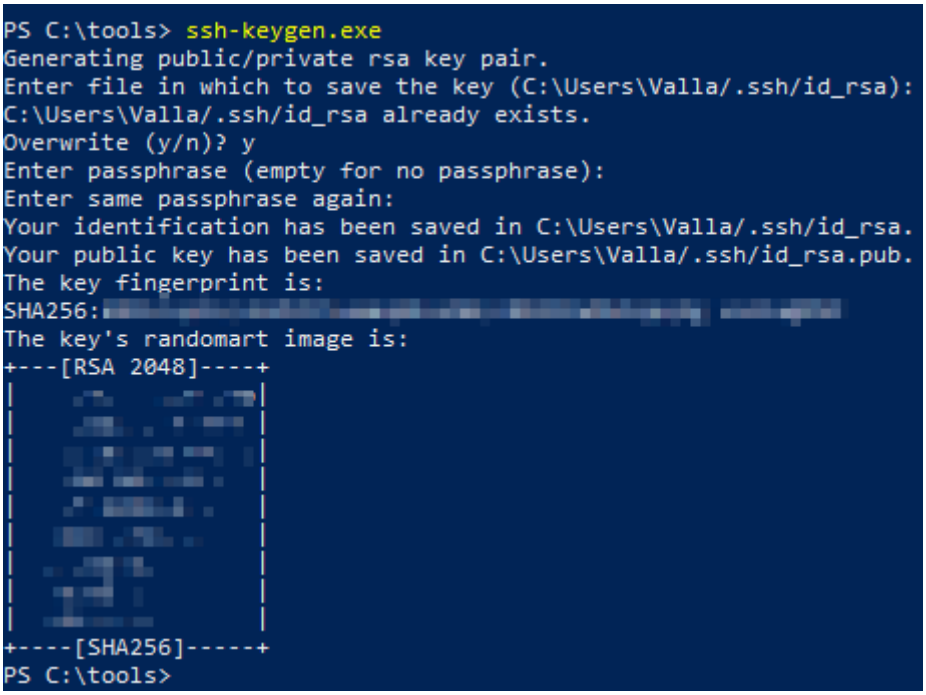


Figur 54::Opprette SSH nøkkel (1/2)

Siden vi allerede har en nøkkel trenger vi ikke nødvendigvis opprette en ny en, men for sikkerhets skyld gjør vi dette da denne nøkkelen er blitt brukt andre steder tidligere.

Windows 10 versjon 1803 og senere kommer med SSH klient allerede installert, hvis du ikke har programmene ssh.exe og ssh-keygen.exe må disse installeres før neste steg blir gjennomført.

Vi bruker `ssh-keygen.exe` for å opprette en ny nøkkel.



Figur 55::Opprette SSH nøkkel (1/2)

5.4 Azure Service Principal

AKS Engine og Kubernetes trenger tilgang til å distribuere ressurser i Azure miljøet vårt for å bygge clusteret samt for konfigurering av ressurser som Azure Load Balancer når clusteret kjører. Vi oppretter derfor en Azure Service Principal for å tildele disse rettighetene. Den sikreste måten å gjøre det på er å lage en egen ressursgruppe, som i vårt tilfelle heter Bachelor121, og gi Service Principal tilgang til

kun denne ressursgruppen slik at den ikke kan gjøre endringer på resten av abonnementet.

Vi finner først ID-en til ressursgruppen Bachelor121 og legger den i variabelen $gruppeID med kommandoen:

```
$gruppeId = (az group show --resource-group bachelor121 --query id --output tsv)
```

Deretter lager vi Service Principal og legger informasjon om den i variabelen $sp med kommando:

```
$sp = az ad sp create-for-rbac --role="Contributor" --scopes=$gruppeId |
ConvertFrom-JSON
```

5.5 Konfigurasjon av JSON mal

Etter alle nødvendige verktøy er på plass, SSH nøkkel og Service Principal er opprettet, kan vi begynne med å konfigurere JSON filen som vi skal bruke med AKS Engine for å opprette Kubernetes clusteret.

Slik ser den tomme malen for et Kubernetes cluster med kun Windows worker noder ut:

```
{
  "apiVersion": "vlabs",
  "properties": {
    "orchestratorProfile": {
      "orchestratorType": "Kubernetes",
      "orchestratorRelease": "1.13"
    },
    "masterProfile": {
      "count": 1,
      "dnsPrefix": "",
      "vmSize": "Standard_D2_v3"
    },
    "agentPoolProfiles": [
      {
        "name": "windowspool2",
        "count": 2,
        "vmSize": "Standard_D2_v3",
        "availabilityProfile": "AvailabilitySet",
        "osType": "Windows",
        "osDiskSizeGB": 128,
        "extensions": [
            {
                "name": "winrm"
            }
        ]
      }
    ],
    "windowsProfile": {
      "adminUsername": "azureuser",
      "adminPassword": "replacepassword1234$",
```

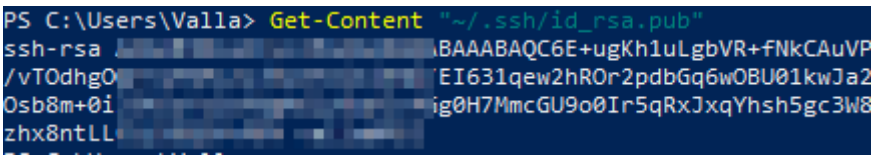
```
      "sshEnabled": true
    },
    "linuxProfile": {
      "adminUsername": "azureuser",
      "ssh": {
        "publicKeys": [
          {
            "keyData": ""
          }
        ]
      }
    },
    "servicePrincipalProfile": {
      "clientId": "",
      "secret": ""
    },
    "extensionProfiles": [
      {
        "name": "winrm",
        "version": "v1"
      }
    ]
  }
}
```

Vi skal nå konfigurere denne malen slik at AKS Engine bygger og implementerer et hybrid windows/linux Kubernetes cluster i vårt Azure miljø.

5.5.1 SSH nøkkel

Først må vi hente noe data som skal fylles inn i JSON malen.
Vi starter med å hente SSH nøkkel data. Det gjør vi med kommandoen:

```
Get-Content "~\.ssh\id_rsa.pub"
```

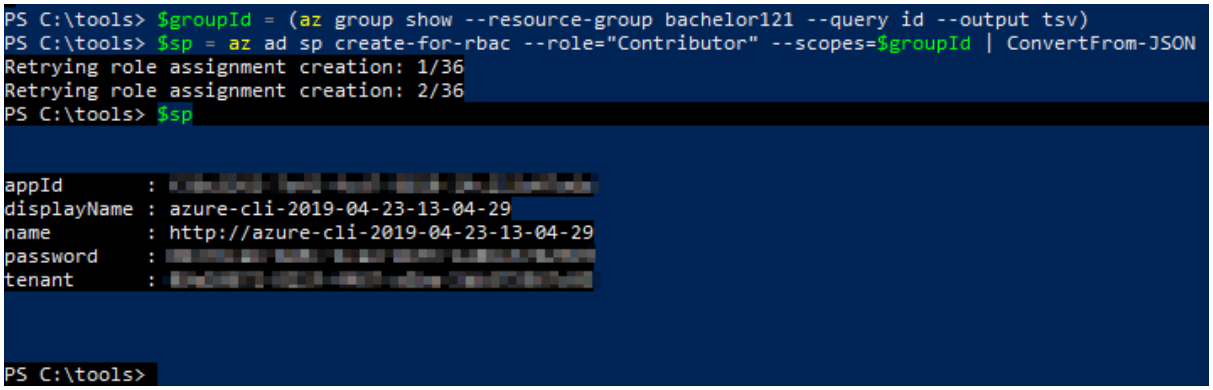


Figur 56: SSH nøkkel data

Dette kopierer vi og fyller inn i feltet “KeyData” i JSON malen.

5.5.2 Service Principal Profile

Vi henter så appID og password fra Service Principal som vi opprettet tidligere ved å skrive ut innholdet i variabelen $sp. Data i appID blir fylt inn i feltet “clientID” og data i password blir fylt inn i feltet “secret” i servicePrincipalProfile i JSON malen.



Figur 57: Service Principal data

5.5.3 Addons - Cluster Autoscaler

Siden vi ønsker å teste muligheten for skalering på node-nivå i clusteret legger vi til en addon kalt Cluster Autoscaler. Med denne addonen aktivert i clusteret vårt blir det automatisk laget nye noder hvis kjørende noder ikke har nok ledige ressurser til å kjøre opp nye pods som står med status “Pending”.
Vi konfigurerer cluster autoscaler slik at hvert scale set har minst 2 noder og maks 5 noder samt at det skal sjekkes hvert 10 sekund etter pods i “Pending” status. For at det skal fungere må worker noder være konfigurert VMSS, virtual machine scale sets, og det er nettopp dette vi gjør i neste steg.

5.5.4 Maskiner

De virtuelle maskinene som skal distribueres og være del av Kubernetes clusteret må konfigureres. Vi konfigurerer en linux Kubernetes master. Denne maskinen trenger ikke å være veldig kraftig så vi bruker vmSize Standard D2s v3 på denne, denne har 2 vCPU og 8GB ram. I feltet “dnsPrefix” her fyller vi inn navnet vi ønsker på Kubernetes clusteret. I dette tilfellet er det ‘helseidv6’.
Maskinene som skal være worker noder vil vi skal være litt kraftigere så vi lager to “agentPoolProfiles”, ett for Windows og ett for Linux og legger to maskiner i hvert pool. Disse maskinene gir vi vmSize Standard D4s v3 som tilsvarer 4 vCPU og 16GB ram. Feltet “availabilityProfile” endrer vi til ‘VirtualMachineScaleSets’ fra standard ‘availabilitySet’ da dette som sagt er nødvendig for at cluster-autoscaler skal fungere.

Til slutt konfigurerer vi brukernavn og passord til Windows maskinene og brukernavn til linux maskinene. Må passe på her at man følger reglene Azure har for krav til passord.

5.5.5 Utfylt mal

Slik ser den ferdig utfylte malen ut som vi bruker for vårt tilpasset hybride Kubernetes cluster:

```
{
        "apiVersion": "vlabs",
        "properties": {
                "orchestratorProfile": {
                        "orchestratorType": "Kubernetes",
                        "orchestratorRelease": "1.13",
                        "kubernetesConfig": {
                                "useManagedIdentity": true,
                                "addons": [{
                                        "name": "cluster-autoscaler",
                                        "enabled": true,
                                        "config": {
                                                "min-nodes": "2",
                                                "max-nodes": "5",
                                                "scan-interval": "10s"
                                        }
                                }
                        ]
                        }
                },
                "masterProfile": {
                        "count": 1,
                        "dnsPrefix": "helseidv6",
                        "vmSize": "Standard_D2s_v3"
                },
                "agentPoolProfiles": [{
                        "name": "windowspool2",
                        "count": 2,
                        "vmSize": "Standard_D4s_v3",
                        "availabilityProfile": "VirtualMachineScaleSets",
                        "osType": "Windows",
                        "osDiskSizeGB": 128

                },
                {
                        "name": "linuxpool2",
                        "count": 2,
                        "vmSize": "Standard_D4s_v3",
                        "availabilityProfile": "VirtualMachineScaleSets"
                }
                ],
                "windowsProfile": {
                        "adminUsername": "kube",
                        "adminPassword": "**********",
                        "sshEnabled": true
                },
                "linuxProfile": {
                        "adminUsername": "kube",
                        "ssh": {
                                "publicKeys": [{
                                        "keyData": "ssh-rsa ***************"
                                }]
                        }
                },
                "servicePrincipalProfile": {
                        "clientId": "********",
```

```
                    "secret": "******"
                }

        }
}
```

5.6 Opprette Kubernetes clusteret

Nå som vi har fylt ut JSON malen er vi veldig nære klar til å bruke AKS Engine for å opprette Kubernetes clusteret i vårt Azure miljø. Det aller siste vi må gjøre før vi kan kjøre opp clusteret er å generere alle artifacts som AKS Engine trenger for å gjennomføre oppgavene sine. Dette gjør vi med en kommandoen som peker på ferdig utfylt JSON fil:

```
aks-engine.exe generate kubernetes-windows-complete.json
```

```
PS C:\tools> aks-engine.exe generate .\kubernetes-windows-complete.json
INFO[0000] Generating assets into _output/helseidv6...
PS C:\tools>
```

Figur 58: Generere AKS Engine artifacts

Vi ser fra output at disse artifacts blir laget i mappen "_output/helseidv6" i stående sti. Helseidv6 er som vi husker navnet vi gav Kubernetes Clusteret når vi konfigurerte JSON filen tidligere. Vi skal bruke to av filene som ble generert i neste steg, som er den faktiske utrullingen av Kubernetes clusteret.

Nå som vi har alt på plass er det eneste som gjenstår for å få Kubernetes clusteret distribuert i Azure miljøet vårt å kjøre en siste kommando. Filene azuredeploy.json og azuredeploy.parameters.json finner vi i mappen som ble laget tidligere og full sti til disse brukes som vi ser i kommandoen:

```
az group deployment create --name helseid-deploy --resource-group bachelor121 --
template-file "C:/tools/_output/helseidv6/azuredeploy.json" --parameters
"C:/tools/_output/helseidv6/azuredeploy.parameters.json"
```

Når denne kjøres begynner det å opprettes ressurser i Azure. Hele prosessen kan ta opp til 15 minutter, lengre hvis du mer komplekse og større clusters du forsøker kjøre opp. Når den er ferdig får du en lang rekke output, men starten skal se lignende ut som i bildet under.

```
PS C:\tools> aks-engine.exe generate kubernetes-windows-complete.json
INFO[0000] Generating assets into _output/helseidv6...
PS C:\tools> az group deployment create --name helseidv6-deploy --resource-group bachelor121
--template-file "C:/tools/_output/helseidv6/azuredeploy.json" --parameters "C:/tools/_output/
helseidv6/azuredeploy.parameters.json"
{
  "id": "/subscriptions/bc15a4e7-34c4-4645-933f-a3b8dc5d01d8/resourceGroups/bachelor121/provi
ders/Microsoft.Resources/deployments/helseidv6-deploy",
  "location": null,
  "name": "helseidv6-deploy",
  "properties": {
```

Figur 59: Opprette Kubernetes cluster

5.7 Kubernetes Control - kubectl

For å sjekke at Kubernetes clusteret er på plass og at det vi konfigurerte ble gjennomført korrekt vil vi kjøre vår første kubectl kommando, men det er to små ting som må på plass før vi kan gjøre dette fra en arbeidsstasjon som ikke er i clusteret.

Først må vi konfigurere verktøyet Kubernetes Control (kubectl). Dette er et eget program som må lastes ned og pakkes ut. Vi gjør dette og legger det i samme mappe som AKS Engine. Verktøyet er hentet fra Kubernetes hjemmesider.

Figur 60: Kubectl client binaries

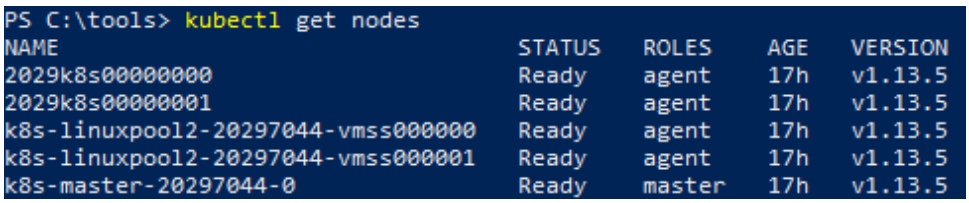
For å få kontakt med Kubernetes clusteret må vi ha tilgang til kubeconfig filen til clusteret. Denne ble laget samtidig som alle de andre filene når vi kjørte `aks engine generate` tidligere. Det blir laget en kubeconfig fil for alle tilgjengelig regioner i Azure

og vi må derfor velge filen for den regionen vi opprettet clusteret i. I vårt tilfelle er dette West Europe og stien til kubeconfig filen blir derfor
`C:/tools/helseidv6/kubeconfig/kubeconfig.westeurope.json`

Nå som vi har funnet riktig kubeconfig fil og stien til den kan vi legge denne stien inn i en miljøvariabel slik:

```
$ENV:KUBECONFIG=(Get-Item
C:\tools\_output\helseidv6\kubeconfig\kubeconfig.westeurope.json).FullName
```

Etter vi har gjort det kan vi endelig sjekke Kubernetes clusteret vårt med en kubectl kommando.



Figur 61: Kubernetes node informasjon

Vi får kontakt med clusteret ser at vi har en master node, to linux worker noder og to windows worker noder.

Hvis man ønsker å administrere clusteret fra en annen arbeidsstasjon trenger man kun tre ting:

1. Last ned og konfigurer kubectl som beskrevet.
2. Ha tilgang til korrekt kubeconfig fil.
3. Legg riktig sti inn i miljøvariabel.

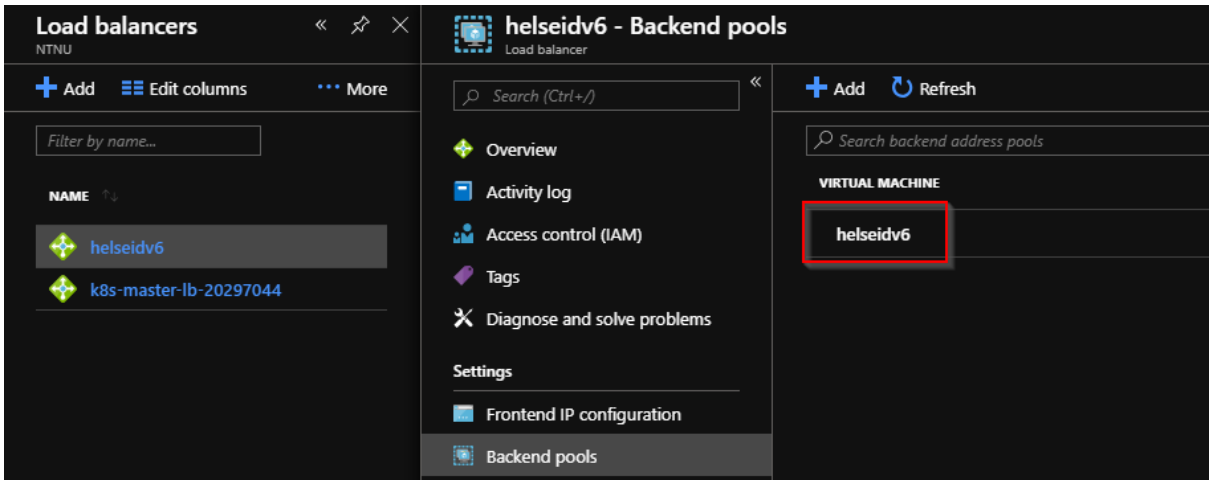
6. Kubernetes

Ikke alt fungerer 100% som vi ønsker rett ut fra boksen når et Kubernetes cluster blir satt opp av AKS Engine. Vi må derfor konfigurere noen containere i Kubernetes og ressurser i Azure før vi fortsetter.

6.1 LoadBalancer

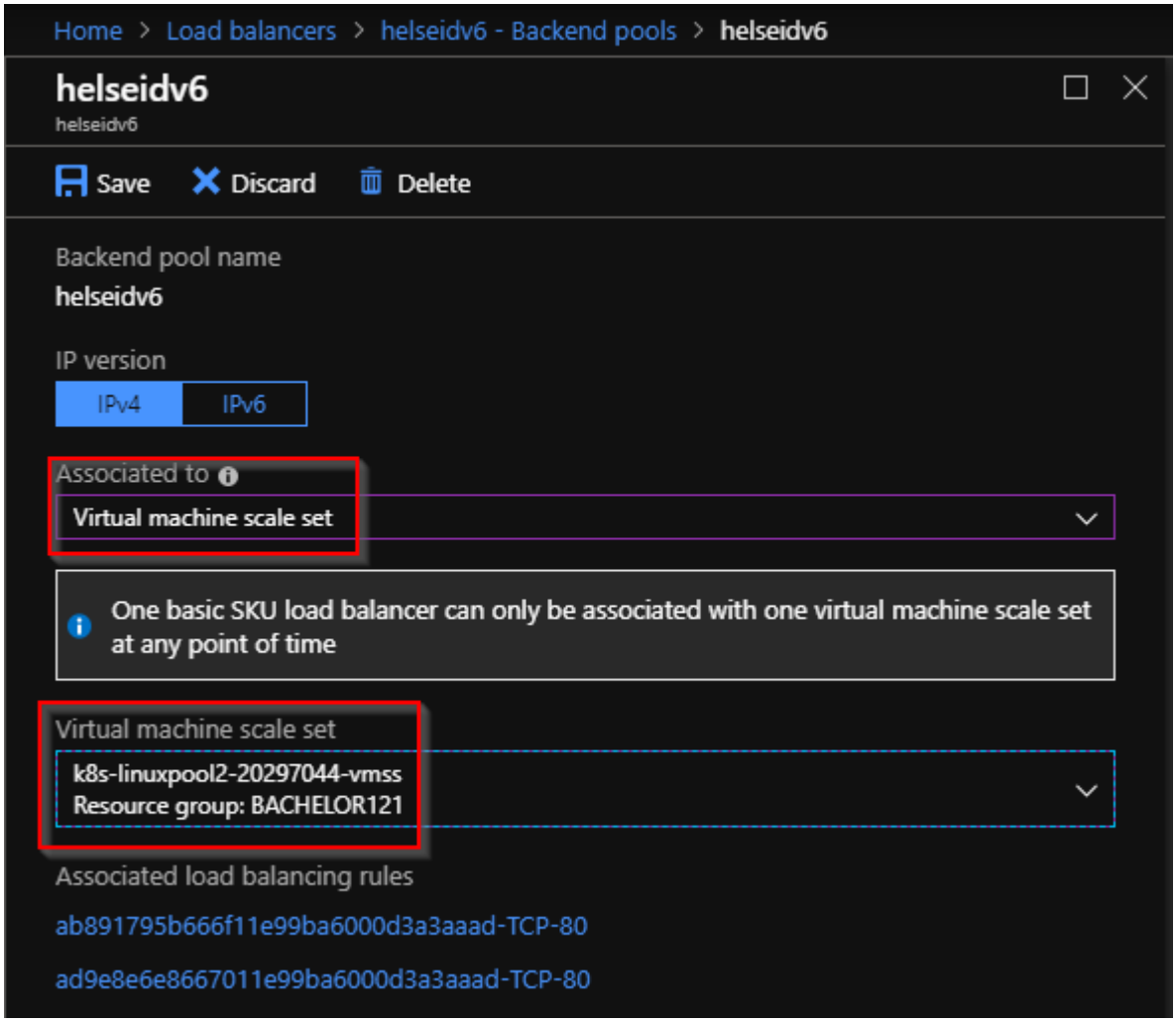
AKS Engine laget en ekstern Azure Load Balancer som brukes av Kubernetes for å eksponere containere til internett. Konfigurasjon av denne er ikke alltid helt riktig, spesielt er ofte Backend Pools tomme, om de i det hele tatt er opprettet, fra starten av. Dette skjer vanligvis hvis du bruker VMSS slik som vi gjør. Hvis Load Balancer ikke har noen maskiner tilknyttet et Backend Pool kommer Load Balancer services/tjenester som vi lager i Kubernetes clusteret vårt ikke til å fungere.

Vi sjekker status på Load Balancer i Azure Portal.



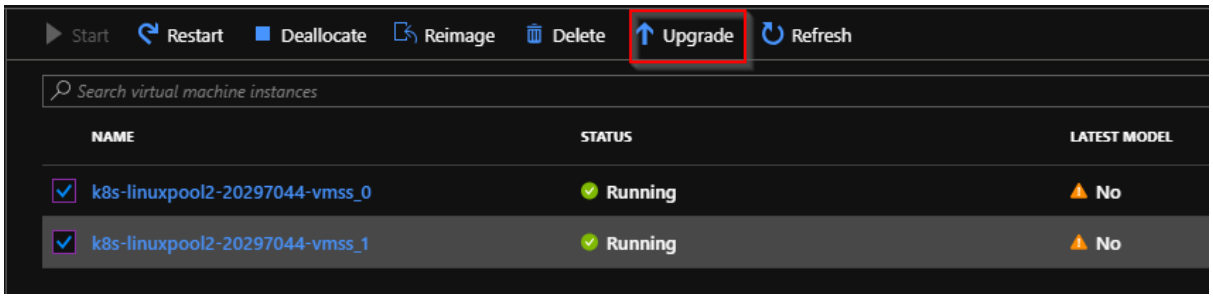
Figur 62: Azure Load Balancer backend pools

Vi ser at det er laget et Backend Pool, men det er tomt. Vi vil derfor legge til et av våre scale sets slik at Load Balancer har noe å jobbe med og forstår koblingen til Kubernetes clusteret vårt.



Figur 63: Konfigurasjon Azure Load Balancer (1/2)

Etter vi har lagt et VMSS til Backend Pool må vi oppgradere de maskinene som tilhører dette.



Figur 64: Konfigurasjon Azure Load Balancer (2/2)

6.2 Cluster autoscaler

Addon-en cluster autoscaler blir konfigurert av AKS Engine når clusteret blir opprettet, men noen ganger har navnet på VMSS-ene som blir opprettet i Azure små forskjeller i fra det som blir konfigurert av AKS Engine. En rask måte å sjekke om

korrekt navn ligger i konfigurasjonen er å se om containeren for Cluster Autoscaler kjører eller ikke. Hvis det er feil konfigurert skal STATUS stå som Error. Vi gjør dette med kommando:

```
kubectl get pods -n kube-system
```

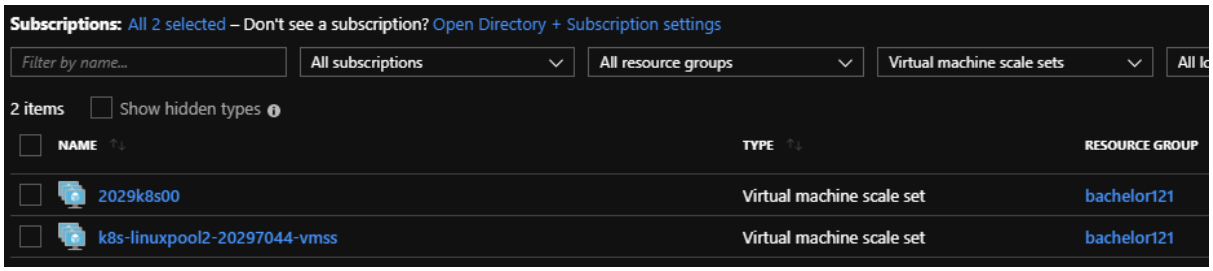
"-n kube-system" sier at vi skal se pods som kjører i "kube-system" namespace, noe Cluster Autoscaler skal gjøre.

```
PS C:\Users\Valla> kubectl get pods -o wide -n kube-system
NAME                                          READY   STATUS
azure-cni-networkmonitor-2r54h                1/1     Running
azure-cni-networkmonitor-4sf4c                1/1     Running
azure-cni-networkmonitor-bss6j                1/1     Running
azure-cni-networkmonitor-jl4q8                1/1     Running
azure-cni-networkmonitor-spf27                1/1     Running
azure-ip-masq-agent-7d6qr                     1/1     Running
azure-ip-masq-agent-h9r7b                     1/1     Running
azure-ip-masq-agent-rkbs9                     1/1     Running
azure-ip-masq-agent-xg6k8                     1/1     Running
azure-ip-masq-agent-z5qqx                     1/1     Running
blobfuse-flexvol-installer-8qxwq              1/1     Running
blobfuse-flexvol-installer-bwlbv              1/1     Running
blobfuse-flexvol-installer-ljrwb              1/1     Running
blobfuse-flexvol-installer-wpls6              1/1     Running
cluster-autoscaler-6ddb4b895f-v98rs           0/1     Error
```

Figur 65: Kube-system pod informasjon

Vi ser her at Cluster Autoscaler ikke kjører. Vi må derfor inn i konfigurasjonen og se om riktig navn til VMSS er lagt inn, som det mest sannsynligvis ikke er. Hvis du i tillegg har flere vmss er det kun ett av dem som blir lagt inn i konfigurasjon av AKS Engine. Siden vi har to vmss og vi vil teste at skalering fungerer må vi også legge til dette i konfigurasjonen.

Vi finner først navnene på de VMSS-ene vi skal legge til. Dette gjør vi i Azure Portal.



Figur 66: VMSS i Kubernetes cluster

Nå som vi har funnet navnene på vmss-ene kan vi legge dem inn i konfigurasjonen for Cluster Autoscaler i Kubernetes. Deploymenten i Kubernetes heter cluster-autoscaler og vi kjører da denne kommandoen for å konfigurere deploymenten:

```
kubectl edit deployment cluster-autoscaler -n kube-system
```

Vi finner deretter frem til riktig sted i deployment YAML filen og legger inn riktig navn som vist i bildet under.

```
template:
  metadata:
    creationTimestamp: null
    labels:
      app: cluster-autoscaler
  spec:
    containers:
    - command:
      - ./cluster-autoscaler
      - --v=3
      - --logtostderr=true
      - --cloud-provider=azure
      - --skip-nodes-with-local-storage=false
      - --nodes=2:5:k8s-linuxpool2-20297044-vmss
      - --nodes=2:5:2029k8s00
      - --scan-interval=10s
      env:
```

Figur 67: Konfigurasjon cluster autoscaler YAML-fil

6.3 Lage secret for docker hub private repository pulls

Siden vi bruker et privat repository for å lagre docker image-ene våre på et sikkert sted må vi lage en secret i Kubernetes for påloggingsinformasjonen til det private repository. Dette gjør vi fordi den secreten kan da brukes i YAML-filene for containere der vi bruker et image fra det private repository.

For å lage en slik secret må vi først ha tilgang til påloggingsinformasjonen. Det er mulig å lage den ved å skrive påloggingsinformasjonen i klartekst rett i kommandolinjen slik:

```
kubectl create secret docker-registry regcred --docker-server=<din-repository-server> --docker-username=<ditt-brukernavn> --docker-password=<ditt-passord> --docker-email=<din-epost>
```

Dette er ikke den sikreste måten å gjøre det på, da du skriver sensitiv informasjon inn i klartekst i kommandolinjen og dette kan lagres ubeskyttet i shell historien i tillegg til at det kan være synlig for andre brukere på PCen i løpet av den tiden kubectl kjører.

Vi vil i stedet lage secreten basert på eksisterende Docker credentials. Når du kjører kommando `docker login` vil det lagre påloggingsinformasjon i en docker config JSON-fil og vi kan bruke denne filen til å opprette en secret. Det gjør vi med denne kommandoen etter å ha kjørt `docker login`:

```
kubectl create secret generic regcred \
    --from-file=.dockerconfigjson=<sti/til/.docker/config.json> \
    --type=kubernetes.io/dockerconfigjson
```

Når en slik secret er blitt opprettet kan vi legge til et argument i YAML-filer der vi bruker et image som er lagret i vårt private repository. Argumentet som må legges til er `imagePullSecrets` og kan se slik ut i en YAML-fil:

```
apiVersion: v1
kind: Pod
metadata:
  name: privat-repo-eksempel
spec:
  containers:
  - name: privat-repo-container
    image: <ditt-private-image>
  imagePullSecrets:
  - name: regcred
```

Uten denne enkle secreten har vi måttet gått inn på hver enkelt node, logget på Docker med credentials til det private repository og hentet hvert enkelt image vi har tenkt å bruke. Dette tar mye mer tid og arbeid enn å lage en secret for det, samt man har måttet gjort dette for hver enkelt nye node som blir opprettet ved f.eks automatisk node/custer skalering. Dette er jo veldig uheldig, da mye av poenget med automatisk node skalering er at alt skjer av seg selv.

7. Bygging av HelseID

Etter å ha fått kildekoden til applikasjonen HelseID og snakket med utviklerne av denne kom vi frem til at det er tre hovedkomponenter som må være på plass for at HelseID skal fungerer som vi ønsker i dette prosjektet. Disse komponentene er:

- HelseID STS - Behandler sikkerhetstokens
- Web Admin - Et web-grensesnitt for administrasjon av HelseID
- Admin API - Et API Web Admin benytter for å utføre sine oppgaver

Alle disse tre komponentene må ha sin egen IP-adresse som vi kan knytte et DNS-navn til og alle tre må kunne kommunisere med hverandre. Det blir derfor naturlig at vi kjører de i hver sin deployment i Kubernetes og lager en service for hver deployment. I tillegg har HelseID noen avhengigheter vi har nevnt tidligere, for eksempel databasene fra kapittel 4, som må konfigureres korrekt. Den kildekoden vi fikk fra utvikling var konfigurert til å fungere på deres testmiljø, så vi måtte inn å endre på ting som connection strings til de databasene vi satte opp tidligere, klientID-er til ID-porten som var satt opp for oss til testbruk og mer.

7.1 HelseID STS

For detaljert informasjon om HelseID STS, se kapittel “3. HelseID” i Designrapporten (Dønnem og Roksvaag, 2019a).

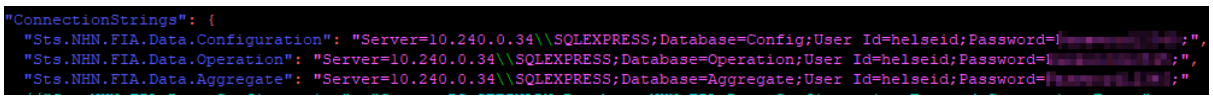
For at integrasjon med ID-porten skal fungere må HelseID STS gå via et spesifikt NHN DNS-navn. NHN har satt opp dette for oss og det DNS-navnet vi fikk er `helseid-sts-bach.test.nhn.no`. Hvis STS ikke går gjennom dette kommer ikke integrasjon med ID-porten til å fungere. I samarbeid med ressurser i NHN blir henvendelser til denne URL-en videresendt til DNS-navnet vi kommer til å gi HelseID STS i Azure som er `helseidwebsts.westeurope.cloudapp.azure.com`. Og henvendelser fra STS går gjennom test.nhn.no slik at ID-porten godtar disse. I tillegg må denne tjenesten kjøre på HTTPS, dette er et krav for ID-porten integrasjon.

7.1.1 Avhengigheter

- Tre databaser - Config, Operation, Aggregate
- .NET Core
- ASP.NET

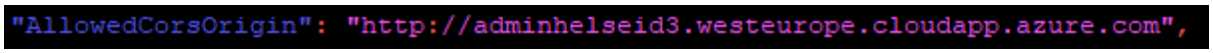
7.1.2 Konfigurere kildekode

HelseID STS bruker flere databaser så det første vi gjør med kildekoden er å endre connection strings til de databasene vi satte opp tidligere. Vi gjør dette i filen `appsettings.json` som i kildekoden har stien `NHN.FIA/source/NHN.FIA.Web.STS/Settings/appsettings.json`



Figur 68: Konfigurasjon kildekode HelseID STS (1/3)

Vi må også endre et felt 'AllowedCorsOrigin' i samme fil slik at HelseID STS stoler på henvendelser som kommer fra Web Admin komponenten vår, da disse er knyttet tett sammen. Vi bruker her DNS navnet som vi kommer til å gi Web Admin når den er konfigurert og rullet ut i Kubernetes.



Figur 69: Konfigurasjon kildekode HelseID STS (2/3)

Til slutt i denne filen vil vi endre autentiseringsdata ID-porten bruker, da vi har fått en ID og secret fra NHN som skal brukes av oss. ID-porten ligger inn som en "Provider" i kildekoden, som sett i bildet under.

Figur 70: Konfigurasjon kildekode HelseID STS (3/3)

Etter vi har konfigurert `appsettings.json` filen må vi konfigurere en fil `Program.cs` for HelseID STS. Denne finner vi på sti `NHN.FIA/source/NHN.FIA.Web.STS/Program.cs` og her legger vi inn navn og passord på selvsignert sertifikat som vi skal bruke for å få ordnet https på STS. Https er nødvendig for at integrasjon med ID-porten skal fungere og er generelt god praksis for webtjenester. Innholdet i filen ser slik ut etter konfigurasjon for vårt miljø:

```
using System.IO;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using System.Net;

namespace NHN.FIA.Web.STS
{

    public class Program
```

```
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
            .UseContentRoot(Directory.GetCurrentDirectory())
            .UseKestrel(options =>
    {
        options.Listen(IPAddress.Any, 443, listenOptions =>
        {
            listenOptions.UseHttps("stsweb.pfx", "**********");
        });
    })
            .ConfigureLogging(factory =>
factory.SetMinimumLevel(LogLevel.Debug).AddDebug())
            .UseStartup<Startup>();
    }
}
```

Tekst markert med rødt er der vi har gjort endringer i kildekoden. Applikasjonen skal lytte på port 443, som er port for https, og vi har lagt inn sertifikat og passordet til sertifikatet.

7.1.3 Bygge Docker image

Det første vi gjør er å se på avhengighetene til HelseID STS og finner et image vi kan bruke i grunn som har disse avhengighetene installert. Vi vet vi trenger .NET Core for å bygge applikasjonen og ASP.NET for å kjøre applikasjonen. For å lage et så lettvektig image som mulig bruker vi et image for å bygge applikasjonen og et annet image for å kjøre det etter det er blitt bygd. Dette fordi det trengs mer ressurser i grunn for å bygge en applikasjon enn å bare kjøre den etter den er bygd. Ved å gjøre det på denne måten blir det endelige image-et vi bruker mindre i størrelse og raskere å rulle ut. Vi kaller image-et som brukes for bygging `build` og det som brukes for å kjøre applikasjonen `runtime`.

Vi har valgt å bruke offisielle Microsoft image-er som base image-er slik at vi slipper ekstra arbeid med å lage dem selv. Disse finner vi i et offentlig Microsoft repository.

7.1.3.1 Lage Dockerfile for HelseID STS

Vi starter med å spesifisere base image-et vi skal bygge applikasjonen vår på

```
FROM mcr.microsoft.com/dotnet/core/sdk:2.2 AS build
```

Fortsetter så med å kopiere over mappen `lib` da denne inneholder en del avhengigheter som vi må ha med oss og setter sti til hvor vi skal arbeide på image-et.

```
WORKDIR /app
COPY lib/. ./lib/
WORKDIR /app/source
```

Videre kopierer vi over prosjektfilen og alle csproj-filer og oppretter dem i eget lag på image-et. Denne delen av Dockerfilen avsluttes med å kjøre kommandoen `dotnet restore`. Denne kommandoen bruker NuGet til å gjenopprette avhengighetene, samt prosjektspesifikke verktøy som er spesifisert i prosjektfilen.

```
COPY source/*.sln .
COPY source/NHN.FIA.Web.STS/*.csproj ./NHN.FIA.Web.STS/
.
.
RUN dotnet restore
```

Etter vi har gjenopprettet alle avhengigheter kopierer vi inn resten av kildekoden og bygger applikasjonen. Vi bygger applikasjonen med å kjøre kommando `dotnet publish -c Release -o out` i mappen `/app/source/NHN.FIA.Web.STS`.

Til slutt vil vi kjøre applikasjonen og vi bruker da vårt runtime image. Her kopierer vi output som vi fikk fra bygge-delen og i tillegg to .pfx filer som er sertifikater for applikasjonen vår. Det siste vi gjør er å konfigurere entrypoint for applikasjonen vår.

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS runtime
WORKDIR /app
COPY --from=build /app/source/NHN.FIA.Web.STS/out ./
COPY stsweb.pfx ./
COPY helseid.pfx ./
ENTRYPOINT ["dotnet", "NHN.FIA.Web.STS.dll"]
```

Etter dette ser den endelige Dockerfile slik ut for HelseID STS:

```
FROM mcr.microsoft.com/dotnet/core/sdk:2.2 AS build
WORKDIR /app
COPY lib/. ./lib/
WORKDIR /app/source

# kopierer solution-fil, csproj-filer og gjenoppretter som forskjellige lag
COPY source/*.sln .
COPY source/NHN.FIA.Web.STS/*.csproj ./NHN.FIA.Web.STS/
COPY source/HelseId.Authentication.Buypass/*.csproj
./HelseId.Authentication.Buypass/
COPY source/NHN.FIA.STS.IntegrationTests/*.csproj
./NHN.FIA.STS.IntegrationTests/
COPY source/NHN.FIA.Task.OperationCleaner/*.csproj
```

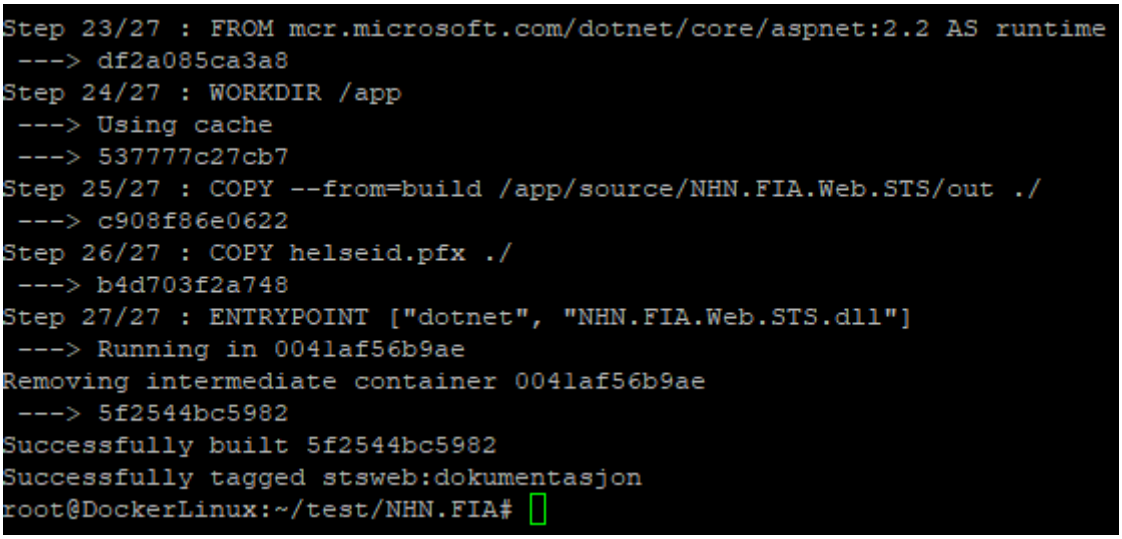
```
./NHN.FIA.Task.OperationCleaner/
COPY source/NHN.FIA.Web.STS.DataMigrator/*.csproj
./NHN.FIA.Web.STS.DataMigrator/
COPY source/NHN.FIA.Web.STS.Tests/*.csproj ./NHN.FIA.Web.STS.Tests/
RUN dotnet restore

# kopierer resten av kildekode og bygger applikasjonen
COPY source/NHN.FIA.Web.STS/. ./NHN.FIA.Web.STS/
COPY source/HelseId.Authentication.Buypass/. ./HelseId.Authentication.Buypass/
COPY source/NHN.FIA.Integrations/. ./NHN.FIA.Integrations
COPY source/NHN.FIA.STS.IntegrationTests/. ./NHN.FIA.STS.IntegrationTests
COPY source/NHN.FIA.Task.OperationCleaner/. ./NHN.FIA.Task.OperationCleaner
COPY source/NHN.FIA.Web.STS.DataMigrator/. ./NHN.FIA.Web.STS.DataMigrator
COPY source/NHN.FIA.Web.STS.Tests/. ./NHN.FIA.Web.STS.Tests
COPY source/Dependencies/. ./Dependencies


WORKDIR /app/source/NHN.FIA.Web.STS
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS runtime
WORKDIR /app
COPY --from=build /app/source/NHN.FIA.Web.STS/out ./
COPY stsweb.pfx ./
COPY helseid.pfx ./
ENTRYPOINT ["dotnet", "NHN.FIA.Web.STS.dll"]
```

7.1.3.2 Bygge image

Nå som Dockerfile er laget kan vi bygge et Docker image. Dette gjør vi ved å kjøre kommando fra mappen der vi har lagt Dockerfile.

```
docker build -t stsweb:dokumentasjon .
```

Dette image-et ble laget for dokumentasjon så vi har gitt det et relevant navn. Vi skal ikke vise bilder for hele prosessen, det hadde blitt for mye, men slik ser siste del av output ut ved et vellykket forsøk:



Figur 71: Bygge image HelseID STS

7.1.3.3 Laste image opp til privat repository

Etter bygging av Docker image-et er gjennomført kan vi laste det opp til vårt private Docker Hub repository slik at vi kan bruke det til å rulle ut applikasjonen i en deployment i Kubernetes.
Vi starter med å endre navn på image-et som en sti til vårt repository og tagger det med navnet vi ønsker det skal få i vårt repository. Bruker kommando

```
docker tag stsweb:dokumentasjon nhnkubebachelor/helseid:stswebdokumentasjon
```

Og til slutt kan vi laste det opp til vårt repository med kommando

```
docker tag stsweb:dokumentasjon nhnkubebachelor/helseid:stswebdokumentasjon
```

Når dette er gjort har vi et image for HelseID STS klart til å brukes i Kubernetes.

7.2 Admin API

Denne tjenesten tilbyr et sett med API-er som Web Admin benytter for å utføre oppgavene sine.

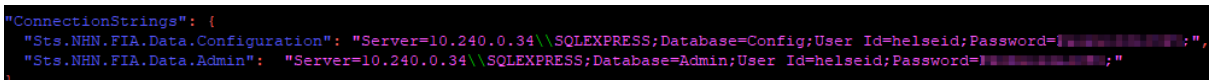
7.2.1 Avhengigheter

- To databaser - Config, Admin
- .NET Core
- ASP.NET

7.2.2 Konfigurerer kildekode

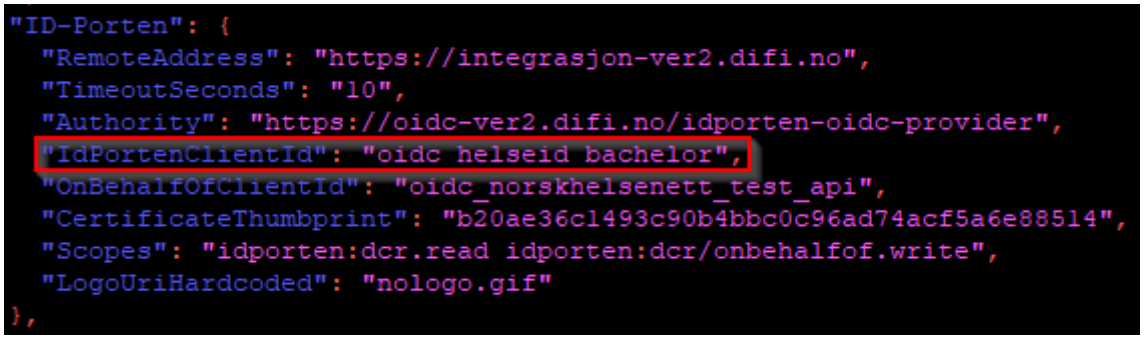
Konfigurering av kildekode for Admin API ligner noe på det vi gjorde med HelseID STS. Vi har igjen database connection strings og ID-Porten autentiseringsdata som

må konfigureres. Dette gjøres for Admin API i fil med samme navn som HelseID STS; `appsettings.json`. Den har her stien `NHN.FIA.Web.Admin/NHN.FIA.Web.Admin/Settings/appsettings.json`.



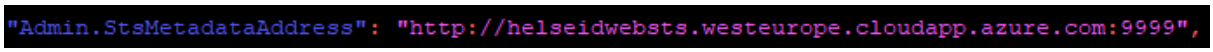
Figur 72: Konfigurasjon kildekode Admin API (1/3)

Så gjør vi det samme for ID-porten som vi gjorde i HelseID STS, men her har vi ikke med secret.



Figur 73: Konfigurasjon kildekode Admin API (2/3)

Det som er nytt her i forhold til HelseID STS er et felt `Admin.StsMetaAddress`. Vi må her endre url til adressen til HelseID STS. Vi endrer det til DNS navnet vi kommer til å gi HelseID når det er rullet ut i Kubernetes.



Figur 74: Konfigurasjon kildekode Admin API (3/3)

7.2.3 Bygge Docker image

Som med konfigurasjonen av kildekoden er det mye som er likt HelseID STS når vi skal bygge et Docker image for Admin API. Den har like avhengigheter som HelseID STS og vi kan derfor bruke samme base image for både `build` og `runtime`.

7.2.3.1 Lage Dockerfile for Admin API

Vi kopierer også her `lib` mappen av samme grunn. Vi kopierer igjen prosjektfiler og gjenoppretter dem med `dotnet restore` i et eget lag på image-et. Dette blir etterfulgt av kopiering av resten av kildekoden og bygging av applikasjonen med samme kommando igjen: `dotnet publish -c Release -o out`.

Eneste forskjellen i runtime delen er at mappen og filen har fått et annet navn.

Slik ser den endelige Dockerfile ut for Admin API.

```
FROM mcr.microsoft.com/dotnet/core/sdk:2.2 AS build
WORKDIR /app
COPY lib/. ./lib/

# kopierer solution-fil, csproj-filer og gjenoppretter som forskjellige lag
COPY *.sln ./
COPY HashIt/*.csproj ./HashIt/
COPY HelseId.Api.XdsSaml/*.csproj ./HelseId.Api.XdsSaml/
COPY HelseId.Api.XdsSaml.Tests/*.csproj ./HelseId.Api.XdsSaml.Tests/
COPY NHN.FIA.Web/*.csproj ./NHN.FIA.Web/
COPY NHN.FIA.Web.Admin/*.csproj ./NHN.FIA.Web.Admin/
COPY NHN.FIA.Web.Admin.DataMigrator/*.csproj ./NHN.FIA.Web.Admin.DataMigrator/
COPY nhn.fia.web.admin.devsts/*.csproj ./nhn.fia.web.admin.devsts/
COPY NHN.FIA.Web.Admin.Tests/*.csproj ./NHN.FIA.Web.Admin.Tests/
COPY NHN.FIA.Web.Dcr/*.csproj ./NHN.FIA.Web.Dcr/
COPY NHN.FIA.Admin.DataSeeder/*.csproj ./NHN.FIA.Admin.DataSeeder/
COPY NHN.FIA.Web.Integrations/*.csproj ./NHN.FIA.Web.Integrations/
COPY NHN.FIA.Web.Integrations.Tests/*.csproj ./NHN.FIA.Web.Integrations.Tests/
RUN dotnet restore NHN.FIA.Web.Admin.sln
RUN dotnet restore NHN.FIA.Web.Admin.Util.sln

# kopierer resten av kildekode og bygger applikasjonen
COPY HashIt/. ./HashIt/
COPY HelseId.Api.XdsSaml/. ./HelseId.Api.XdsSaml/
COPY HelseId.Api.XdsSaml.Tests/. ./HelseId.Api.XdsSaml.Tests/
COPY NHN.FIA.Admin.DataSeeder/. ./NHN.FIA.Admin.DataSeeder/
COPY NHN.FIA.Web/. ./NHN.FIA.Web/
COPY NHN.FIA.Web.Admin/. ./NHN.FIA.Web.Admin/
COPY nhn.fia.web.admin.client/. ./nhn.fia.web.admin.client/
COPY NHN.FIA.Web.Admin.DataMigrator/. ./NHN.FIA.Web.Admin.DataMigrator/
COPY nhn.fia.web.admin.devsts/. ./nhn.fia.web.admin.devsts/
COPY NHN.FIA.Web.Admin.Tests/. ./NHN.FIA.Web.Admin.Tests/
COPY NHN.FIA.Web.Dcr/. ./NHN.FIA.Web.Dcr/
COPY NHN.FIA.Web.Integrations/. ./NHN.FIA.Web.Integrations/
COPY NHN.FIA.Web.Integrations.Tests/. ./NHN.FIA.Web.Integrations.Tests/

WORKDIR /app/NHN.FIA.Web.Admin
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS runtime
WORKDIR /app
COPY --from=build /app/NHN.FIA.Web.Admin/out ./
ENTRYPOINT ["dotnet", "NHN.FIA.Web.Admin.dll"]
```

7.2.3.2 Bygge image

Vi bygger dette image-et på samme måte, eneste forskjellen er navnet vi gir det. Kommandoen vi bruker for dette dokumentasjons eksempelet blir:

```
docker build -t adminapi:dokumentasjon .
```

Og slutten på output blir som vi ser nærmest identisk:

```
Step 34/37 : FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS runtime
 ---> df2a085ca3a8
Step 35/37 : WORKDIR /app
 ---> Using cache
 ---> 537777c27cb7
Step 36/37 : COPY --from=build /app/NHN.FIA.Web.Admin/out ./
 ---> 7c1fbf1d0683
Step 37/37 : ENTRYPOINT ["dotnet", "NHN.FIA.Web.Admin.dll"]
 ---> Running in 3432cb404b5a
Removing intermediate container 3432cb404b5a
 ---> a14c7c39a9be
Successfully built a14c7c39a9be
Successfully tagged adminapi:dokumentasjon
```

Figur 75: Bygge image Admin API

Vi ser det er 10 flere steg i forhold til HelseID STS, dette er fordi hver linje der vi kopierer noe er et eget steg. Siden det er flere mapper å kopiere i kildekoden får vi flere steg å gjennomføre. Hver kopiering tar derimot veldig lite tid, så tiden det tar å bygge de to forskjellige image-ene er ikke veldig forskjellige.

7.2.3.3 Laste image opp til privat repository

Samme fremgangsmåte som med HelseID STS.
Starter med å endre navn på image-et og tagger det med navnet vi ønsker skal vises i vårt repository. Kommandoen her blir:

```
docker tag adminapi:dokumentasjon nhnkubebachelor/helseid:adminapidokumentasjon
```

Laster opp med kommando:

```
docker push nhnkubebachelor/helseid:adminapidokumentasjon
```

Nå er også et image for Admin API klart til å brukes i Kubernetes.

7.3 Web Admin

Denne tjenesten er et web-grensesnitt for administrasjon av HelseID. Den ble laget med Angular CLI.

7.3.1 Avhengigheter

- Angular/Node.js

7.3.2 Konfigurerer kildekode

Web Admin har direkte tilknytning til både HelseID STS og Admin API, og kildekoden reflekterer dette. Det er spesielt en fil vi må konfigurere til å peke på våre tjenester og dette er `clientsettings.json` som finnes i stien `NHN.FIA.Web.Admin/nhn.fia.web.admin.client.ng2/src/assets/clientsettings.json` i kildekoden.
Vi må her endre feltet `"adminApiUrl"` til å peke mot vårt Admin API. Det er også flere felt der Web Admin peker mot seg selv og et felt der vi peker til HelseID STS. Vi må også her endre ID-porten autentiseringsdata som brukes. Etter konfigurasjon ser delen av filen vi konfigurerer slik ut:

```
"environment": "LOCAL_DEV",
"adminApiUrl" : "http://helseidapi.westeurope.cloudapp.azure.com:5000/api",
"brRegApiUrl": "https://data.brreg.no/enhetsregisteret/enhet",
"sts": {
 "idporten_client_id": "oidc_helseid_bachelor",
 "authority": "https://helseid-sts-bach.test.nhn.no",
 "client_id": "admin.helseid.nhn.no",
 "redirect_uri": "http://adminhelseid3.westeurope.cloudapp.azure.com/signin-oidc",
 "post_logout_redirect_uri": "http://adminhelseid3.westeurope.cloudapp.azure.com",
 "response_type": "id_token token",
 "scope": "openid profile helseid://scopes/client/sts_configuration_admin",
 "silent_redirect_uri": "http://adminhelseid3.westeurope.cloudapp.azure.com/signin-
silent-oidc",
```

7.3.3 Bygge Docker image

7.3.3.1 Lage Dockerfile for Web Admin

Ettersom dette er en applikasjon som ble laget med Angular CLI og har andre avhengigheter enn HelseID STS og Admin API kan vi ikke bruke de samme base image-ene. Vi forsøkte å finne et offentlig image med Angular installert og klart til bruk men de vi fant fungerte ikke som vi ønsket når vi testet dem. Vi endte derfor opp med å bygge et eget image fra grunn av. Ulempen med dette var litt ekstra arbeid og vi endte opp med et større image enn det vi ville anbefalt brukt i et produksjonsmiljø, da vi ikke fikk skilt bygge-delen og kjøre-delen fra hverandre.

Vi startet med et enkelt Ubuntu image i grunn. I Dockerfile kjørte vi kommandoer for å installere og oppdatere nødvendige moduler for å bygge og kjøre en Angular applikasjon.

Til slutt brukes kommandoen `ng serve --host 0.0.0.0 --port 80 --disable-host-check` for å kjøre opp applikasjonen på port 80.

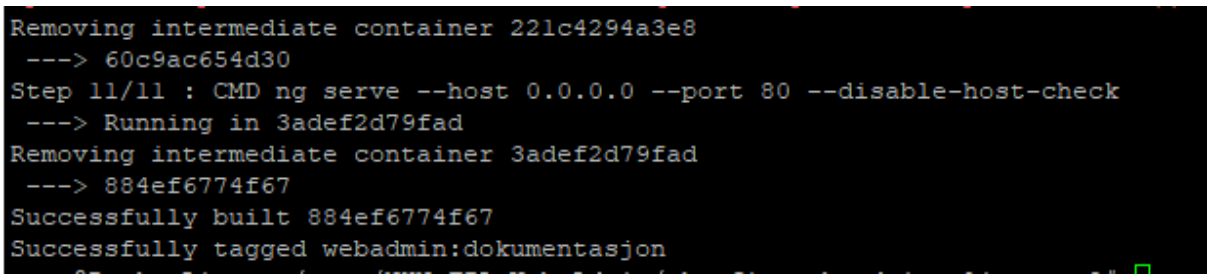
Den endelige Dockerfile ser slik ut:

```
FROM ubuntu:18.04
WORKDIR /app

COPY . .

RUN apt update
RUN apt install npm -y
RUN npm update
RUN apt install nodejs -y
RUN npm install -g @angular/cli -y
RUN npm install --save-dev @angular-devkit/build-angular -y
RUN ng update @angular/cli -y

CMD ng serve --host 0.0.0.0 --port 80 --disable-host-check
```

7.3.3.2 Bygge image

Nå er vi tilbake på kjent grunn. Samme prosessen som vi har brukt tidligere. Kommandoen vi bruker er:

```
docker build -t webadmin:dokumentasjon .
```



Figur 75: Bygge image Web Admin

7.3.3.3 Laste image opp til privat repository

Starter med å endre navn på image-et og tagger det med navnet vi ønsker skal vises i vårt repository. Kommandoen her blir:

```
docker tag webadmin:dokumentasjon nhnkubebachelor/helseid:webadmindokumentasjon
```

Laster opp med kommando:

```
docker push nhnkubebachelor/helseid:webadmin:dokumentasjon
```

Nå er også et image for Web Admin klart til å brukes i Kubernetes.

8. HelseID i Kubernetes

Etter vi har konfigurert kildekode, bygget image for de nødvendige tjenestene og lastet disse opp i et privat repository kan vi nå begynne prosessen med å kjøre applikasjonene ut som deployments i Kubernetes og lage services for disse.

8.1 HelseID STS

Fra Designrapporten (Dønnem og Roksvaag, 2019a) vet vi at containere kan rulles ut i deployments i Kubernetes og man bruker YAML-filer til konfigurasjon av slike deployments. Så det vi må gjøre først er å opprette og konfigurere en slik YAML-fil. Vi vil i den samme YAML-filen konfigurere en service som eksponerer HelseID STS på internett via ekstern Load Balancer i Azure.

8.1.1 Konfigurasjon av YAML-fil

8.1.1.1 Konfigurasjon av Deployment

Når vi skal konfigurere YAML-fil for HelseID STS er det fem felt der innholdet er viktig. Disse er:

- kind - hvilken type ressurs lages i Kubernetes fra denne YAML-filen
- image - hvilket image skal containerne bygges fra
- imagePullSecrets - for å få lov til å hente et image fra privat repository
- nodeSelector - siden vi har et hybrid cluster må vi si hvilke typer noder denne deploymenten skal rulle ut containere på. Linux image kan ikke kjøre på Windows noder og vica versa.
- ports - hvilken port skal containere i denne deploymenten lytte på

Det første en YAML-fil starter med er hvilken API og versjon av API som skal brukes. `apps` er en gruppe Kubernetes API-er som brukes når det skal rulles ut Deployments, ReplicaSets, RollingUpdates med mer. Dette er den mest brukte API-gruppen i Kubernetes. v1 er en stabil versjon av denne gruppen API-er. Vi bruker derfor dette i våre YAML-filer når vi lager en deployment for våre applikasjoner.

Det vi skal lage er en deployment og vi fyller derfor inn `kind:` med `Deployment`.

Det image-et vi bruker for HelseID STS er siste stabile versjon vi har laget for denne applikasjonen, som er `stsweb-16`, og siden det ligger i vårt private repository konfigurerer vi YAML-filen til å bruke det med: `nhnkubebachelor/helseid:stsweb-16`. Vi navngir også Deploymenten og applikasjonen “stsweb-16”.
Vi bruker den secret-en vi lagde tidligere, regcred, som imagePullSecret for å få lov til å hente image-et.

Dette er et Linux image og vi må derfor velge Linux som nodeSelector for at containerne skal kunne kjøre i vårt Kubenetes cluster.

Siden denne applikasjonen bruker https og dermed lytter på port 443, konfigurerer vi det.

Til slutt konfigurerer vi deploymenten slik at hver enkelt container reserverer 10% av en vCPU og 300Mb minne på maskinen som hoster den. Dette er ikke nødvendig for å få containerne rullet ut, de kommer til å fungere uten dette, men for at vi skal kunne benytte oss av autoskalering må det være konfigurert. Reservasjon av ressurser kan gjøres etter deploymenten er rullet ut men like greit å gjøre det fra starten av.

Konfigurasjon av Deployment delen av HelseID STS ser da slik ut:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: stsweb-16
  labels:
    app: stsweb-16
spec:
  replicas: 1
  template:
    metadata:
      name: stsweb-16
      labels:
        app: stsweb-16
    spec:
      containers:
      - name: stsweb-16
        image: nhnkubebachelor/helseid:stsweb-16
        resources:
          requests:
            cpu: .1
            memory: 300m
        ports:
          - containerPort: 443
      imagePullSecrets:
      - name: regcred
      nodeSelector:
        "beta.kubernetes.io/os": linux
  selector:
    matchLabels:
      app: stsweb-16
```

8.1.1.2 Konfigurasjon av Service

Siden vi ønsker å eksponere HelseID STS til internett konfigurerer vi en Service for å gjøre dette via den eksterne Load Balancer som ble opprettet av AKS Engine i Azure når det satte opp Kubernetes clusteret.

Typen service blir derfor LoadBalancer og vi eksponerer container port 443 på host port 443 siden vi har konfigurert applikasjonen og deploymenten til å lytte på denne porten da denne brukes til https trafikk.

Ellers gir vi den samme navn som deploymenten den skal eksponere og peker den til riktig deployment med å konfigurere samme navn i feltet `app:` som vi gjorde for deploymenten.

Konfigurasjon av Service delen av HelseID STS YAML-fil blir derfor slik:

```
apiVersion: v1
kind: Service
metadata:
 name: stsweb-16
spec:
 type: LoadBalancer
 ports:
 - protocol: TCP
   port: 443
   targetPort: 443
 selector:
   app: stsweb-16
```

8.1.1.3 Konfigurert YAML-fil

Til slutt kan vi da slå sammen de to delene i en ferdig YAML-fil med konfigurasjon for HelseID STS. Vi kan nå bruke filen til å rulle ut HelseID STS i Kubernetes.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: stsweb-16
  labels:
    app: stsweb-16
spec:
  replicas: 1
  template:
    metadata:
      name: stsweb-16
      labels:
        app: stsweb-16
    spec:
      containers:
      - name: stsweb-16
        image: nhnkubebachelor/helseid:stsweb-16
        resources:
          requests:
```

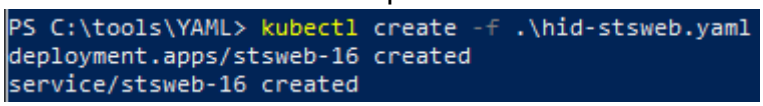
```
            cpu: .1
            memory: 300m
        ports:
          - containerPort: 443
      imagePullSecrets:
      - name: regcred
      nodeSelector:
        "beta.kubernetes.io/os": linux
  selector:
    matchLabels:
      app: stsweb-16
---
apiVersion: v1
kind: Service
metadata:
 name: stsweb-16
spec:
 type: LoadBalancer
 ports:
 - protocol: TCP
   port: 443
   targetPort: 443
 selector:
   app: stsweb-16
```

8.1.2 Rulle ut i Kubernetes

Nå som vi har en YAML-fil for deployment er vi klar til å rulle ut applikasjonen i Kubernetes. Dette gjøres med en enkel kommando slik:

```
kubectl create -f hid-stsweb.yaml
```

Når vi kjører denne kommandoen får vi output:



Figur 76: Rulle ut i Kubernetes HelseID STS (1/2)

Vi ser at det er laget en deployment og en service for applikasjonen vår.

Vi sjekker om det blir laget noen pods med kommando `kubectl get pods` like etter deployment er rullet ut og vi får da denne outputen:

Figur 77: Rulle ut i Kubernetes HelseID STS (2/2)

Vi ser det blir laget èn pod som konfigurert i YAML-filen.

8.2 Admin API

Prosessen med å rulle ut denne applikasjonen blir veldig lik slik det ble gjort med HelseID STS. Forklarer derfor ikke like detaljert hver del av YAML-filen, henviser til kapittel 8.1 for dette. Vi kommer igjen til å ha en deployment og service konfigurert i samme YAML-fil, da Admin API også skal eksponeres til internett.

8.2.1 Konfigurasjon av YAML-fil

8.2.1.1 Konfigurasjon av Deployment

Vi gir deploymenten navnet adminapi-14, da dette er siste stabile versjon av image-et vi har laget for Admin API konfigurerer vi dette i YAML-filen som `nhnkubebachelor/helseid:adminapi-14`. Bruker secret regcred for å kunne hente image-et fra vårt private repository.

Applikasjonen er konfigurert til å lytte på port 5000, så vi bruker denne porten for containerne som deploymenten skal rulle ut.

Vi vil som med HelseID STS reservere 10% av en vCPU og 300Mb minne på hosten containerne kommer til å kjøre på, slik at autoskalering er mulig også her.

Dette er også et image bygget i linux, så vi konfigurerer nodeSelector til å være linux.

Deployment delen av YAML-filen blir da seende slik ut for Admin API:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: adminapi-14
  labels:
    app: adminapi-14
spec:
  replicas: 1
  template:
    metadata:
      name: adminapi-14
      labels:
        app: adminapi-14
    spec:
      containers:
      - name: adminapi-14
        image: nhnkubebachelor/helseid:adminapi-14
        resources:
          requests:
            cpu: .1
```

```
            memory: 300m
        ports:
          - containerPort: 5000
      imagePullSecrets:
      - name: regcred
      nodeSelector:
        "beta.kubernetes.io/os": linux
  selector:
    matchLabels:
      app: adminapi-14
```

8.2.1.2 Konfigurasjon av Service

Nærmest identisk den Service som vi konfigurerte for HelseID STS. Forskjellene er navnet og den deploymenten tjenesten pekes til. Og siden Admin API lytter på port 5000, konfigurerer vi LoadBalancer tjenesten til å eksponere container port 5000 på host port 5000.

```
apiVersion: v1
kind: Service
metadata:
  name: adminapi-14
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 5000
    targetPort: 5000
  selector:
    app: adminapi-14
```

8.2.1.3 Konfigurert YAML-fil

Når vi slår sammen de to delene ender vi opp med en YAML-fil for utrulling av Admin API som ser slik ut:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: adminapi-14
  labels:
    app: adminapi-14
spec:
  replicas: 1
```

```
  template:
    metadata:
      name: adminapi-14
      labels:
        app: adminapi-14
    spec:
      containers:
      - name: adminapi-14
        image: nhnkubebachelor/helseid:adminapi-14
        resources:
          requests:
            cpu: .1
            memory: 300m
        ports:
          - containerPort: 5000
      imagePullSecrets:
      - name: regcred
      nodeSelector:
        "beta.kubernetes.io/os": linux
  selector:
    matchLabels:
      app: adminapi-14
---
apiVersion: v1
kind: Service
metadata:
  name: adminapi-14
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 5000
    targetPort: 5000
  selector:
    app: adminapi-14
```

8.2.2 Rulle ut i Kubernetes

For å rulle ut deployment-en i Kubernetes bruker vi igjen en enkel kubectl kommando, vi endrer bare navnet på YAML-filen vi skal bruke i forhold til slik vi gjorde det med HelseID STS:

```
kubectl create -f hid-adminapi.yaml
```

Når vi kjører den ser vi at der blir laget en deployment og service:

```
PS C:\tools\YAML> kubectl create -f .\hid-adminapi.yaml
deployment.apps/adminapi-14 created
service/adminapi-14 created
```

Figur 78: Rulle ut i Kubernetes Admin API (1/2)

Sjekker at det blir laget en pod som konfigurert og ser at den kjører med kommandoen `kubectl get pods`

```
adminapi-14-7ddb5c566c-wc7pc        1/1     Running             0          3m17s
```

Figur 79: Rulle ut i Kubernetes Admin API (2/2)

Pod er nå oppe å kjører.

8.3 Web Admin

Prosessen med å rulle ut denne applikasjonen blir igjen veldig lik de to tidligere gangene vi har gjort det. Henviser derfor igjen til kapittel 8.1 for mer detaljert beskrivelse og forklaring av de forskjellige delene av YAML-filen. Det kommer også igjen til å bli konfigurert en deployment og en service i samme YAML-fil.

8.3.1 Konfigurasjon av YAML-fil

8.3.1.1 Konfigurasjon av Deployment

Siste stabile versjon av Web Admin finner vi med image nhnadmin-15. Vi konfigurerer derfor YAML-filen til å bruke dette; `nhnkubebachelor/helseid:nhnadmin-15`. Bruker secret regcred for å kunne hente dette image-et fra vårt private repository.

Web Admin er konfigurert til å lytte på port 80, så vi bruker denne porten for containerne som deploymenten skal rulle ut.

Igjen konfigurerer vi deploymenten slik at alle containere i den reserverer 10% av en vCPU og 300Mb minne til hosten den kjører på slik at vi kan teste autoskalering.

Image er bygget i Linux, så nodeSelector blir linux.

Med dette blir deployment delen av YAML-filen slik:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nhnadmin-15
  labels:
    app: nhnadmin-15
spec:
  replicas: 1
  template:
    metadata:
      name: nhnadmin-15
```

```
      labels:
        app: nhnadmin-15
    spec:
      containers:
      - name: nhnadmin-15
        image: nhnkubebachelor/helseid:nhnadmin-15
        resources:
          requests:
            cpu: .1
            memory: 300m
        ports:
          - containerPort: 80
      imagePullSecrets:
      - name: regcred
      nodeSelector:
        "beta.kubernetes.io/os": linux
  selector:
    matchLabels:
      app: nhnadmin-15
```

8.3.1.2 Konfigurasjon av Service

Igjen en Service av typen LoadBalancer og siden vi har konfigurert Web Admin til å lytte på port 80 eksponerer vi container port 80 på host port 80.

```
apiVersion: v1
kind: Service
metadata:
  name: nhnadmin-15
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  selector:
    app: nhnadmin-15
```

8.3.1.3 Konfigurert YAML-fil

Den endelige YAML-filen for utrulling av Web Admin i Kubernetes blir da slik:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nhnadmin-15
  labels:
```

```
    app: nhnadmin-15
spec:
  replicas: 1
  template:
    metadata:
      name: nhnadmin-15
      labels:
        app: nhnadmin-15
    spec:
      containers:
      - name: nhnadmin-15
        image: nhnkubebachelor/helseid:nhnadmin-15
        resources:
          requests:
            cpu: .1
            memory: 300m
        ports:
          - containerPort: 80
      imagePullSecrets:
      - name: regcred
      nodeSelector:
        "beta.kubernetes.io/os": linux
  selector:
    matchLabels:
      app: nhnadmin-15
---
apiVersion: v1
kind: Service
metadata:
  name: nhnadmin-15
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  selector:
    app: nhnadmin-15
```

8.3.2 Rulle ut i Kubernetes

Vi bruker så kommando for å opprette deployment og service ut i fra den YAML-filen vi nettopp konfigurerte:

```
kubectl create -f hid-stsweb.yaml
```

Når vi kjører den ser vi at deployment og service blir opprettet som konfigurert:

```
PS C:\tools\YAML> kubectl create -f .\hid-adminweb.yaml
deployment.apps/nhnadmin-15 created
service/nhnadmin-15 created
```

Figur 80: Rulle ut i Kubernetes Web Admin (1/2)

Så sjekker vi om det kjøres opp en pod med kommando `kubectl get pods`:

```
nhnadmin-15-6fc59dbbc8-xrtnd        1/1     Running             0          2m27s
```

Figur 81: Rulle ut i Kubernetes Web Admin (2/2)

Pod er oppe å kjører.

8.4 Konfigurere DNS

Nå som vi har deployments og services for alle komponentene vi skal bruke for HelseID har vi bare en liten ting igjen å gjøre før vi kan sjekke om alt er oppe og fungerer som det skal. Vi har jo konfigurert kildekoden til de forskjellige applikasjonene til å kommunisere med hverandre på DNS-navn og dette får de ikke automatisk rett ut fra boksen. Så vi må inn i Azure og konfigurere dette selv, nå som vi har rullet ut deployments og services.

Det første vi gjør er å se om alle servicene er oppe å kjører som konfigurert, vi gjøre dette med kommandoen: `kubectl get svc`

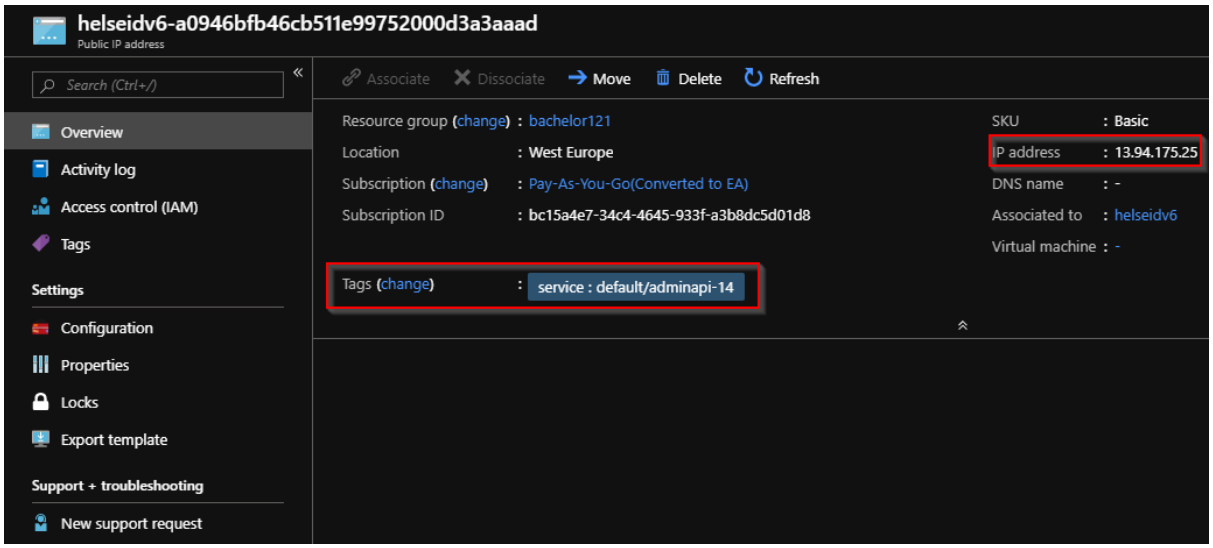
```
PS C:\tools\YAML> kubectl get svc
NAME          TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)          AGE
adminapi-14   LoadBalancer   10.0.47.222    13.94.175.25    5000:31473/TCP   19m
kubernetes    ClusterIP      10.0.0.1       <none>          443/TCP          8d
nhnadmin-15   LoadBalancer   10.0.213.229   13.94.187.117   80:32635/TCP     18m
stsweb-16     LoadBalancer   10.0.225.170   13.80.73.37     443:32458/TCP    17m
```

Figur 82: Kubernetes Services informasjon

Vi ser at alle tre er oppe å kjører og har fått ekstern IP. Vi ser også at korrekte porter er blitt konfigurert på de tre tjenestene.

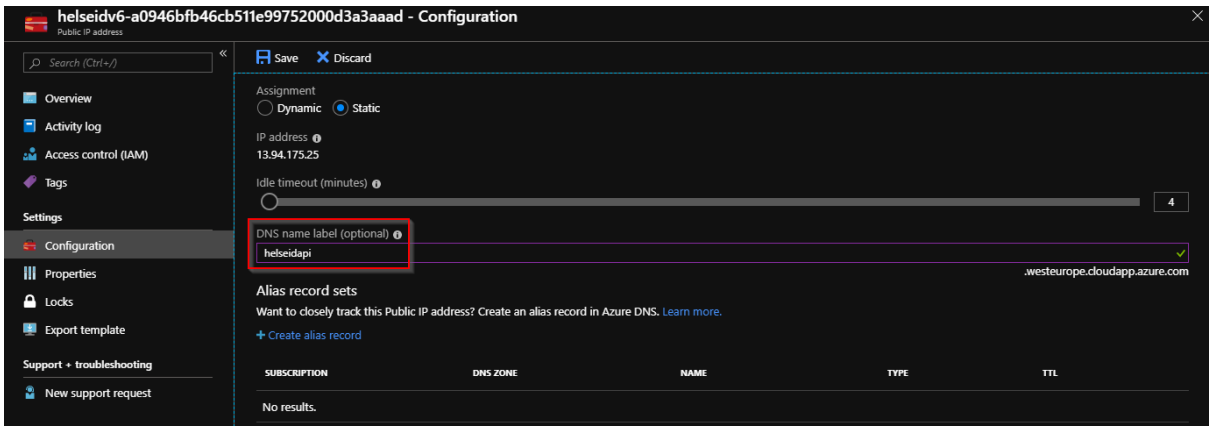
Videre går vi til Azure Portal. Der finner vi at tre nye public IP-adresser er blitt opprettet. Vi går inn på disse, finner IP-adressen eller ser på tag-en og gir dem det DNS-navnet vi har konfigurert i kildekoden.

På den første IP-en ser vi at den er blitt tag-et med adminapi-14 og vi ser at IP-adressen stemmer med det vi ser i Kubernetes.



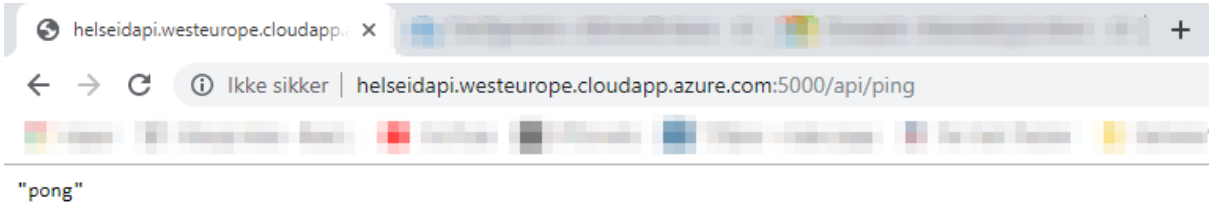
Figur 83: Konfigurasjon DNS (1/2)

Vi går til “Configuration” i menyen og gir den DNS-navnet `helseidapi.westeurope.cloudapp.azure.com` som er det vi har konfigurert i kildekoden at Admin API skal ha.



Figur 84: Konfigurasjon DNS (2/2)

Nå som dette er konfigurert kan vi prøve å åpne det i en nettleser. Vi forsøker adressen `helseidapi.westeurope.cloudapp.azure.com:5000/api/ping`, denne skal nå gi oss et eller annet svar.

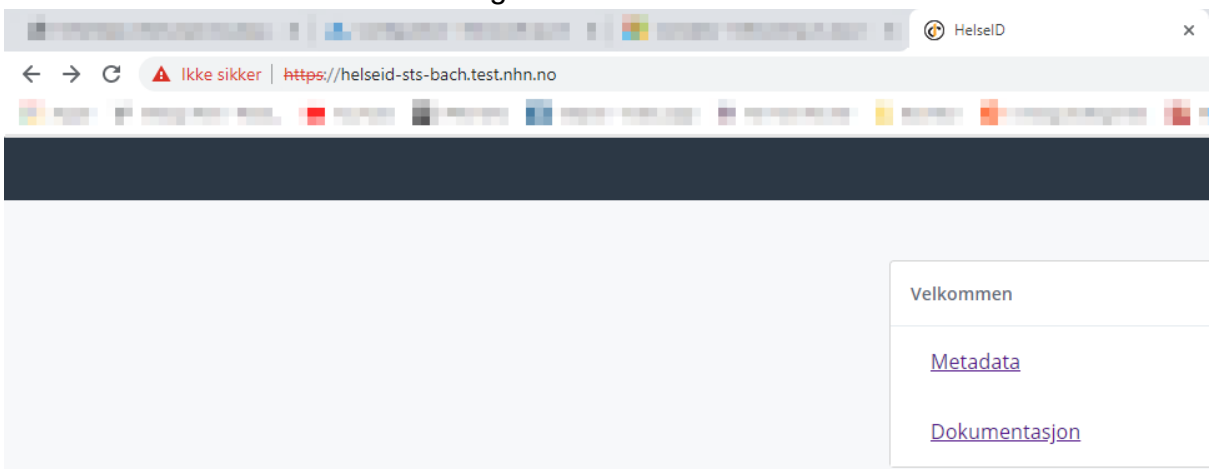


Figur 85: Teste Admin API på DNS-navn

Vi ser at vi har fått svar “pong” fra applikasjonen.

Vi går gjennom samme prosess med HelseID STS og Web Admin.
HelseID STS får DNS-navnet `helseidwebsts.westeurope.cloudapp.azure.com` og Web Admin gir vi `adminhelseid3.westeurope.cloudapp.azure.com`.

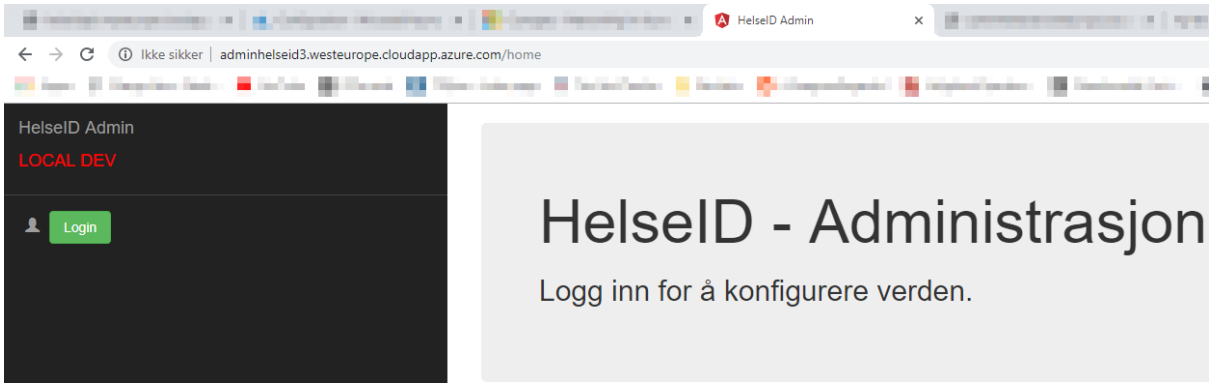
På grunn av integrasjon med ID-porten og det at ID-porten kun godkjenner henvendelser fra spesifikke DNS-navn har `https://helseid-sts-bach.test.nhn.no` blitt satt opp av NHN til å være kobling mellom ID-porten og HelseID STS. Vi forsøker derfor å gå til denne adressen etter vi har fikset DNS-navn.



Figur 86: Teste HelseID STS på DNS-navn

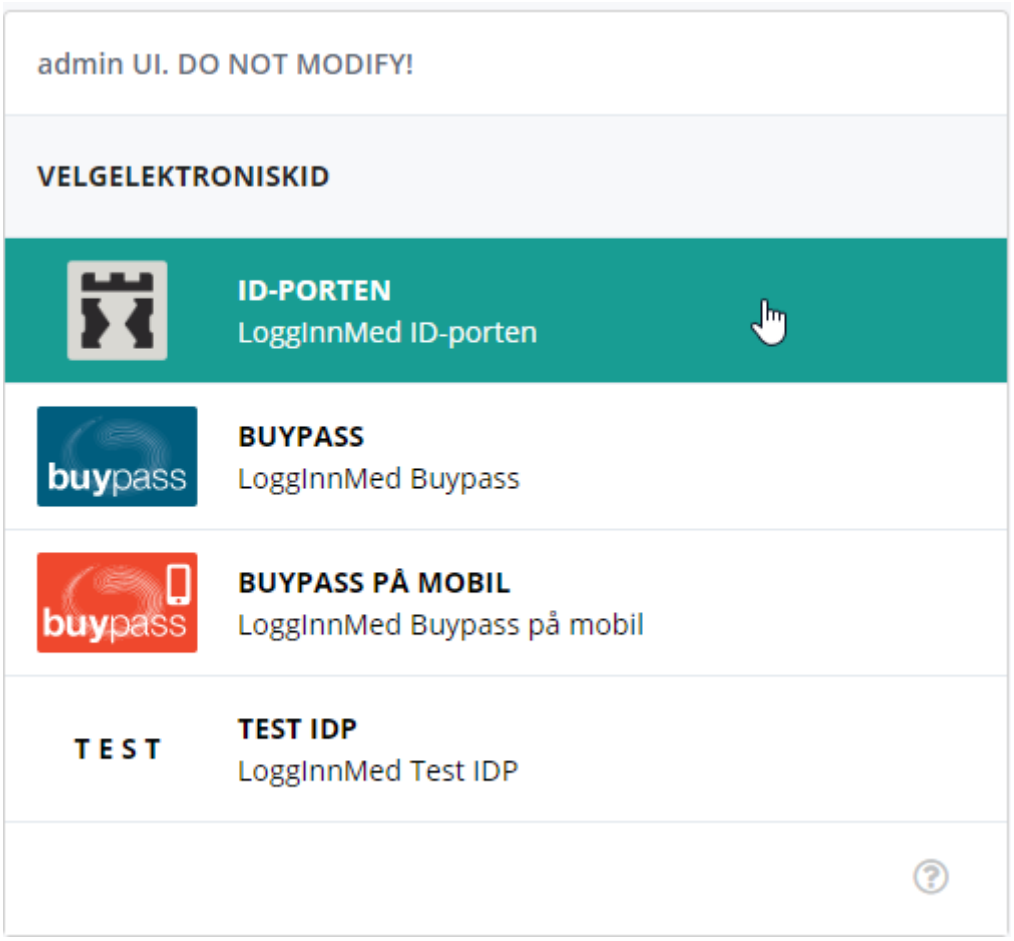
Vi ser koblingen fungerer.

Og til slutt Web Admin.



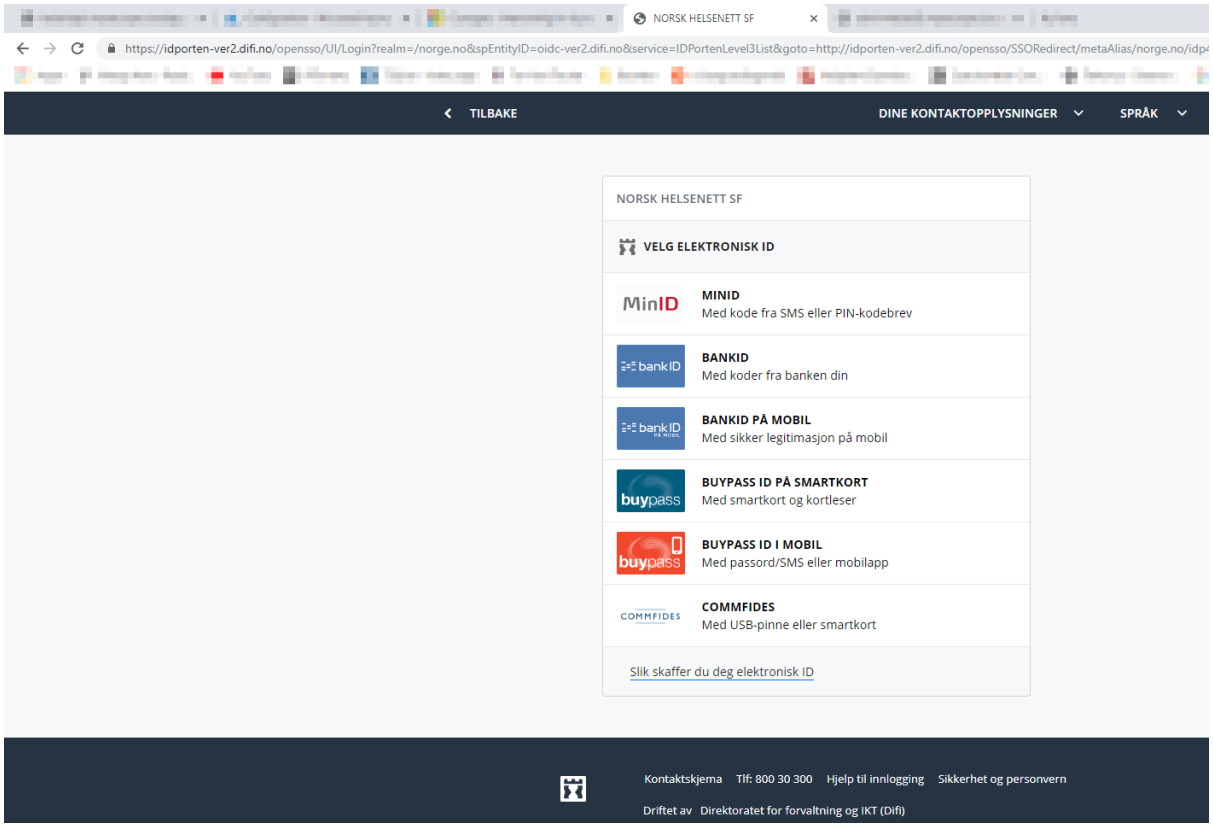
Figur 87: Teste Web Admin på DNS-navn

Flott, alle hovedkomponentene er oppe å går. Vi kan nå forsøke å logge oss inn via ID-porten for å se om integrasjonen der er konfigurert korrekt. Trykker på Login knappen på Web Admin siden.



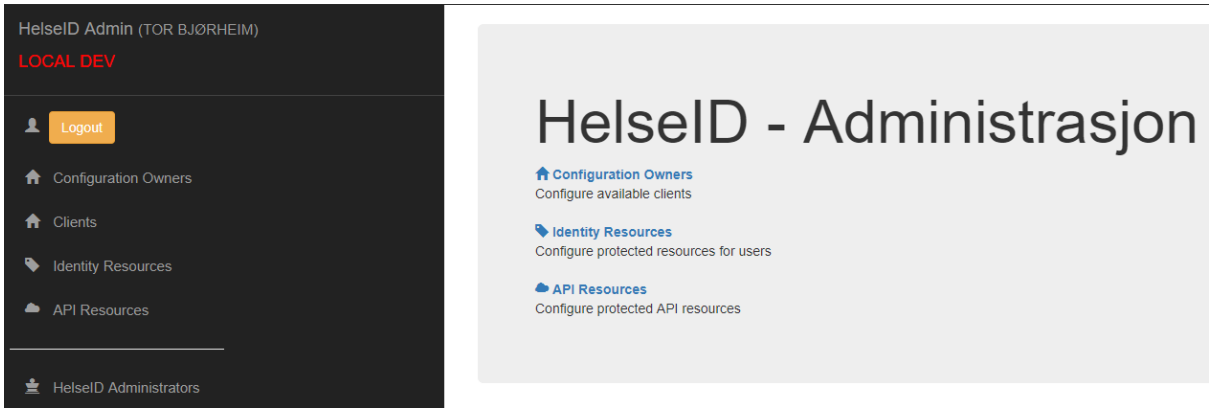
Figur 88: Teste ID-Porten integrasjon (1/3)

Vi velger ID-porten og blir sendt videre til ID-portens sider. Alt ser bra ut så langt.



Figur 89: Teste ID-Porten integrasjon (2/3)

Vi velger BANKID og forsøker å logge inn med testbruker som ble lagt inn i Admin databasen tidligere.



Figur 90: Teste ID-Porten integrasjon (3/3)

Vi er logget inn!

Alle komponenter er oppe å kjører, komponentene kan kommunisere med hverandre og integrasjon med ID-porten er på plass.

9. Skalering

Skalering har vært et av hovedfokusene med denne oppgaven. Spesielt det å se på hvordan Kubernetes behandler skalering i forhold til den tradisjonelle driftsmodellen NHN bruker i dag for drift av since Windows applikasjoner. NHN forventer at brukerbasen til HelseID kommer til å øke kraftig i tiden fremover og da blir mulighetene rundt skalering viktigere og viktigere. Får vi realisert noen av de teoretiske fordelene med Kubernetes med tanke på skalering? Det vil vi prøve å finne ut av nå som vi har konvertert HelseID til å kjøre som containere i Kubernetes. Vi vil se på skalering både på pod-nivå og på node-nivå.

9.1 Skalering på pod-nivå

Til å starte med skal vi prøve oss på skalering på pod-nivå, eller container-nivå som man også kan si. Det endelige målet med denne typen skalering er å konfigurere Kubernetes og deployments i Kubernetes slik at det blir oppdaget automatisk når arbeidslasten på de kjørende containerne blir for høy og det dermed blir kjørt opp nye pods for å hjelpe med å behandle den økte lasten.

9.1.1 Manuell skalering

Til å starte med kan vi teste manuell skalering av en deployment, vi prøver å manuelt skalere deployment for HelseID STS. Dette gjøres med en enkel kommando som ser slik ut:

```
kubectl scale deployment stsweb-16 --replicas=5
```

Vi sjekker status før vi kjører kommandoen og ser at det kun kjøres en HelseID STS pod for øyeblikket

```
PS C:\tools\YAML> kubectl get pods
NAME                          READY   STATUS
adminapi-14-7ddb5c566c-wc7pc  1/1     Running
datadog-agent-689d4           1/1     Running
datadog-agent-b9mr6           1/1     Running
nhnadmin-15-6fc59dbbc8-xrtnd  1/1     Running
stsweb-16-6d4bfc5fc-p28m7     1/1     Running
```

Figur 91: Manuell skalering av pods (1/4)

Vi kjører så kommandoen for skalering og ser hva som skjer:

```
PS C:\tools\YAML> kubectl scale deployment stsweb-16 --replicas=5
deployment.extensions/stsweb-16 scaled
PS C:\tools\YAML> kubectl get pods
NAME                            READY   STATUS              RESTARTS   AGE
adminapi-14-7ddb5c566c-wc7pc    1/1     Running             0          3h16m
datadog-agent-689d4             1/1     Running             4          7d23h
datadog-agent-b9mr6             1/1     Running             4          7d
nhnadmin-15-6fc59dbbc8-xrtnd    1/1     Running             0          3h15m
stsweb-16-6d4bfc5fc-8kk54       0/1     ContainerCreating   0          3s
stsweb-16-6d4bfc5fc-mc41c       0/1     ContainerCreating   0          3s
stsweb-16-6d4bfc5fc-p28m7       1/1     Running             0          3h14m
stsweb-16-6d4bfc5fc-tcchx       0/1     ContainerCreating   0          3s
stsweb-16-6d4bfc5fc-tchgn       0/1     ContainerCreating   0          3s
PS C:\tools\YAML> kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
adminapi-14-7ddb5c566c-wc7pc    1/1     Running   0          3h16m
datadog-agent-689d4             1/1     Running   4          7d23h
datadog-agent-b9mr6             1/1     Running   4          7d
nhnadmin-15-6fc59dbbc8-xrtnd    1/1     Running   0          3h15m
stsweb-16-6d4bfc5fc-8kk54       1/1     Running   0          13s
stsweb-16-6d4bfc5fc-mc41c       1/1     Running   0          13s
stsweb-16-6d4bfc5fc-p28m7       1/1     Running   0          3h14m
stsweb-16-6d4bfc5fc-tcchx       1/1     Running   0          13s
stsweb-16-6d4bfc5fc-tchgn       1/1     Running   0          13s
```

Figur 92: Manuell skalering av pods (2/4)

Fire nye pods er oppe å kjører innen 15 sekunder. De kan nå hjelpe med å behandle last på HelseID STS.

Vi skalerer så tilbake til en pod slik det er konfigurert i deploymenten og ser at de begynner å bli slettet med en gang.

```
PS C:\tools\YAML> kubectl scale deployment stsweb-16 --replicas=1
deployment.extensions/stsweb-16 scaled
PS C:\tools\YAML> kubectl get pods
NAME                            READY   STATUS        RESTARTS   AGE
adminapi-14-7ddb5c566c-wc7pc    1/1     Running       0          3h20m
datadog-agent-689d4             1/1     Running       4          7d23h
datadog-agent-b9mr6             1/1     Running       4          7d
nhnadmin-15-6fc59dbbc8-xrtnd    1/1     Running       0          3h19m
stsweb-16-6d4bfc5fc-8kk54       1/1     Terminating   0          4m24s
stsweb-16-6d4bfc5fc-mc41c       1/1     Terminating   0          4m24s
stsweb-16-6d4bfc5fc-p28m7       1/1     Running       0          3h19m
stsweb-16-6d4bfc5fc-tcchx       1/1     Terminating   0          4m24s
stsweb-16-6d4bfc5fc-tchgn       1/1     Terminating   0          4m24s
```

Figur 93: Manuell skalering av pods (3/4)

Innen 15 sekunder er vi tilbake til der vi startet

```
PS C:\tools\YAML> kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
adminapi-14-7ddb5c566c-wc7pc    1/1     Running   0          3h20m
datadog-agent-689d4             1/1     Running   4          7d23h
datadog-agent-b9mr6             1/1     Running   4          7d
nhnadmin-15-6fc59dbbc8-xrtnd    1/1     Running   0          3h19m
stsweb-16-6d4bfc5fc-p28m7       1/1     Running   0          3h19m
```

Figur 94: Manuell skalering av pods (4/4)

Så enkelt er det med manuell skalering. Og mens dette kan være ekstremt nyttig i noen tilfeller, vil vi jo heller at skalering skjer automatisk med en gang det blir

oppdaget uforventet høy last på containerne i Kubernetes. Målet er å kunne sømløst skalere opp og ned ut i fra behov uten at brukerne av HelseID merker noe som helst. Manuell skalering kan bli litt for tregt for dette, og krever konstant monitorering og en person må være tilgjengelig for å utføre dette.

9.1.2 Automatisk skalering

For å kunne teste automatisk skalering må vi først konfigurere en HPA (Horizontal Pod Autoscaler). Vi bruker igjen her HelseID STS for å teste skalering, denne gangen automatisk skalering. For å konfigurere en HPA for HelseID STS bruker vi kommando:

```
kubectl autoscale deployment stsweb-16 --min=1 --max=20 --cpu-percent=50
```

Denne kommandoen lager en HPA, som skalerer opp eller ned antall pods som kjører i deployment for HelseID STS. Den bestemmer antall pods som skal kjøres, ut i fra observert CPU-bruk. Vi har satt cpu-percent til 50. Dette betyr at når cpu-bruk av deploymenten går over 50% av det som totalt er reservert av alle pods i deployment, da blir det kjørt opp nye pods for å hjelpe med å behandle last. Vi setter også minst antall pods til 1 og max til 20.

Vi kjører kommandoen og HPA blir opprettet:

```
PS C:\tools\YAML> kubectl autoscale deployment stsweb-16 --min=1 --max=20 --cpu-percent=50
horizontalpodautoscaler.autoscaling/stsweb-16 autoscaled
```

Figur 95: Automatisk skalering av pods (1/7)

Vi sjekker at alt ser ok ut med `kubectl get hpa`:

```
PS C:\tools\YAML> kubectl get hpa
NAME         REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
stsweb-16    Deployment/stsweb-16    0%/50%    1         20        1          11m
```

Figur 96: Automatisk skalering av pods (2/7)

Nå som HPA er konfigurert kan vi teste hva som skjer hvis HelseID STS plutselig mottar en overflod av https henvendelser samtidig uten stopp. Vi bruker verktøyet Siege til å gjøre dette.

Etter rundt 30 sekunder sjekker vi status på pods og status på HPA.

```
NAME         REFERENCE               TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
stsweb-16    Deployment/stsweb-16    444%/50%   1         20        1          19m
```

Figur 97: Automatisk skalering av pods (3/7)

Vi ser at observert CPU-bruk er langt over det vi satte HPA-en skulle se etter. Vi sjekker om det er kjørt ut noen ny pods.

```
PS C:\tools\YAML> kubectl get pods -o wide
NAME                            READY   STATUS    RESTARTS   AGE
adminapi-14-7ddb5c566c-jswmz    1/1     Running   0          79m
datadog-agent-689d4             1/1     Running   4          8d
datadog-agent-b9mr6             1/1     Running   4          7d2h
nhnadmin-15-6fc59dbbc8-xrtnd    1/1     Running   0          5h1m
stsweb-16-6955d75cdf-6d4cb      1/1     Running   0          24m
stsweb-16-6955d75cdf-jq6dq      1/1     Running   0          10s
stsweb-16-6955d75cdf-pg68t      1/1     Running   0          10s
stsweb-16-6955d75cdf-vrv2m      1/1     Running   0          10s
```

Figur 98: Automatisk skalering av pods (4/7)

Her ser vi at 3 nye pods er kjørt ut, vi har nå 4 totalt. Men dette stemmer ikke helt opp med de tallene vi så tidligere. 444% er jo over åtte ganger så mye som 50%. Vi venter 10-15 sekunder til og sjekker på nytt.

```
PS C:\tools\YAML> kubectl get hpa
NAME        REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
stsweb-16   Deployment/stsweb-16   47%/50%   1         20        9          22m
PS C:\tools\YAML> kubectl get pods -o wide
NAME                            READY   STATUS    RESTARTS   AGE     IP
adminapi-14-7ddb5c566c-jswmz    1/1     Running   0          82m     10.240.0.98
datadog-agent-689d4             1/1     Running   4          8d      10.240.0.105
datadog-agent-b9mr6             1/1     Running   4          7d2h    10.240.0.203
nhnadmin-15-6fc59dbbc8-xrtnd    1/1     Running   0          5h4m    10.240.0.101
stsweb-16-6955d75cdf-6d4cb      1/1     Running   0          27m     10.240.0.106
stsweb-16-6955d75cdf-8kqdc      1/1     Running   0          2m43s   10.240.0.118
stsweb-16-6955d75cdf-9rn7f      1/1     Running   0          2m43s   10.240.0.102
stsweb-16-6955d75cdf-bz5pv      1/1     Running   0          2m27s   10.240.0.113
stsweb-16-6955d75cdf-cfjc4      1/1     Running   0          2m43s   10.240.0.191
stsweb-16-6955d75cdf-jq6dq      1/1     Running   0          2m58s   10.240.0.112
stsweb-16-6955d75cdf-pg68t      1/1     Running   0          2m58s   10.240.0.190
stsweb-16-6955d75cdf-q982g      1/1     Running   0          2m43s   10.240.0.204
stsweb-16-6955d75cdf-vrv2m      1/1     Running   0          2m58s   10.240.0.195
```

Figur 99: Automatisk skalering av pods (5/7)

Totalt 9 pods for HelseID STS kjører og vi ser at HPA-en har stabilisert seg på 47%/50%. Nå som vi har kjørt ut 8 nye pods, blir den totale mengden observert CPU-bruk under 50% av den sammenlagte reserverte mengden CPU for hele deploymenten. HPA-en kjører dermed ikke ut flere pods for å behandle den økte lasten. Dette stemmer godt opp med det initielle tallet på 444% som vi så tidligere. 8 nye pods ble kjørt opp innen 1 minutt, flere av dem innen 30 sekunder. Hvis vi nå derimot hadde økt lasten litt igjen hadde det nok blitt kjørt opp flere nye pods, da den ligger veldig nærme den grensen vi satte på 50%.

Pods kjørt opp av en autoscaler har en grace-periode på 5 minutter. Dette betyr at selv om lasten på deploymenten går tilbake til normale verdier gir HPA-en containerne minst 5 minutter før den begynner å skalere ned igjen. Vi sjekker status et minutt eller to etter vi stoppet verktøyet Siege og dermed overfloden av henvendelser.

```
PS C:\tools\YAML> kubectl get hpa
NAME        REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
stsweb-16   Deployment/stsweb-16   0%/50%    1         20        9          33m
PS C:\tools\YAML> kubectl get pods -o wide
NAME                            READY   STATUS    RESTARTS   AGE     IP
adminapi-14-7ddb5c566c-jswmz    1/1     Running   0          92m     10.240.0.98
datadog-agent-689d4             1/1     Running   4          8d      10.240.0.105
datadog-agent-b9mr6             1/1     Running   4          7d2h    10.240.0.203
nhnadmin-15-6fc59dbbc8-xrtnd    1/1     Running   0          5h14m   10.240.0.101
stsweb-16-6955d75cdf-6d4cb      1/1     Running   0          38m     10.240.0.106
stsweb-16-6955d75cdf-8kqdc      1/1     Running   0          13m     10.240.0.118
stsweb-16-6955d75cdf-9rn7f      1/1     Running   0          13m     10.240.0.102
stsweb-16-6955d75cdf-bz5pv      1/1     Running   0          12m     10.240.0.113
stsweb-16-6955d75cdf-cfjc4      1/1     Running   0          13m     10.240.0.191
stsweb-16-6955d75cdf-jq6dq      1/1     Running   0          13m     10.240.0.112
stsweb-16-6955d75cdf-pg68t      1/1     Running   0          13m     10.240.0.190
stsweb-16-6955d75cdf-q982g      1/1     Running   0          13m     10.240.0.204
stsweb-16-6955d75cdf-vrv2m      1/1     Running   0          13m     10.240.0.195
```

Figur 100: Automatisk skalering av pods (6/7)

Vi ser at Kubernetes observerer 0% CPU-bruk, men at de pods som ble kjørt opp tidligere fortsatt kjører.
Hvis vi venter ut grace-perioden og sjekker igjen 4-5 minutter senere ser vi dette resultatet:

```
PS C:\tools\YAML> kubectl get hpa
NAME        REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
stsweb-16   Deployment/stsweb-16   0%/50%    1         20        1          38m
PS C:\tools\YAML> kubectl get pods -o wide
NAME                            READY   STATUS    RESTARTS   AGE     IP
adminapi-14-7ddb5c566c-jswmz    1/1     Running   0          97m     10.240.0.98
datadog-agent-689d4             1/1     Running   4          8d      10.240.0.105
datadog-agent-b9mr6             1/1     Running   4          7d2h    10.240.0.203
nhnadmin-15-6fc59dbbc8-xrtnd    1/1     Running   0          5h19m   10.240.0.101
stsweb-16-6955d75cdf-6d4cb      1/1     Running   0          43m     10.240.0.106
PS C:\tools\YAML>
```

Figur 101: Automatisk skalering av pods (7/7)

Grace-perioden er nå over og HPA-en har skalert ned til 1 pod igjen som konfigurert som minimum i både Deployment og HPA.

Autoskalering på pod-nivå ser ut til å fungere utmerket og vi går videre til skalering på node-nivå.

9.2 Skalering på node-nivå

Når vi utførte testen på pod skalering hadde vi nok ressurser tilgjengelig på de to linux worker nodene som kjører i vårt Kubernetes cluster. Men hva skjer hvis lasten hadde vært større og HPA-en hadde forsøkt å rulle ut den maksimale mengden på 20 pods som vi konfigurerte i stedet for “bare” 9. Det er ikke nok ressurser tilgjengelig på de to worker nodene til å kjøre ut alle 20 pods. Og det er da den addon-en vi la til når vi konfigurerte AKS Engine kommer inn i bildet, Cluster Autoscaler. Med denne implementert i clusteret vårt vil da nye noder bli provisjonert i Azure og lagt inn i clusteret for å kunne kjøre ut alle pods som HPA-en ønsker. Dette er kun om vi har konfigurert Cluster Autoscaler korrekt så klart. Vi skal nå teste dette.

Før vi starter sjekker vi status på noder i clusteret.

```
PS C:\tools\YAML> kubectl get nodes
NAME                                STATUS   ROLES    AGE    VERSION
2029k8s00000001                     Ready    agent    8d     v1.13.5
2029k8s00000002                     Ready    agent    7d3h   v1.13.5
k8s-linuxpool2-20297044-vmss000000  Ready    agent    8d     v1.13.5
k8s-linuxpool2-20297044-vmss000002  Ready    agent    7d3h   v1.13.5
k8s-master-20297044-0               Ready    master   8d     v1.13.5
```

Figur 102: Automatisk skalering av noder (1/9)

Vi ser at 2 linux worker noder kjører.

Vi øker lasten slik at 20 pods blir rullet ut og ser hva som skjer.

```
PS C:\tools\YAML> kubectl get pods
NAME                              READY   STATUS              RESTARTS   AGE
adminapi-14-7ddb5c566c-jswmz      1/1     Running             0          121m
datadog-agent-689d4               1/1     Running             4          8d
datadog-agent-b9mr6               1/1     Running             4          7d3h
nhnadmin-15-6fc59dbbc8-xrtnd      1/1     Running             0          5h43m
stsweb-16-6955d75cdf-56v9s        0/1     Pending             0          9s
stsweb-16-6955d75cdf-64sxt        0/1     ContainerCreating   0          9s
stsweb-16-6955d75cdf-6d4cb        1/1     Running             0          66m
stsweb-16-6955d75cdf-cq776        0/1     Pending             0          9s
stsweb-16-6955d75cdf-czrhv        0/1     ContainerCreating   0          9s
stsweb-16-6955d75cdf-d7whd        0/1     ContainerCreating   0          9s
stsweb-16-6955d75cdf-dn8ds        0/1     ContainerCreating   0          9s
stsweb-16-6955d75cdf-gjk8d        0/1     ContainerCreating   0          9s
stsweb-16-6955d75cdf-jkv72        0/1     Pending             0          9s
stsweb-16-6955d75cdf-n4658        0/1     ContainerCreating   0          9s
stsweb-16-6955d75cdf-nvptt        0/1     Pending             0          9s
stsweb-16-6955d75cdf-qq5v4        0/1     Pending             0          9s
stsweb-16-6955d75cdf-rc4rs        0/1     Pending             0          9s
stsweb-16-6955d75cdf-s6k6t        0/1     Pending             0          9s
stsweb-16-6955d75cdf-t75xc        0/1     ContainerCreating   0          9s
stsweb-16-6955d75cdf-vmksz        0/1     ContainerCreating   0          9s
stsweb-16-6955d75cdf-w86mp        0/1     ContainerCreating   0          9s
stsweb-16-6955d75cdf-wr4sc        0/1     ContainerCreating   0          9s
stsweb-16-6955d75cdf-xbcfh        0/1     Pending             0          9s
stsweb-16-6955d75cdf-xr5fv        0/1     Pending             0          9s
```

Figur 103: Automatisk skalering av noder (2/9)

Vi ser at Kubernetes forsøker å kjøre opp 20 pods i deploymenten stsweb-16. Men flere av dem står med status “Pending”. Vi undersøker en av de som har denne statusen for å se hvorfor det er tilfellet med kommando

```
kubectl describe pod stsweb-16-6955d75cdf-56v9s.
```

Vi får dette resultatet:

```
Message
-------
0/5 nodes are available: 1 node(s) had taints that the pod didn't tolerate, 2 Insufficient cpu, 2 node(s) didn't match node selector.
```

Figur 104: Automatisk skalering av noder (3/9)

Vi ser av feilmeldingen at poden ikke finner noen noder som har nok tilgjengelig CPU til å kjøre.

Etter å ha ventet 4-5 minutter sjekker vi status på pods igjen

```
PS C:\tools\YAML> kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
adminapi-14-7ddb5c566c-jswmz      1/1     Running   0          127m
datadog-agent-2xr8v               1/1     Running   0          3m16s
datadog-agent-689d4               1/1     Running   4          8d
datadog-agent-b9mr6               1/1     Running   4          7d3h
datadog-agent-swbtm               1/1     Running   0          3m11s
nhnadmin-15-6fc59dbbc8-xrtnd      1/1     Running   0          5h49m
stsweb-16-6955d75cdf-56v9s        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-64sxt        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-6d4cb        1/1     Running   0          72m
stsweb-16-6955d75cdf-cq776        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-czrhv        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-d7whd        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-dn8ds        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-gjk8d        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-jkv72        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-n4658        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-nvptt        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-qq5v4        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-rc4rs        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-s6k6t        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-t75xc        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-vmksz        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-w86mp        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-wr4sc        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-xbcfh        1/1     Running   0          6m31s
stsweb-16-6955d75cdf-xr5fv        1/1     Running   0          6m31s
```

Figur 105: Automatisk skalering av noder (4/9)

Alle 20 pods kjører. Vi ser mer detaljert på samme pod som vi gjorde tidligere med samme `kubectl describe pod` kommando

```
Message
-------
0/5 nodes are available: 1 node(s) had taints that the pod didn't tolerate, 2 Insufficient cpu, 2 node(s) didn't match node selector.
0/6 nodes are available: 2 Insufficient cpu, 2 node(s) didn't match node selector, 2 node(s) had taints that the pod didn't tolerate.
Successfully assigned default/stsweb-16-6955d75cdf-56v9s to k8s-linuxpool2-20297044-vmss000008
pod triggered scale-up: [{k8s-linuxpool2-20297044-vmss 2->4 (max: 5)}]
pulling image "nhnkubebachelor/helseid:stsweb-16"
Successfully pulled image "nhnkubebachelor/helseid:stsweb-16"
Created container
Started container
```

Figur 106: Automatisk skalering av noder (5/9)

```
Events:
  Type     Reason            Age                  From
  ----     ------            ----                 ----
  Warning  FailedScheduling  5m47s (x6 over 8m14s)  default-scheduler
  Warning  FailedScheduling  5m                   default-scheduler
  Normal   Scheduled         4m59s                default-scheduler
  Normal   TriggeredScaleUp  4m52s                cluster-autoscaler
  Normal   Pulling           4m41s                kubelet, k8s-linuxpool2-20297044-vmss000008
  Normal   Pulled            3m48s                kubelet, k8s-linuxpool2-20297044-vmss000008
  Normal   Created           3m24s                kubelet, k8s-linuxpool2-20297044-vmss000008
  Normal   Started           3m24s                kubelet, k8s-linuxpool2-20297044-vmss000008
```

Figur 107: Automatisk skalering av noder (6/9)

Vi ser at den fortsatte å lete etter en godkjent node med tilgjengelig ressurser helt til den fant en. Denne pod var med på å trigge Cluster Autoscaler til å kjøre opp nye noder og vi ser at det tok under 5 minutter fra poden ble forsøkte kjørt opp til den faktisk kjører i clusteret på en nyopprettet node.

Vi sjekker status på noder i clusteret for å bekrefte at det nå er mer enn to Linux worker noder.

```
PS C:\tools\YAML> kubectl get nodes
NAME                                STATUS   ROLES    AGE    VERSION
2029k8s00000001                     Ready    agent    8d     v1.13.5
2029k8s00000002                     Ready    agent    7d3h   v1.13.5
k8s-linuxpool2-20297044-vmss000000  Ready    agent    8d     v1.13.5
k8s-linuxpool2-20297044-vmss000002  Ready    agent    7d3h   v1.13.5
k8s-linuxpool2-20297044-vmss000007  Ready    agent    13m    v1.13.5
k8s-linuxpool2-20297044-vmss000008  Ready    agent    13m    v1.13.5
k8s-master-20297044-0               Ready    master   8d     v1.13.5
PS C:\tools\YAML>
```

Figur 108: Automatisk skalering av noder (7/9)

Vi ser nå at det kjører fire Linux worker noder i clusteret. To nye er blitt opprettet i clusteret for å få kjørt opp alle HelseID STS podene slik at den økte lasten ikke påvirker brukerne av HelseID.

Vi stopper så alle henvendelser til HelseID STS og ser etter rundt 5 minutter at vi er tilbake til å kjøre kun en pod for denne deploymenten.

```
PS C:\tools\YAML> kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
adminapi-14-7ddb5c566c-jswmz    1/1     Running   0          140m
datadog-agent-2xr8v             1/1     Running   0          16m
datadog-agent-689d4             1/1     Running   4          8d
datadog-agent-b9mr6             1/1     Running   4          7d3h
datadog-agent-swbtm             1/1     Running   0          16m
nhnadmin-15-6fc59dbbc8-xrtnd    1/1     Running   0          6h2m
stsweb-16-6955d75cdf-6d4cb      1/1     Running   0          85m
```

Figur 109: Automatisk skalering av noder (8/9)

Grace-perioden for å skalere ned noder er litt lengre enn for pods, som regel 10 minutter. Dette er fordi skalering av noder tar lengre tid og bruker mer ressurser enn skalering av pods og vi vil derfor være helt sikre på at perioden med økt last faktisk er helt over før det skaleres ned igjen.

Vi sjekker status på noder etter rundt 10 minutter

```
PS C:\tools\YAML> kubectl get nodes
NAME                                STATUS   ROLES    AGE
2029k8s00000001                     Ready    agent    9d
2029k8s00000002                     Ready    agent    7d3h
k8s-linuxpool2-20297044-vmss000000  Ready    agent    9d
k8s-linuxpool2-20297044-vmss000002  Ready    agent    7d3h
k8s-master-20297044-0               Ready    master   9d
```

Figur 110: Automatisk skalering av noder (9/9)

Vi ser det er tilbake til kun to linux worker noder som vanlig. Skalering på node-nivå fungerer også flott.

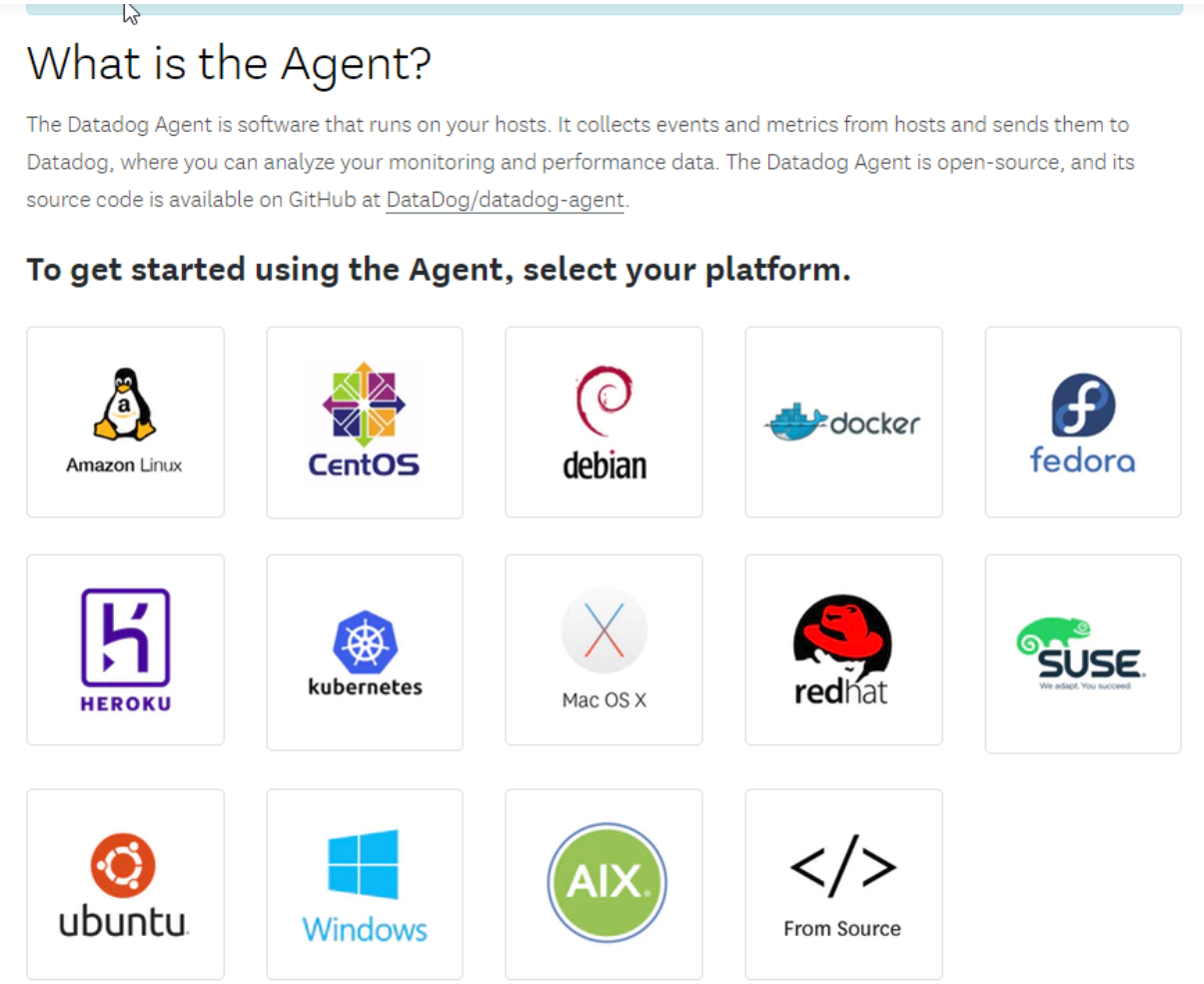
10. Monitorering

Vi kom i designrapporten frem til at vi skal bruke verktøyet Datadog for å monitorere vårt Kubernetes cluster.

10.1 Installasjonsmetode

Datadog tilbyr 14 dagers gratis prøveperiode, som vi har opprettet flere ganger under bachelor perioden. Etter å ha opprettet en konto er vi nødt til å lage en kobling mellom Datadog og vårt Kubernetes miljø.

Datadog kaller integrasjoner for Agents, og en raskt titt på hjemmesiden deres viser de har agents til mange forskjellige plattformer. Vi er dog mest interessert i Kubernetes.



Figur 111: Valg av Datadog Agent (Datadog, 2019)

Vi får valget mellom å sette opp Datadog på hostnivå eller som en container (datadog agenten kjører som en Pod). De anbefaler selv som en container, og siden vi uansett jobber med containere så faller det naturlig for oss å velge en container basert installasjon. Å kjøre agenten som en Pod er tilstrekkelig for de fleste bruksscenarioer, men det er verdt å merke seg at det ikke gir synlighet til

komponenter i systemet som eksisterer utenfor Kubernetes. En slik løsning vil heller ikke gi mulighet å se på oppstartingsfasen av Kubernetes clusteret ditt. Allikevel er det den mest anbefalte måten å installere agenten på.

10.2 Konfigurasjon av RBAC og Secret

Siden vårt Kubernetes cluster har enablet RBAC, så må vi legge til RBAC rettigheter på vår Datadog service account.

Vi starter med å laste ned ClusterRole, ServiceAccount, and ClusterRoleBinding fra Datadog sitt GitHub repository.

Vi laster ned følgende filer på Kubernetes masteren

Clusterrole

```
https://raw.githubusercontent.com/DataDog/datadog-
agent/master/Dockerfiles/manifests/rbac/clusterrole.yaml
```

ServiceAccount

```
https://raw.githubusercontent.com/DataDog/datadog-
agent/master/Dockerfiles/manifests/rbac/serviceaccount.yaml
```

ClusterRoleBinding

```
https://raw.githubusercontent.com/DataDog/datadog-
agent/master/Dockerfiles/manifests/rbac/clusterrolebinding.yaml
```

Vi lager så disse rollene med å bruke kommandoen

```
kubectl create -f ./clusterrolebinding.yaml -f ./clusterrole.yaml -f
./serviceaccount.yaml
```

Output fra kommandoen over vises i bildet fra Putty

```
kube@k8s-master-20297044-0:~$ kubectl create -f ./clusterrolebinding.yaml -f ./clusterrole.yaml -f ./serviceaccount.yaml
clusterrolebinding.rbac.authorization.k8s.io/datadog-agent created
clusterrole.rbac.authorization.k8s.io/datadog-agent created
serviceaccount/datadog-agent created
kube@k8s-master-20297044-0:~$
```

Figur 112: Konfigurasjon RBAC for Datadog

Nå som alle serviceaccounts er på plass trenger vi å lage en secret som lagrer API nøkkelen vår. Vi bruker kubectl create secret til dette.

Vi henter ut API nøkkelen inne på datadog sine konto sider. Vi lager secreten med følgende kommando

```
kubectl create secret generic datadog-secret --from-literal api-key="#SECRET"
```

Bruker vi kommandoen `kubectl get secret` ser vi at har en secret kalt datadog-secret

```
kube@k8s-master-20297044-0:~$ kubectl get secret
NAME                           TYPE                                  DATA   AGE
datadog-agent-token-qjzrd      kubernetes.io/service-account-token   3      9m56s
datadog-secret                 Opaque                                1      10s
default-token-qxjk4            kubernetes.io/service-account-token   3      19h
```

Figur 113: Konfigurasjon secret for Datadog

10.3 Installasjon og konfigurasjon av Datadog agenten

For å kjøre ut en datadog agent på hostene våre trenger vi å lage et manifest for denne agenten som skal kjøre. Vi velger å bruke et DaemonSet, fordi vi ønsker en Pod/Agent på hver node, også på nye noder som blir opprettet ved for eksempel automatisk skalering. Under secretKeyRef så blir navnet på secreten vi lagde i forrige kapittel hentet (datadog-secret). Manifestet ser slik ut:

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: datadog-agent
spec:
  template:
    metadata:
      labels:
        app: datadog-agent
      name: datadog-agent
    spec:
      serviceAccountName: datadog-agent
      containers:
      - image: datadog/agent:latest
        imagePullPolicy: Always
        name: datadog-agent
        ports:
          - containerPort: 8125
            name: dogstatsdport
            protocol: UDP
          - containerPort: 8126
            name: traceport
            protocol: TCP
        env:
          - name: DD_API_KEY
            valueFrom:
              secretKeyRef:
                name: datadog-secret
                key: api-key
          - name: DD_COLLECT_KUBERNETES_EVENTS
            value: "true"
          - name: DD_LEADER_ELECTION
            value: "true"
          - name: KUBERNETES
```

```
            value: "true"
          - name: DD_KUBERNETES_KUBELET_HOST
            valueFrom:
              fieldRef:
                fieldPath: status.hostIP
          - name: DD_APM_ENABLED
            value: "true"
        resources:
          requests:
            memory: "256Mi"
            cpu: "200m"
          limits:
            memory: "256Mi"
            cpu: "200m"
        volumeMounts:
          - name: dockersocket
            mountPath: /var/run/docker.sock
          - name: procdir
            mountPath: /host/proc
            readOnly: true
          - name: cgroups
            mountPath: /host/sys/fs/cgroup
            readOnly: true
        livenessProbe:
          exec:
            command:
            - ./probe.sh
          initialDelaySeconds: 15
          periodSeconds: 5
      volumes:
        - hostPath:
            path: /var/run/docker.sock
          name: dockersocket
        - hostPath:
            path: /proc
          name: procdir
        - hostPath:
            path: /sys/fs/cgroup
          name: cgroups
```

Vi kjører så denne kommandoen for å kjøre ut dette daemonsettet

```
kubectl create -f datadog-agent.yaml
```

Vi sjekker så status på det vi nettopp kjørte ut

```
kubectl get daemonset
```

```
kube@k8s-master-20297044-0:~$ kubectl get daemonset
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
datadog-agent   4         4         2       4            2           <none>          48s
```

Figur 114: Installasjon og konfigurasjon Datadog (1/3)

Vi ser fra bildet over at det er ønskelig med fire pods, fordi vi har to linux VMer og to Windows VMer. Men vi ser at vi bare har to som kjører. Vi bruker Kubernetes kommandoen `kubectl get pods`, til å finne ut hvorfor.

```
kube@k8s-master-20297044-0:~$ kubectl get pods

NAME                                READY   STATUS         RESTARTS   AGE
datadog-agent-dgkfl                 0/1     ErrImagePull   0          65s
datadog-agent-hxv65                 0/1     ErrImagePull   0          65s
datadog-agent-mp6sc                 1/1     Running        0          65s
datadog-agent-vsvfj                 1/1     Running        0          65s
iis2-79d9f8cd49-2lzk5               1/1     Running        0          98m
iis2-79d9f8cd49-gnrj9               1/1     Running        0          98m
iis2-79d9f8cd49-lv8b5               1/1     Running        0          98m
iis2-79d9f8cd49-w46v2               1/1     Running        0          98m
nginx-deployment-59b6966569-2vlrk   1/1     Running        0          91m
nginx-deployment-59b6966569-wnspt   1/1     Running        0          91m
```

Figur 115: Installasjon og konfigurasjon Datadog (2/3)

Som vi ser fra bildet har vi fått statusen “ErrImagePull” på to av Podene. Vi kan undersøke nærmere med kommandoen `kubectl describe pod datadogagent-dgkfl`

Vi ser at image-et ikke vil kjøre ut, og vi finner ut at det er fordi datadog manifestet vi kjører bare fungerer på linux hoster, og ikke på Windows hoster.

```
node.kubernetes.io/unschedulable:NoSchedule
nts:
ype     Reason          Age                 From                        Message
---     ------          ----                ----                        -------
ormal   Scheduled       11m                 default-scheduler           Successfully assigned default/datadog-agent-dgkfl to 202
s00000001
ormal   SandboxChanged  10m (x2 over 10m)   kubelet, 2029k8s00000001    Pod sandbox changed, it will be killed and re-created.
ormal   Pulling         9m20s (x4 over 10m) kubelet, 2029k8s00000001    pulling image "datadog/agent:latest"
arning  Failed          9m18s (x4 over 10m) kubelet, 2029k8s00000001    Error: ErrImagePull
arning  Failed          39s (x43 over 10m)  kubelet, 2029k8s00000001    Back-off pulling image "datadog/agent:latest"
```

Figur 116: Installasjon og konfigurasjon Datadog (3/3)

Vi kan bruke Windows agenter, men dette må legges inn manuelt på OS nivå. Siden vi ønsker automatisk skalering av noder som blir med i clusteret, så blir det ekstremt tungvint å få til det slik vi ønsker med Windows hoster.

10.4 Logging

I tillegg til metrics, trenger vi også å aktivert logging på alt som kjøres på hostene. Vi må endre på daemonsettet vi lagde ved å bruke kommandoen

```
kubectl create -f datadog-agent.yaml
```

Vi endrer et daemonset ved å bruke kommandoen

```
kubectl edit daemonset datadog-agent
```

Vi konfigurerer så .yaml-filen, og legger inn følgende i “env” seksjonen i filen.

```
- name: DD_LOGS_ENABLED
       value: "true"
   - name: DD_LOGS_CONFIG_CONTAINER_COLLECT_ALL
       value: "true"
```

I tillegg til dette trenger vi å mounte pointdir volumet i volumeMounts. Det gjør vi ved å legge inn følgende kode i den samme .yaml-fil.

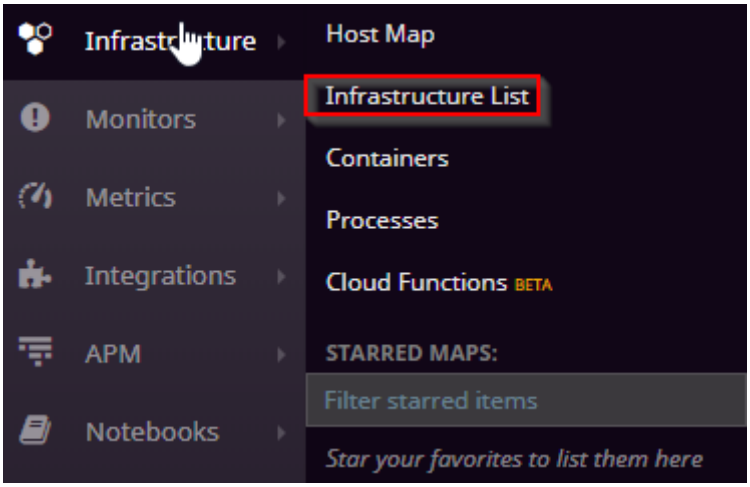
```
  volumeMounts:
     - name: pointerdir
         mountPath: /opt/datadog-agent/run
volumes:
  - hostPath:
       path: /opt/datadog-agent/run
       name: pointerdir
```

Vi har nå aktivert både logging og metrics data, så i neste delkapittel kan vi se om vi faktisk nå får denne dataen inn i datadog grensesnittet.

10.5 Streaming av data

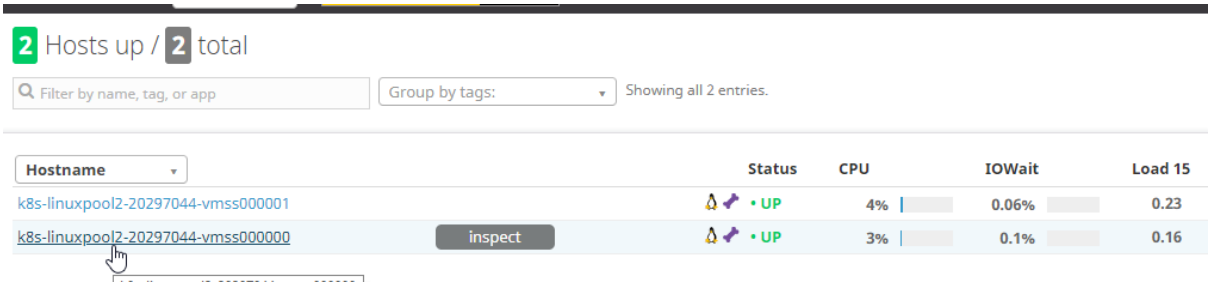
10.5.1 Metrics

Vi logger oss inn med Datadog kontoen vår og går på Infrastructure og velger først “Infrastructure List”.



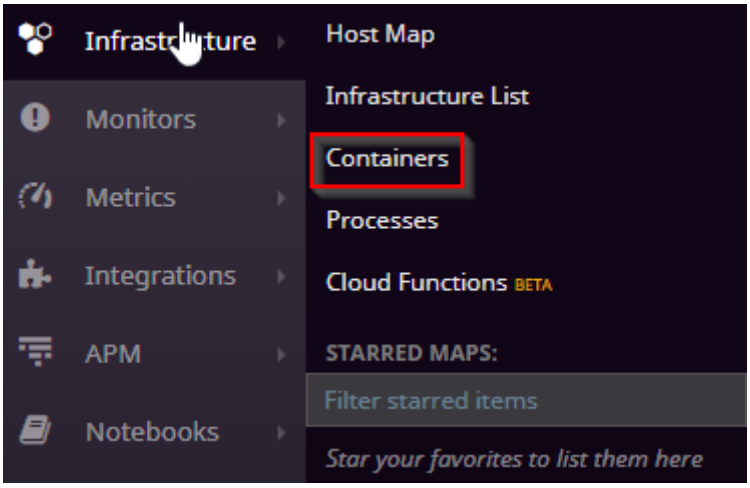
Figur 117: Datadog metrics data - Infrastructure List (1/2)

Vi ser at den har funnet de to linux hostene som vi vet at agenten kjører på. Vi kan se diverse data om hvilket OS som kjøres, hva det brukes av CPU etc.



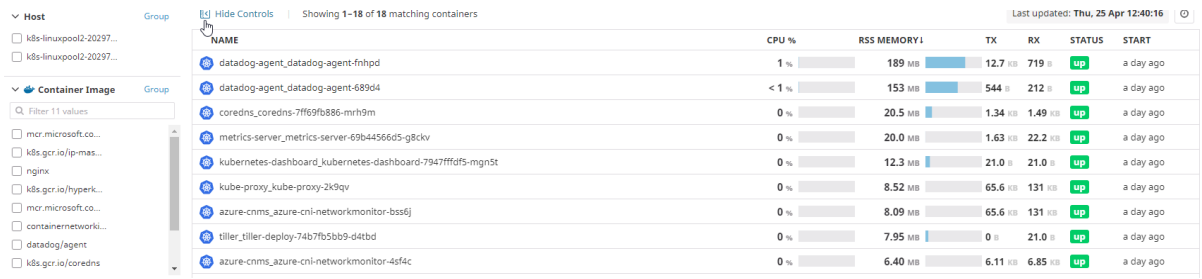
Figur 118: Datadog metrics data - Infrastructure List (2/2)

Det hentes ikke inn så mye data på infrastruktur nivå, vi er mer interessert i data fra containere som kjører på hosten. Vi går tilbake og velger “Containers” istedenfor “Infrastructure List”



Figur 119: Datadog metrics data - Containers (1/2)

Som vi ser fra bildet, får vi opp her alle containere som kjører i vårt Kubernetes cluster, med litt info om status. Man kan også trykke på hver enkelt Pod og få mer informasjon. Vi kan også her filtrere basert på Deployments, Pods og Services.



Figur 120: Datadog metrics data - Containers (2/2)

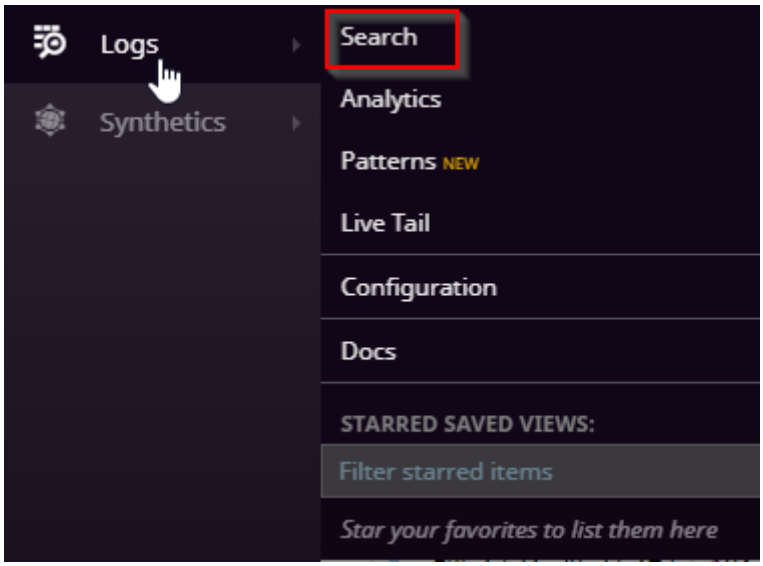
Som vi ser så får vi alt av metrics fra containere, vi kan nå gå og se om vi også får detaljerte logger fra de forskjellige containerne.

10.5.2 Loggdata

Vi går inn på “Logs” og velger “Search”. Her har vi egentlig mange valg

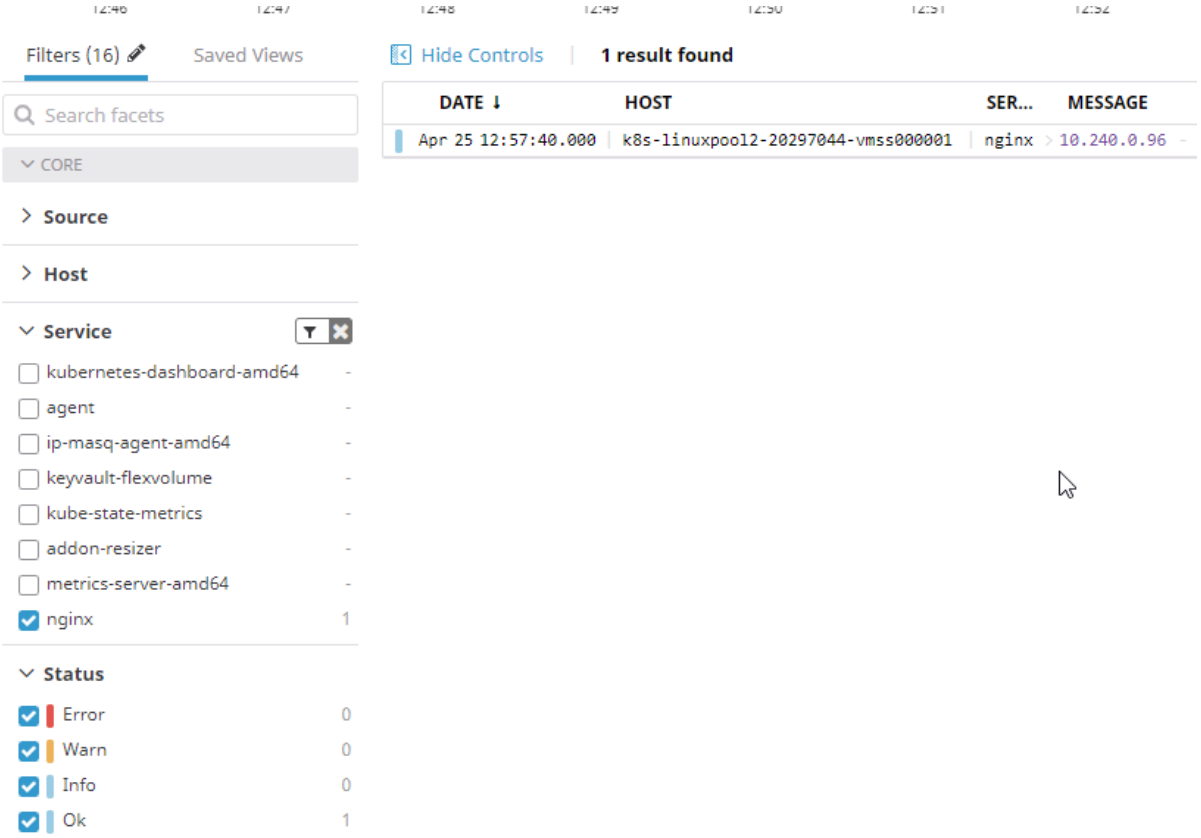
- Analytics: Ser på all logging innenfor et tidsrom
- Patterns: Ser på logger over i en periode og prøver å se på avvik fra normal logging.
- Live tail: Er all logging live, mens det kommer inn.

Siden vi skal bare se om vi får inn logg data kan vi bare velge search.



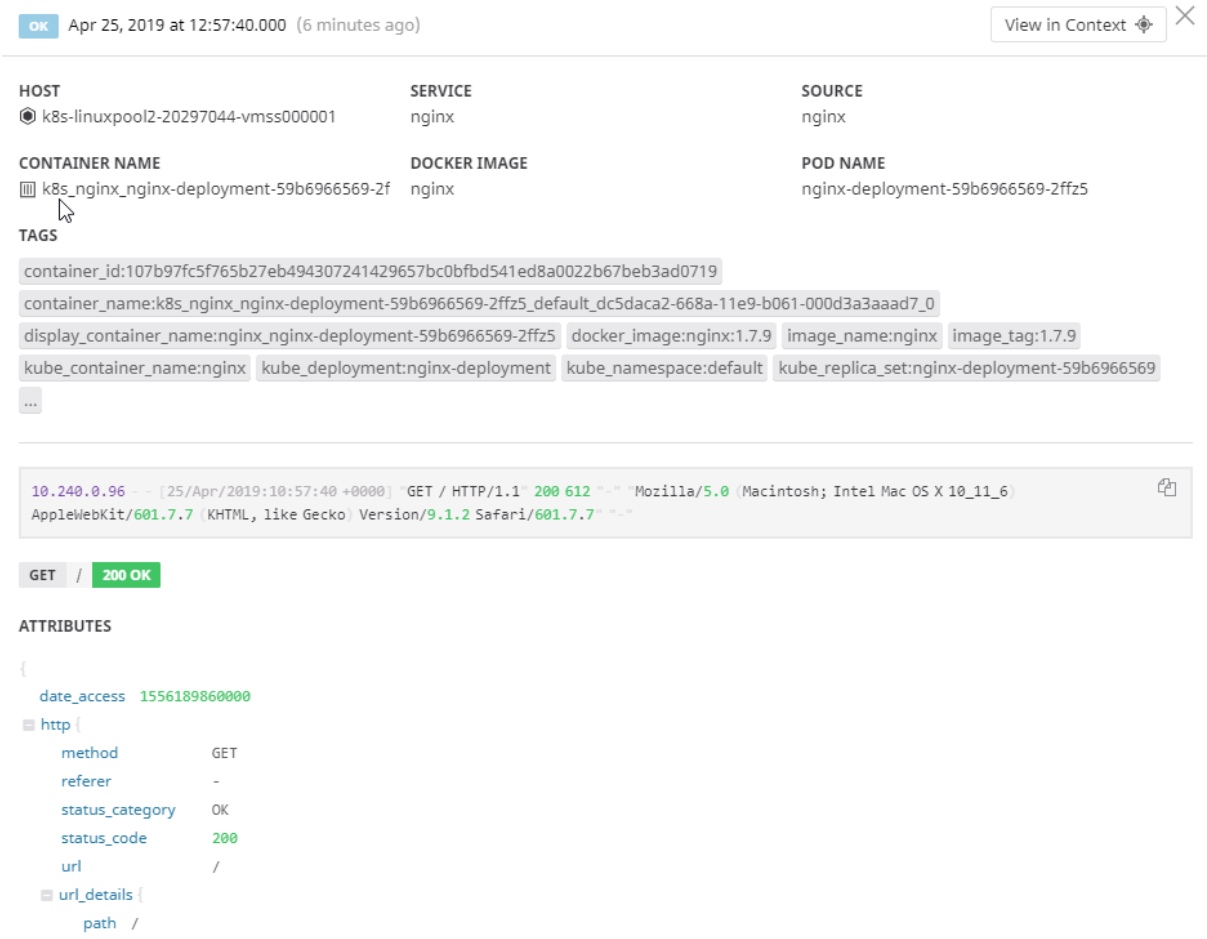
Figur 121: Datadog loggdata (1/6)

Vi velger her å se på nginx servicen vår og ser at det er et logg innslag fra nginx servicen vår som ligger ute på en public IP. Vi ser at det er et HTTP GET logg innslag.



Figur 122: Datadog loggdata (2/6)

Vi kan gå mer i detalj på hva logg innslaget er. Som vi ser fra bildet under er det ett HTTP log innslag med statuskode 200 OK.



Figur 123: Datadog loggdata (3/6)

Alt virker å fungere, og loggdata kommer inn, som en siste test, kan vi prøve å tvinge frem litt mange HTTP requests til nginx sida vår ved å bruke et program som heter Siege. Siege simulerer bruk av et et konfigurerbart antall simulerte nettlesere.

Det lastes ned på en maskin, og vi kjører følgende kommando fra CMD.

```
siege.exe -b -c50 -t20s http://52.166.241.235/
```

Med følgende kommando kjører vi 50 threads over 20 sekunder mot adressen på nginx siden vår.

Vi gjør klart “Live tail” vinduet vårt i Datadog og peker mot nginx servicen vår.

Figur 124: Datadog loggdata (4/6)

Vi starter så siege programmet, og lar det kjøre i 20 sekunder. Etter at det har kjørt

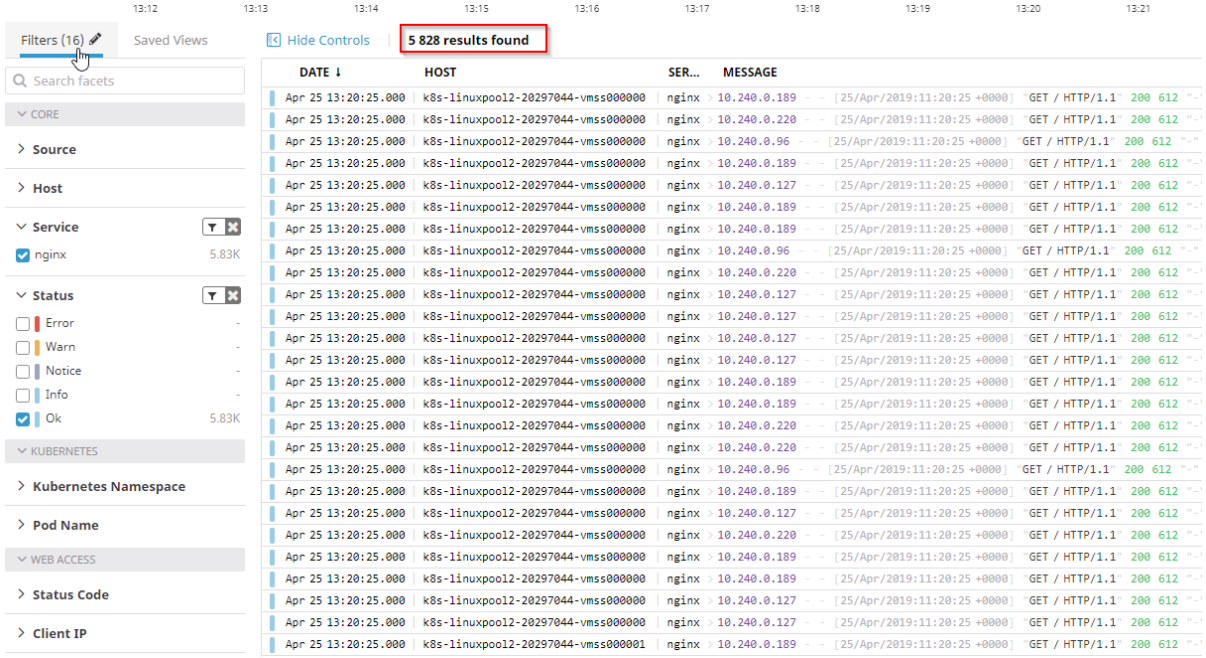
får vi følgende fra CMD vinduet vårt.

```
Transactions:                   5810 hits
Availability:                 100.00 %
Elapsed time:                  19.33 secs
Data transferred:               3.40 MB
Response time:                  0.08 secs
Transaction rate:             300.58 trans/sec
Throughput:                     0.18 MB/sec
Concurrency:                   23.76
Successful transactions:        5825
Failed transactions:               0
Longest transaction:            0.13
Shortest transaction:           0.07

FILE: siege.log
You can disable this annoying message by editing
the C:\siege-windows\etc\siegerc file; change
the directive 'show-logfile' to false.
```

Figur 125: Datadog loggdata (5/6)

Vi ser at siden har blitt requested 5825 ganger i løpet av 20 sekunder, som gir cirka 300 requests per sekund. Heller ingen feil.

Vi kan nå se på Datadog loggingen vår og se om vi har noen lignende.



Figur 126: Datadog loggdata (6/6)

Som vi ser har vi mange HTTP GET meldinger, med status kod 200 (Alt ok). Vi ser at vi har logget 5828 results, som er temmelig likt 5825 vi hadde fra Siege verktøyet. Alt i alt virker logging veldig bra.

10.6 Kube-state metrics

I tillegg til metrics vi allerede har, som for eksempel, cpu/minne etc. trenger vi få på plass metrics som går litt dypere.

Kubernetes har noe som heter Kube-state metrics vi ønsker å ta i bruk med Datadog. Kube-state metrics blir integrert med Kubernetes API og gjør tilgjengelig en rekke nye metrics om alle Kubernetes-objektene i clusteret vårt.

Tar vi bruk kube-state metrics så kan vi hente inn informasjon om følgende objekter i clusteret vårt:

- CronJob
- DaemonSet
- Deployment
- Job
- LimitRange
- Node
- PersistentVolume
- PersistentVolumeClaim
- Pod
- ReplicaSet
- ReplicationController
- ResourceQuota
- Service
- StatefulSet
- Namespace
- Horizontal Pod Autoscaler
- Endpoint
- Secret
- ConfigMap

10.6.1 Installasjon av Kube-state metrics

Vi starter med å git clone følgende repository ved å bruke kommandoen

```
git clone https://github.com/kubernetes/kubernetes.git
```

Vi går inn i mappestrukturen og finner mappen som heter kubernetes, her ligger følgende filer.

```
kube-state-metrics-cluster-role-binding.yaml
kube-state-metrics-cluster-role.yaml
kube-state-metrics-deployment.yaml
```

```
kube-state-metrics-role-binding.yaml
kube-state-metrics-role.yaml
kube-state-metrics-service-account.yaml
kube-state-metrics-service.yaml
```

Vi kjører kubectl create igjen for å opprette RBAC kontoer og samtidig en deployment. Istedenfor å kjøre hver enkelt fil, kan vi heller bruke kubectl create på mappen filene ligger i.

```
kubectl create -f kubernetes
```

Vi kjører kommandoen

```
kubectl get deployments -n kube-system
```

Og ser følgende

```
kube@k8s-master-20297044-0:~$ kubectl get deployments -n kube-system
NAME                   READY   UP-TO-DATE   AVAILABLE   AGE
cluster-autoscaler     1/1     1            1           46h
coredns                1/1     1            1           46h
kube-state-metrics     1/1     1            1           25h
kubernetes-dashboard   1/1     1            1           46h
metrics-server         1/1     1            1           46h
tiller-deploy          1/1     1            1           46h
kube@k8s-master-20297044-0:~$
```

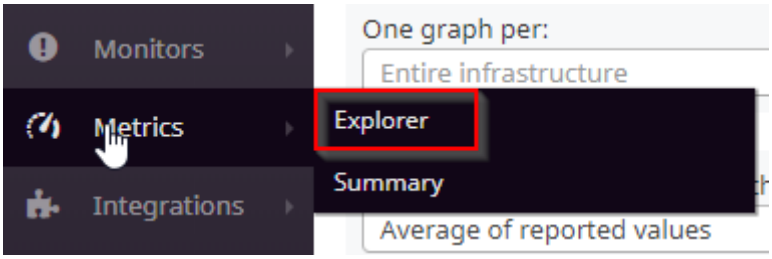
Figur 127: Installasjon Kube-state metrics

Vi ser at kube-state-metrics kjører, og vi må nå vente 5-10 minutter før kube-state-metrics er mulig å se i Datadog.

10.6.2 Kube-state-metrics data i Datadog

Alt av metrics bak syntaksen kubernetes_state.* er bare mulig å hente hvis man har kube-state-metrics deployment kjørende.

Vi tar en titt i Datadog og prøver å hente ut informasjon om nginx ved å bruke kube-state-metrics data.

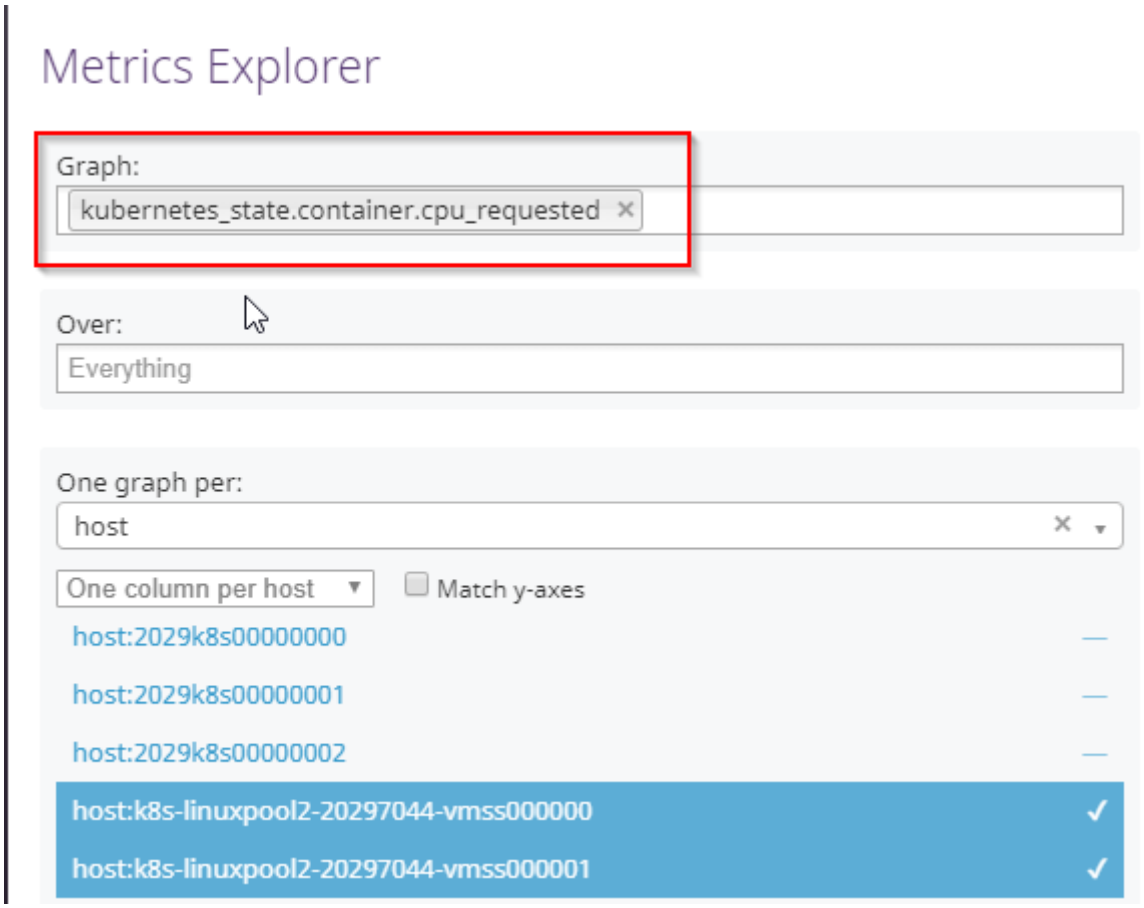


Figur 128: Kube-state metrics i Datadog (1/3)

Vi velger så å etterspørre følgende metrics

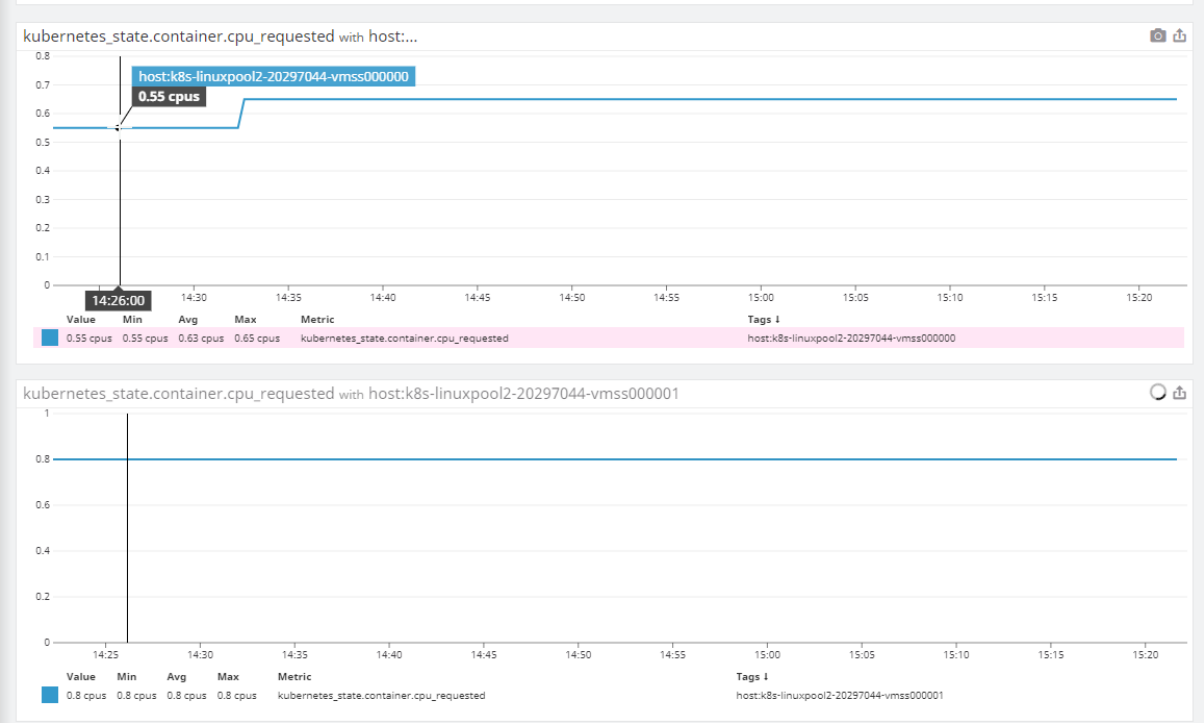
```
kubernetes_state.container.cpu_requested
```

Denne sier noe om hvor mye CPU bruk er allokert på våre to linux hoster. Altså hvor mye CPU det er reservert til sammen av alle podene på hver enkelt node. Som vi vet fra tidligere har vi 4 vCPU-er på hver node i Azure.



Figur 129: Kube-state metrics i Datadog (2/3)

Som vi ser fra bildet, kan vi se at den ene linux noden har 0.65 vCPU-er allokert, mens den andre noden har 0.8 vCPU-er allokert.



Figur 130: Kube-state metrics i Datadog (3/3)

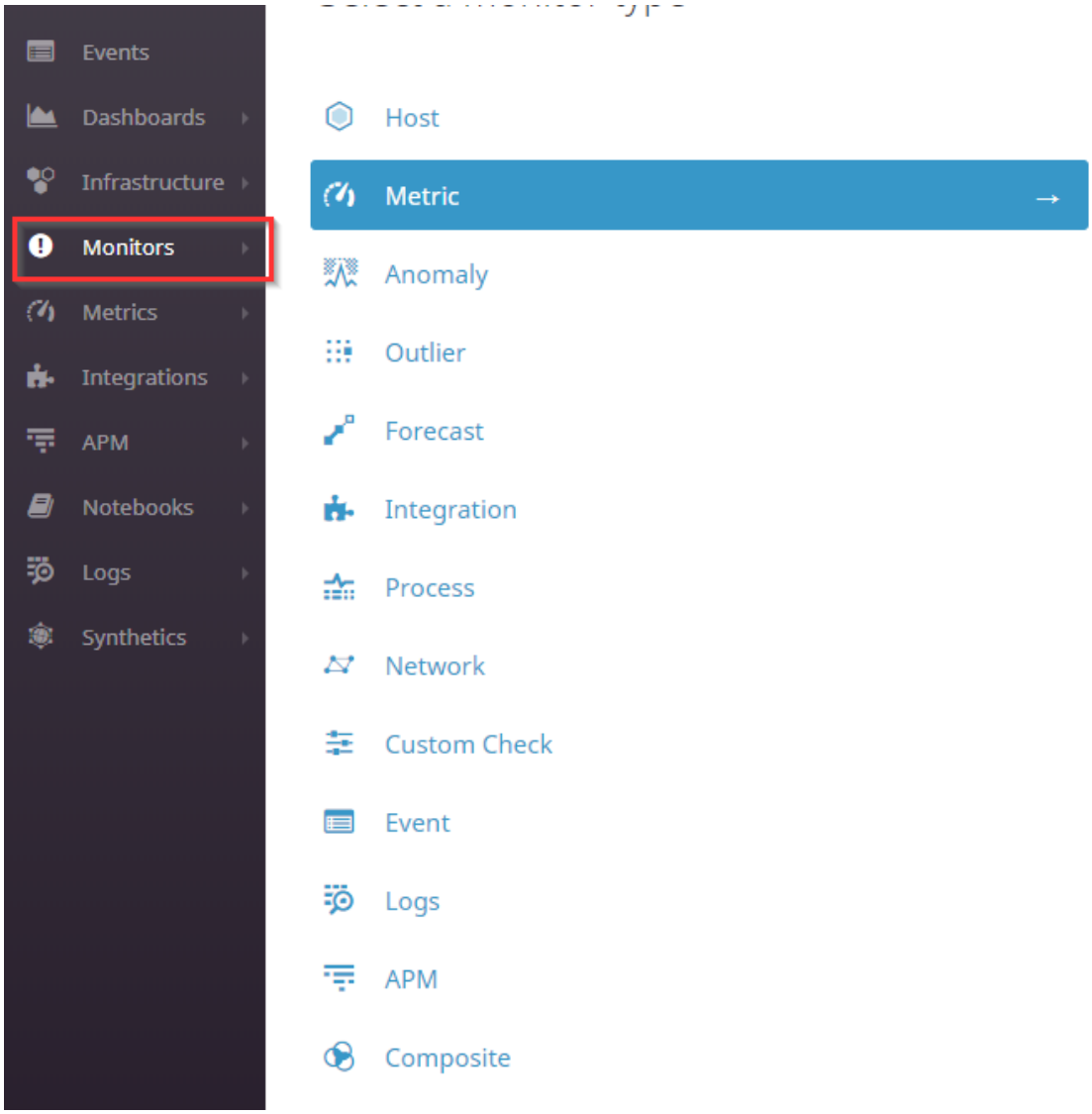
Vi kan med trygghet si at vi får kube-state-metrics fra Kubernetes clusteret vårt inn i Datadog.

10.7 Alarmhåndtering

Nå som vi får metrics og logger inn i Datadog kan vi starte med å lage noen alarmer. Vi starter med å lage en enkel metric alarm. Denne omhandler alarmering når CPU bruk er høy.

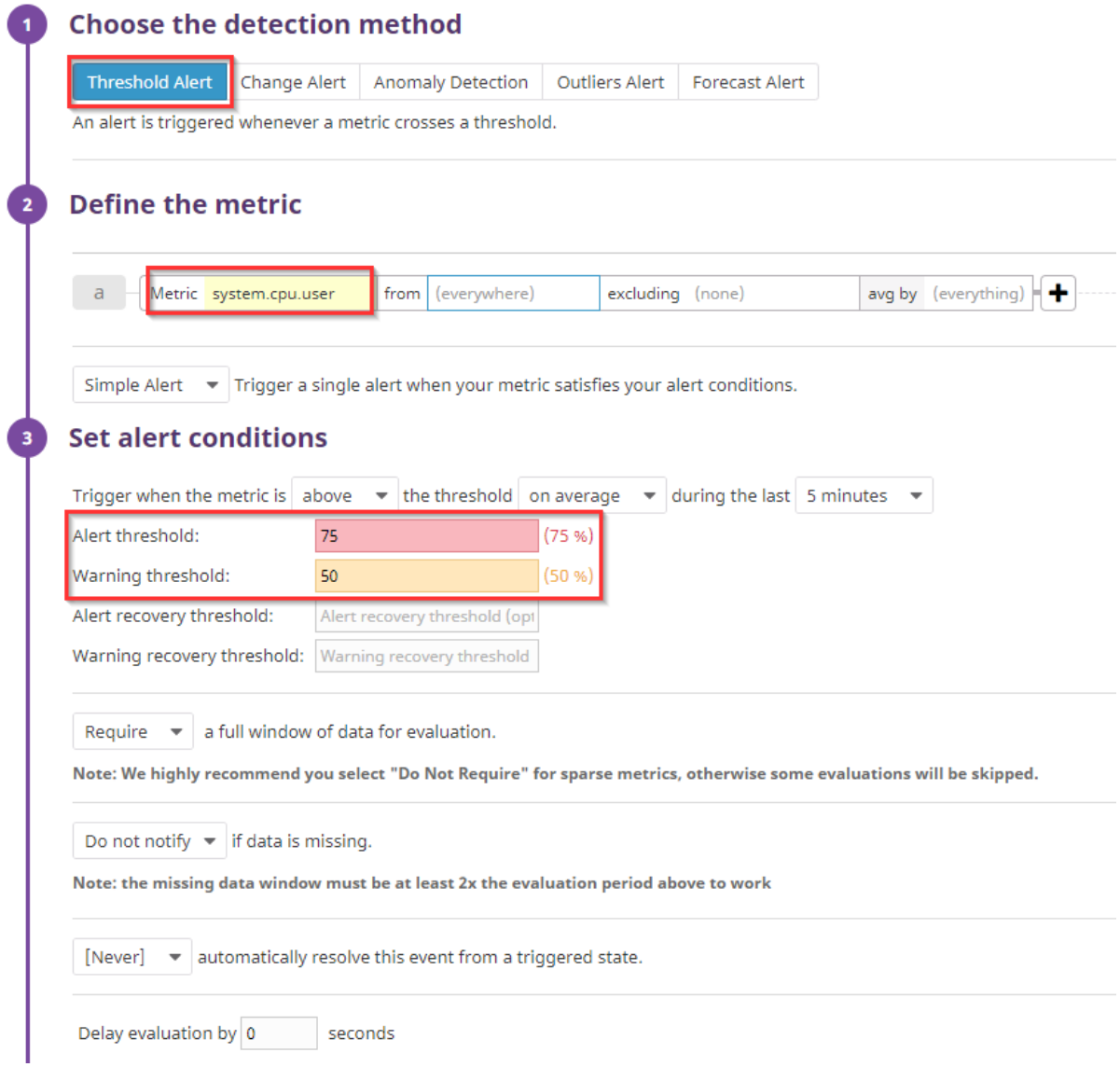
10.7.1 Konfigurasjon av en monitor

Vi velger Monitors og så Metric.



Figur 131: Konfigurasjon av Datadog metric alarm (1/5)

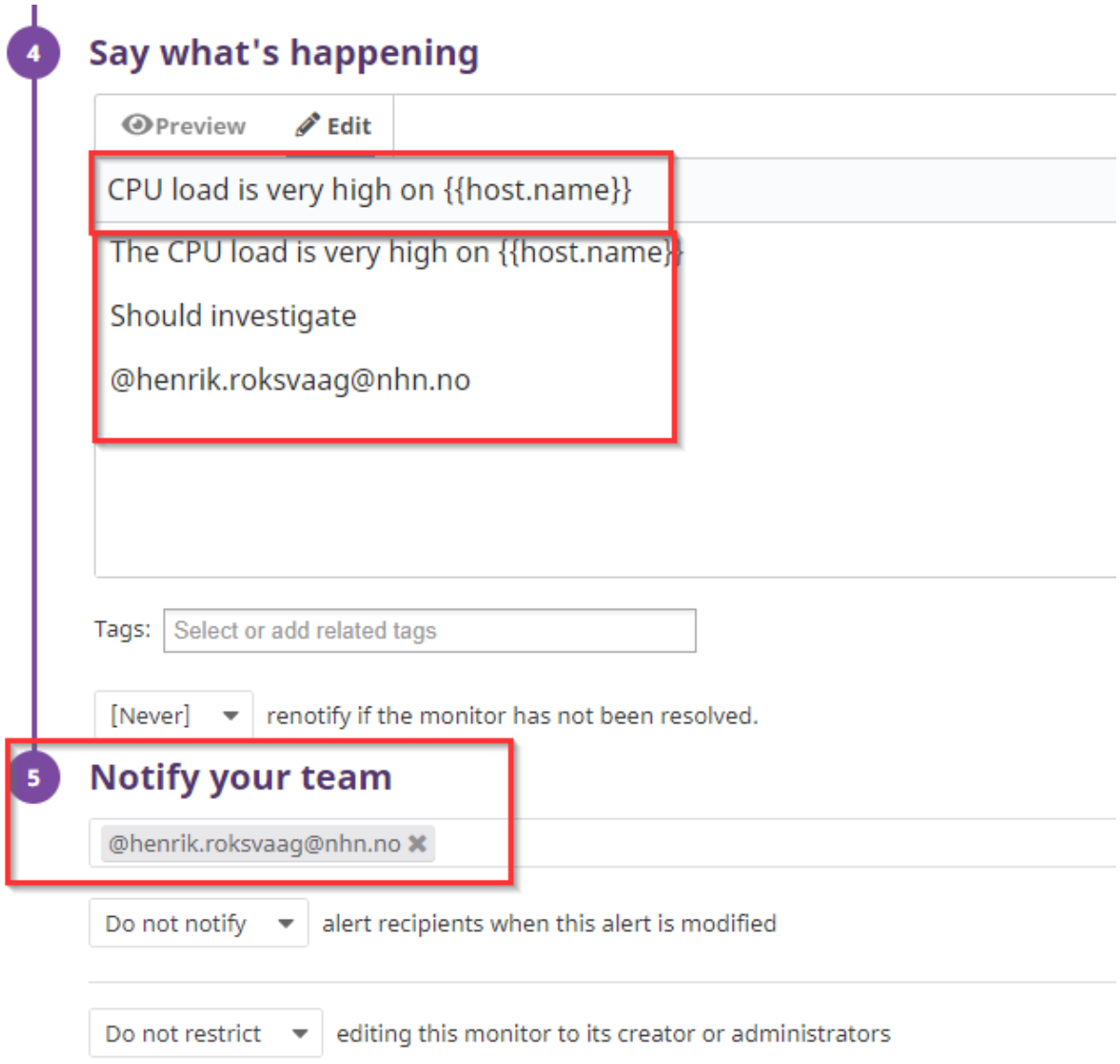
Her får vi flere valg. Vi velger “Threshold Alert” fordi vi ønsker at når bruk går over et visst nivå så skal det trigge en alarm. Bruken vi velger er her av typen system.cpu.user. Det vil si at den ser på CPU-bruken per host, og ikke spesifikt til en container. Vi velger 50% som warning og 75% som alert.



Figur 132: Konfigurasjon av Datadog metric alarm (2/5)

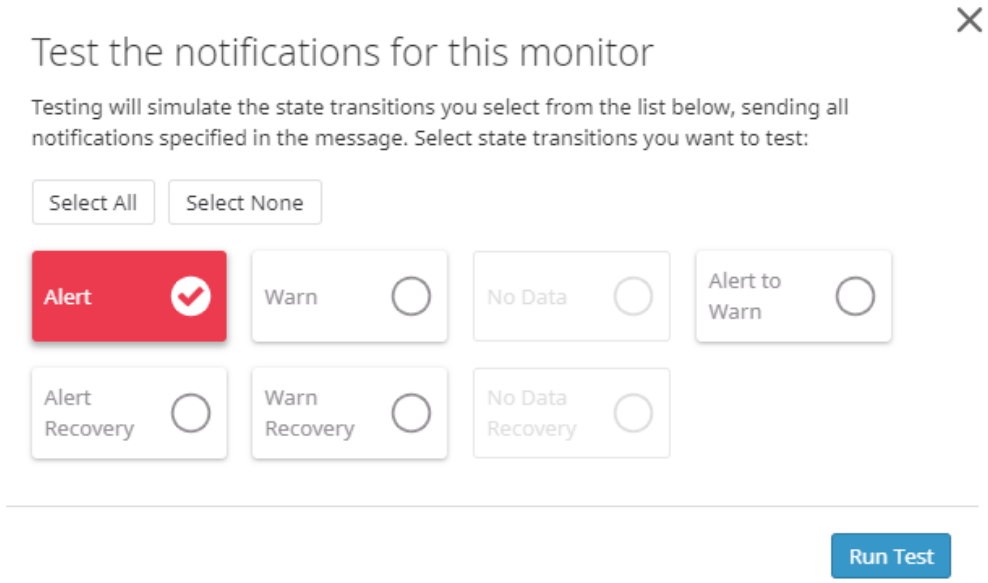
Videre kan vi velge hva det skal stå i tittelen på varselet. Her kan vi bruke en enkel parameter for å spesifisere hvilken host som trøbler ved å legge inn {{host.name}}. Vi legger inn så noe vilkårlig tekst for hva det skal stå i varselet. Her kan man akkurat som i tittelen spesifisere hvilken host det er snakk om. Her kan det være naturlig å linke til interne prosedyrer i bedriften når et slikt varsel inntreffer.

Til slutt kan vi spesifisere hvilke av våre teammedlem som skal varsles, i dette tilfellet er det bare Henrik Roksvaag som er et medlem og varselet vil derfor bare bli sendt til han. Her er det naturlig å varsle ansvarlig driftsteam skulle løsningen bli implementert hos NHN.



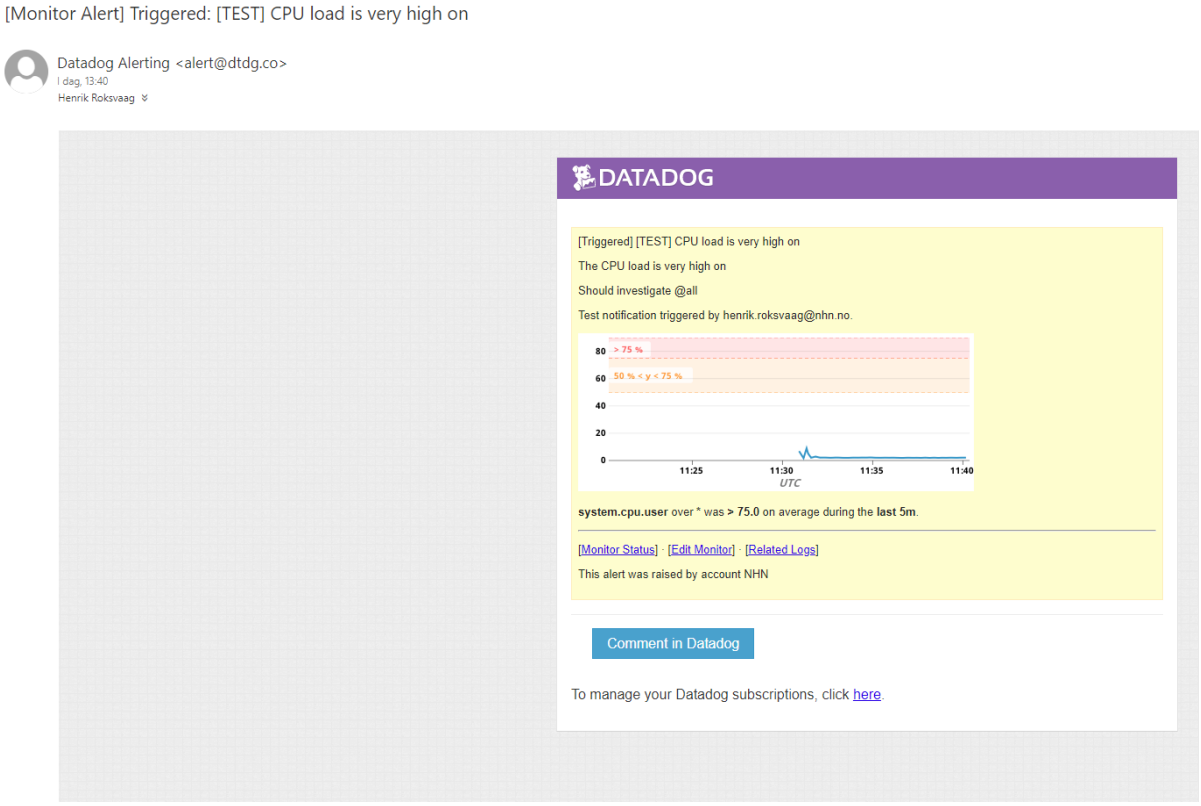
Figur 133: Konfigurasjon av Datadog metric alarm (3/5)

Datadog har innebygde verktøy for å teste hvordan denne varslingen fungere. Vi velger type “Alert” og kjører testen.



Figur 134: Konfigurasjon av Datadog metric alarm (4/5)

Henrik Roksvaag får e-post om hva feilen er og hva som har skjedd. Her vises også grafer.
I testen hentes ikke {{host.name}} ut fordi det bare en test, det er ikke en reell host som har problemer. Dette gjelder også grafen i e-posten, som holder seg langt under grensene satt i alarmen.

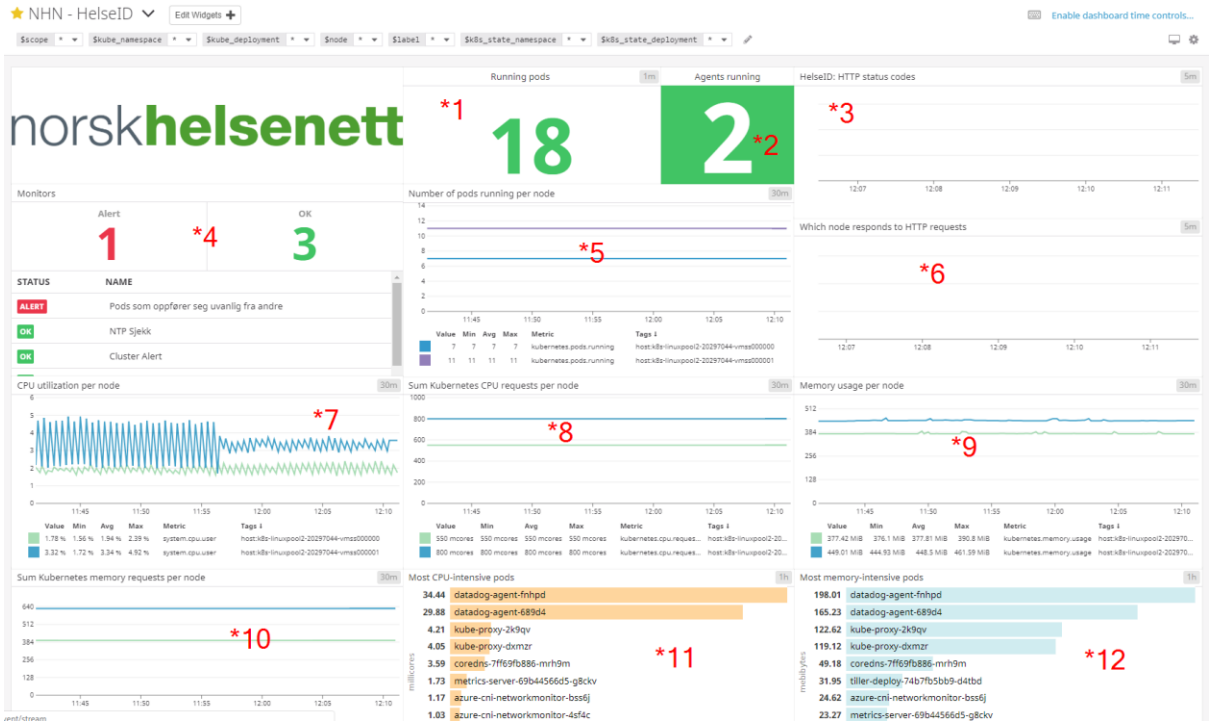


Figur 135: Konfigurasjon av Datadog metric alarm (5/5)

10.8 Design av HelseID dashboard

Nå som vi har både data strømmene inn til Datadog og vi har fått laget noen monitorer, kan vi begynne å designe et dashboard som gjør at vi får bedre innblikk i helsen til HelseID.

Det er bygget opp av forskjellige widgets som vi kan justere størrelsen på og sette slik vi ønsker. Vårt endelig design ble slik.



Figur 136: Datadog dashboard for HelseID

*1: Dette (tallet 18) viser antall pods vi kjører i vårt Kubernetes cluster. Det inkluderer alle namespace vi har i Kubernetes clusteret vårt.

*2: Viser hvor mange agenter vi har i vårt system. Hvis det er behov for flere hoster så vil ny host automatisk få en ny agent. Det skal alltid være to agenter kjørende. Og denne boksen vil bli rød hvis tallet går under to.

*3: Her vises alle HTTP status koder som blir returnert når noen besøker HelseID frontend. De vil her bli sortert på OK, warnings og errors.

*4: Her vises alle monitorene vi har, og om de har status OK, WARN eller ALERT.

*5: Her viser vi hvor mange pods som kjører per host. Denne bør være nokså lik, da Pod-ene i utgangspunktet skal bli jevnt fordelt.
*6: Her ser vi hvilken node som svarer på HTTP requests. Vi har minimum to noder, så da skal disse nodene svare ca like mye på innkomne forespørsler.

*7: Her viser vi CPU bruk per host.

*8: Her viser vi CPU requests per node. Det er også hvor mye som er reservert på en CPU, dette vises i mcores. der 1000 mcore er 1 vCPU. Vi har tidligere fortalt at våre noder har 4 vCPU-er hver. Hvis det har blitt reservert 3900mcore, og en ny pod som ønsker å reservere 400mcore blir forsøkt kjørt opp, må en ny node kjøres opp.

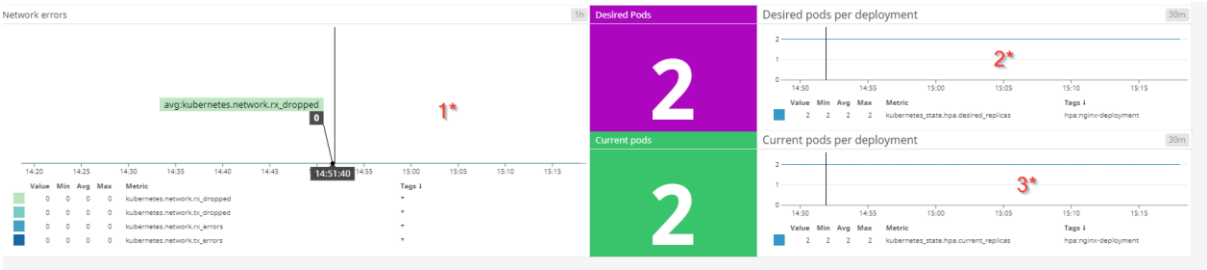
*9: Her vises minnebruk per host.

*10: Her vises memory requests per host, oppgitt i MiB. Samme som CPU requests, men her vises det hvor mye minne som er reservert på en host.

*11: Her vises pods som bruker mest CPU, sortert i en topp 10 liste.

*12: Her vises pods som bruker mest minne, sortert i en topp 10 liste.

Til slutt har vi et ekstra dashboard, som vi skal forklare litt nærmere.



Figur 137: Ekstra Datadog dashboard for HelseID

* 1: Her vises alle nettverk-errors internt i Kubernetes clusteret vårt. Sortert etter TX og RX errors og TX og RX dropped. Det skal ikke være mange forekomster av dette.

* 2: Her vises hvor mange pods vi ønsker skal kjøre i en bestemt deployment. Her kan vi filtrere og legge inn flere deployments for å få flere grafer.

* 3: Her vises nåværende pods i den deploymenten. Så her kan vi oppdage forskjell i det vi ønsker og realiteten. Den lilla boksen skal være lik den grønne. Og hvis ønsket antall pods skulle vært fem, men nåværende pods er tre, så vet vi at det er to pods som ikke kjører. Dette kan være greit å bruke til feilsøking.

11. Sikkerhet i Kubernetes

I januar i år ble Kubernetes økosystemet for første gang rystet etter at den første store sikkerhetsfeilen ble oppdaget i Kubernetes. Sikkerhetsproblemet - CVE-2018-1002105 - gjorde det mulig for angripere å kompromittere clustere via Kubernetes API-serveren, slik at de kunne kjøre kode for å utføre skadelig aktivitet som installering av skadelig programvare etc.

Tidligere i år led Tesla av et komplekst cryptocurrency malware angrep forårsaket av en feilkonfigurasjon i Kubernetes konsollet. Angrepet utnyttet det faktum at dette bestemte Kubernetes konsollet ikke var passordbeskyttet, slik at de kunne få tilgang til en av pod-ene som inneholdt påloggingsdata for Tesla større AWS-miljø.

11.1 Hvordan sikre seg best mulig mot angrep

Etter hvert som bedrifter tar i bruk containere mer og mer, er det viktig å ta nødvendige skritt for å beskytte deres infrastruktur. Vi har kommet frem til noen punkter vi har implementert i vårt miljø, som vi mener er helt nødvendig for å beskytte infrastrukturen mot angrep.

11.1.1 Hold Kubernetes oppdatert

Kubernetes kommer med oppdateringer hvert kvartal. Disse inneholder sikkerhetspatcher, samt bugfixes. Vi anbefaler at man alltid kjører den siste versjonen av Kubernetes stable. Oppdatering kan også bli vanskeligere hvis man hopper over noen oppdateringer, så det anbefales å oppdatere Kubernetes kvartalsvis.

11.1.2 Ta i bruk RBAC (Role-Based Access Control)

RBAC har siden Kubernetes versjon 1.6 vært aktivert som standard. RBAC kontrollerer hvem som kan ta i bruk Kubernetes APIet, og hvilke rettigheter de vil ha. Det er ikke anbefalt å ta i bruk Cluster-wide tilgangsrettigheter, da er det bedre å ta i bruk namespace rettigheter. Hvis jeg som utvikler eller drifter av HelseID skal kun ha tilgang til namespace HelseID, og ikke andre namespace. Det gjør at man ikke får tilgang til ting man ikke skal ha tilgang til. Vi kan se hvem som har tilgang til hva ved å bruke kommandoen `kubectl get clusterrolebinding` eller kommandoen `kubectl get rolebinding --all-namespaces`. Vi kan for eksempel se hvem som har fått tilgangen “cluster-admin” i vårt miljøet ved å kjøre kommandoen `kubectl describe clusterrolebinding cluster-admin` . Vi ser fra bildet under at den tilhører bare “masters” gruppen.

```
kube@k8s-master-20297044-0:~$ kubectl describe clusterrolebinding cluster-admin
Name:         cluster-admin
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
Role:
  Kind:  ClusterRole
  Name:  cluster-admin
Subjects:
  Kind   Name            Namespace
  ----   ----            ---------
  Group  system:masters
kube@k8s-master-20297044-0:~$
```

Figur 138: Cluster-admin informasjon

Hvis man har applikasjon som trenger tilgang til Kubernetes APIet, så er det viktig man oppretter egne service accounts per applikasjon, som har så lite tilganger som trengs for at applikasjonen skal fungere.

11.1.3 Ta i bruk Namespaces for å sette sikkerhetsgrenser

Det er viktig å ta i bruk Namespaces for å sette sikkerhetsgrenser slik at det oppnås bedre isolasjon mellom de forskjellige applikasjonene og komponentene. Det er også enklere å aktivere sikkerhetspolicyer på namespaces. Som nevnt videre kan vi sette policies at enkelte brukere skal bare ha tilgang på bestemte Namespaces etc.

Vi kan se hvilke Namespaces vi har i clusteret vårt ved å kjøre kommandoen

```
kubectl get ns
```

```
kube@k8s-master-20297044-0:~$ kubectl get ns
NAME          STATUS   AGE
default       Active   6d
helseid       Active   4s
kube-public   Active   6d
kube-system   Active   6d
monitoring    Active   10s
kube@k8s-master-20297044-0:~$
```

Figur 139: Namespace i clusteret

Som vi ser fra bildet så har vi opprettet egne namespaces for monitoring og HelseID.

11.1.4 Separer sensitive jobber

For å begrense den potensielle effekten av et angrep så er det best å kjøre sensitive jobber på et sett dedikerte maskiner. Denne fremgangsmåten reduserer risikoen av at en noen får tilgang til en sensitiv applikasjon gjennom en mindre sikker applikasjon.

11.1.5 Øk node sikkerheten

Det er flere ting man kan gjøre for å øke node sikkerheten, og de er:

- Kontroller at host er sikker og konfigurert riktig. En enkel måte å gjøre dette på er kontrollere host mot en CIS benchmark (CIS, 2019).
- Kontroller nettverkstilgang til sensitive porter. Det er viktig å kontrollere at nettverket blokkerer porter som brukes (10250 og 10255).
- Begrens tilgang til Kubernetes API-server bortsett fra på godkjente nettverk.
- Minimer administrativ tilgang til Kubernetes nodene. Tilgang til nodene i clusteret bør være begrenset, feilsøking og andre oppgaver kan vanligvis håndteres uten direkte tilgang til noden.

11.1.6 Skru på Audit logging

Det er en fordel at man er sikker på at man har skrudd på Audit logging. Hvis Audit logging er aktivert, så overvåkes det for uregelmessige eller uønskede API-kall, spesielt av typen godkjenningsfeil. Disse vil ha status “Forbidden” i loggene. Slike innslag i loggene kan bety at en angriper prøver å trenge seg inn i systemet. Det kan være viktig å alarmere på slike logg innslag for å avdekke et mulig angrep tidlig.

En vellykket log event vil se slik ut

```
{
    "kind": "Event",
    "apiVersion": "audit.k8s.io/v1beta1",
    "metadata": {
        "creationTimestamp": "2019-94-19T08:26:55Z"
    },
    "level": "Request",
    "timestamp": "2019-04-19T08:26:55Z",
    "auditID": "288ace59-97ba-4121-b06e-f648f72c3122",
    "stage": "ResponseComplete",
    "requestURI": "/api/v1/pods?limit=500",
    "verb": "list",
    "user": {
        "username": "admin",
        "groups": ["system:authenticated"]
    },
    "sourceIPs": ["10.0.138.91"],
    "objectRef": {
        "resource": "pods",
        "apiVersion": "v1"
    },
```

```
    "responseStatus": {
        "metadata": {},
        "code": 200
    },
    "requestReceivedTimestamp": "2019-04-19T08:26:55.466934Z",
    "stageTimestamp": "2019-04-19T08:26:55.471137Z",
    "annotations": {
        "authorization.k8s.io/decision": "allow",
        "authorization.k8s.io/reason": "RBAC: allowed by ClusterRoleBinding
"admin-cluster-binding" of ClusterRole "cluster-admin" to User "admin""
    }
}
```

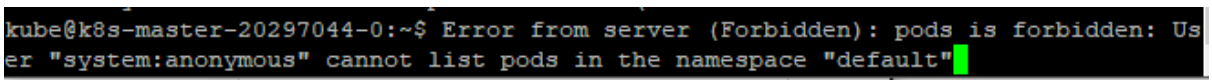
Dette eksempelet under vil logge alle requests gjort av admin brukeren, eller hvilken som helst request gjort av en anonym system bruker.

```
apiVersion: audit.k8s.io/v1beta1
kind: Policy
omitStages:
  - "RequestReceived"
rules:
  - level: Request
    users: ["admin"]
    resources:
      - group: ""
        resources: ["*"]
  - level: Request
    user: ["system:anonymous"]
    resources:
      - group: ""
        resources: ["*"]
```

La oss si vi oppretter en ny bruker som ikke er assosiert til noen ClusterRole, og denne brukeren prøver å kjøre kommandoen `kubectl get pods`.

Vil da få følgende output

```
Error from server (Forbidden): pods is forbidden: User "system:anonymous" cannot
list pods in the namespace "default"
```



Figur 140: Resultat av audit logging

Vi kan se fra Audit loggen at det er også registrert

```
{
```

```
    "kind": "Event",
    "apiVersion": "audit.k8s.io/v1beta1",
    "metadata": {
        "creationTimestamp": "2019-04-20T10:00:20Z"
    },
    "level": "Request",
    "timestamp": "2019-04-20T10:00:20Z",
    "auditID": "5fc5eab3-82f5-480f-93d2-79bfb47789f1",
    "stage": "ResponseComplete",
    "requestURI": "/api/v1/namespaces/default/pods?limit=500",
    "verb": "list",
    "user": {
        "username": "system:anonymous",
        "groups": ["system:unauthenticated"]
    },
    "sourceIPs": ["10.0.141.137"],
    "objectRef": {
        "resource": "pods",
        "namespace": "default",
        "apiVersion": "v1"
    },
    "responseStatus": {
        "metadata": {},
        "status": "Failure",
        "reason": "Forbidden",
        "code": 403
    },
    "requestReceivedTimestamp": "2019-04-20T10:00:20.605009Z",
    "stageTimestamp": "2019-04-20T10:00:20.605191Z",
    "annotations": {
        "authorization.k8s.io/decision": "forbid",
        "authorization.k8s.io/reason": ""
    }
}
```


11.2 Vår oppgave og “Norm for informasjonssikkerhet”

Vi skrev innledningsvis i forstudierapporten at vi ønsket å følge direktoratet for e-helse sin “Norm for informasjonssikkerhet”. Ser vi tilbake på valgene som har blitt gjort i oppgaven, vil vi gjerne trekke frem noen punkter vi mener er opprettholdt. Mange av punktene er ikke gjeldende for oss.

- *Personer i virksomheten skal gis tilgang i henhold til fastsatte prinsipper for tilgangsstyring i henhold til krav i kap. 5.2.*
    - Vi har satt krav til hvilke brukere skal ha tilgang til hvilke applikasjoner. De som driver med monitorering har ikke tilgang til HelseID osv.

- *Risikovurderinger har betydning både i det styrende, det gjennomførende og det kontrollerende informasjonssikkerhetsarbeidet.*
    - Risikovurderingsarbeid har stått sentralt i vårt arbeid med å se på potensielle farer og uønskede hendelser i vår forberedelse.

- *Loggene skal enkelt kunne analyseres ved hjelp av analyseverktøy med henblikk på å oppdage brudd.*
    - Vi har implementert monitoreringsverktøy som gjør det mulig for oss å oppdage uønskede hendelser. Det eneste vi vil her nevne som en bisetning er at ved bruk av Datadog så går trafikkflyten i skyen og lagres der. Ved personsensitiv data er dette uønsket, derfor anbefales det å se på verktøy som kan gjøre dette lokalt On Premise.

12. Kilder

CIS (2019): *Securing Kubernetes*
Tilgjengelig fra: https://www.cisecurity.org/benchmark/kubernetes/
(Hentet 15.04.2019)

Datadog (2019): *What is the Agent?*
Tilgjengelig fra: https://docs.datadoghq.com/agent/?tab=agentv6
(Hentet 10.04.2019)

Dønnem og Roksvaag (2019a): *Designrapport*

Dønnem og Roksvaag (2019b): *Forstudierapport*

Dønnem og Roksvaag (2019c): *Sluttrapport*

Bruk av containere i NHN

Sluttrapport

Av: Henrik Roksvaag og Ole Valla Dønnem

NTNU
Kunnskap for en bedre verden

Revisjonshistorie

Dato	Versjon	Beskrivelse	Forfattere
06.05.19	1.0	Første utkast	Henrik Roksvaag & Ole Valla Dønnem
13.05.19	1.1	Implementert endringer basert på tilbakemelding fra veiledere	Henrik Roksvaag & Ole Valla Dønnem
19.05.19	1.2	Endelig versjon Ferdigstilling gjennomført	Henrik Roksvaag & Ole Valla Dønnem

Innholdsfortegnelse - Sluttrapport

1. Introduksjon

1.1 Oppgavebeskrivelse

Oppgaven har gått ut på å se på mulighetene ved å ta i bruk containere og da spesielt verktøyet Kubernetes for å drifte en av NHNs applikasjoner. Det har blitt sett på skalering, monitorering og sikkerhet som viktige aspekter. En sammenligning med en tradisjonell driftsmetode har også blitt gjennomført, der vi har sett på funksjonalitet og bruk som viktige faktorer.

1.2 Hensikten med dokumentet

Hensikten med sluttrapporten er å gi en helhetlig vurdering av hvordan prosjektet har blitt gjennomført. Vi skal se på de målene vi satt oss innledningsvis i forstudierapporten, og om vi har klart å oppnå det vi ønsket. Prosjektgruppen gir også en anbefaling hvis Norsk Helsenett SF skulle ønske å ta i bruk Kubernetes.

1.3 Oversikt over dokumentet

Kapittel 2 i dette dokumentet beskriver verktøy, metoder og standarder som har blitt brukt i oppgaven. Samt hvordan arbeidsfordelingen har vært mellom medlemmene.

Kapittel 3 tar for seg effektmålene, resultatmålene og prosessmålene vi satte innledningsvis i oppgaven, og om disse har blitt oppnådd. Vi diskuterer videre om framtidsplanen har blitt overholdt, samtidig som vi diskuterer hva som gikk bra, mindre bra, dårlig og hva vi kunne gjort annerledes.

Kapittel 4 inneholder en konklusjon basert på det vi har funnet i oppgaven.

Kapittel 5 inneholder anbefalinger fra prosjektgruppen til NHN hvis de skulle velge å implementere Kubernetes i deres driftsmiljø i fremtiden.

1.4 Prosjektets øvrige dokumenter

Prosjektet har følgende dokumenter:

- Forstudierapport
- Designrapport
- Driftsrapport
- Sluttrapport
- Prosjekthåndbok
- Presentasjonsmateriell
- Individuelt refleksjonsnotat

2. Fremgangsmåte for løsning av oppgave

2.1 Metoder og standarder

Prosjektgruppen har forholdt seg til maler og standarder gitt i kapittel “7. Retningslinjer og standarder” i Forstudierapporten (Dønnem og Roksvaag, 2019c).

2.2 Kildebruk

Prosjektgruppen har benyttet internett som primærkilde i arbeidet, samt ansatte hos NHN. Vi har i all hovedsak brukt offisiell dokumentasjon fra Azure (Microsoft Docs), Datadog (Datadoghq), Kubernetes (Kubernetes Docs) og Docker (Docker Docs). Vi har også hentet informasjon fra noen websider.

2.3 Standardprogramvare

Vi har tatt i bruk verktøyene Office 365, SharePoint, Chrome, Putty, Microsoft Project 2016 og Powershell.

2.4 Arbeidsfordeling

Arbeidet mellom prosjektdeltakerne ble likt fordelt og begge deltakerne deltok i prosjektets arbeidsoppgaver, dette skjedde dynamisk gjennom hele prosjektet.

Alle avgjørelser ble diskutert før det ble fattet en beslutning. Styringsgruppen ble rådført ved behov.

3. Gjennomføring av oppgaven

3.1 Måloppnåelse

I forstudierapporten ble det definert noen mål som prosjektet skulle oppnå. Vi definerte klare resultat- og effektmål. I dette kapittelet vil vi ta for oss hvert enkelt mål og se om vi har klart å oppnå dem.

3.1.1 Effektmål

- *Kartlegging av bruk av Kubernetes og Docker til å supplere drift av applikasjoner hos NHN, da spesielt på Drift 2 og applikasjonen HelseID.*
  Prosjektgruppen har lært svært mye om Docker og Kubernetes. Vi har undersøkt hvordan kommunikasjonen henger sammen på de forskjellige lagene i verktøyene vi har brukt, og sitter igjen med mye kunnskap. Vi har sett spesielt på et punkt der nåværende driftsmodell er mangelfull; skalering.

  Vi har også konvertert en eksisterende NHN-applikasjon til å kjøre i vårt Kubernetes-cluster, ved hjelp av Docker.

- *Kartlegging av å gå over til en mer proaktiv driftsmodell i forhold til dagens driftsmodell som er mer reaktiv.*
  Vi har sett på fordelene og ulempene ved å ta i bruk en driftsmodell basert på Docker og Kubernetes. Vi har kartlagt at konvertering til en proaktiv driftsmodell med Kubernetes gir langt flere fordeler enn ulemper. Den største ulempen kan være at Kubernetes et komplekst verktøy, samt mindre egnet for enkelte typer applikasjoner. Vi har kartlagt at et skifte til Kubernetes kan føre til lavere TCO, mindre nedetid og dynamisk skalering av last for mange av applikasjonene som driftes av NHN.

- *Kartlegge en kost/nytte analyse av dagens system opp mot et nytt system.*
  Vi hadde under forstudierapporten skrevet en kost/nytte analyse der vi vurderte hva NHN ville spare ved å gå over til container-basert drift for HelseID. Prosjektgruppen syntes analysen ble overfladisk og at kvaliteten ikke sto i stil til resten av oppgaven. Den ble senere fjernet, og vi tok opp dette i et veiledningsmøte. Grunnen til at den ble fjernet er at det er for mange parametere inn i bildet når man totalt skifter driftsplattform. Konklusjonen ble derfor at den blir for omfattende hvis vi skal gjøre den ordentlig eller for overfladisk hvis vi gjør den enkel. Vi valgte derfor å kutte den ut.

- *Vurdering av sikkerhet tilknyttet tilgangsrettigheter hvis flere applikasjoner er*

hostet på samme node.
Prosjektgruppen har listet opp flere viktige punkter for hvordan man bør implementere viktige sikkerhetsrutiner i Driftsrapporten (Dønnem og Roksvaag, 2019). Vi gir her også noen eksempler for hvordan brukere kan gies spesifikke rettigheter ved bruk av RBAC. Vi utforsker videre mulighetene til å gi bestemte brukere tilgang til bestemte applikasjoner gjennom bruk av Namespaces, som gjør at brukerne ikke får tilgang til applikasjoner de ikke normalt skal ha tilgang til.

3.1.2 Resultatmål

- *Ta i bruk Kubernetes og Docker på Windows for å drifte en av NHN sine tjenester.*
  Vi har i løpet av oppgaven tatt i bruk Docker og Kubernetes både på Windows og Linux. Vi fant forholdsvis tidlig ut at containerdrift på Windows er umodent og har mange begrensninger. Verktøyene som kan brukes med Windows har ofte begrensninger som gjør at de ikke blir fullverdige alternativer til Linux. Bachelorgruppen prøvde mange løsninger, før Windows ble lagt på hylla og Linux valgt. Essensen i målet var å drifte en av NHNs Windows applikasjoner med å ta i bruk Docker og Kubernetes, så vi fikk aksept i styringsgruppen for å ta i bruk Linux for å oppnå dette når vi oppdaget at vi kom til å få problemer med å få det til på Windows. Vil si dette målet ble delvis oppnådd.

- *Monitorere tjenesten ved bruk av SCOM og/eller Splunk*
  SCOM og Splunk var to av verktøyene vi så på for å monitorere vårt miljø. Vi trodde i januar at dette skulle være gode verktøy å ta en titt på. Vi fant dog tidlig ut at SCOM ikke var egnet til å monitorere et container-basert miljø. En implementasjon av Splunk ble ikke gjennomført grunnet at det ville ta for mye tid. Monitorering av miljøet er noe vi ikke så på som like viktig som konverteringen og selve driftsaspektet. Vi valgte derfor Splunk vekk. Vi valgte å gå for Datadog, som er veldig likt Splunk, og er en skytjeneste. Som vi nevnte i designrapporten, så anbefales det ikke å ha loggverktøy som prosesserer data i skyen. Dette må gjøres på lokal hardware, og da er Splunk et bedre verktøy. Med Datadog fikk vi full monitorering og alarmering på vårt cluster. Så målet er delvis oppnådd, verktøyet er bare endret.

- *Ta i bruk automatisk skalering basert på last og antall forespørsler mot applikasjonen.*
  Som vi har tatt for oss i driftsrapporten har vi testet og implementert skalering for deler av HelseID. Dette er grundig forklart i driftsrapporten. For å summere, så fungerer både manuell og automatisk skalering av pods og noder. Så resultatmålet er oppnådd.

- *Utføre test av self healing mulighetene til Kubernetes ved feil på applikasjon.*
  Kubernetes self healing fikk vi aldri muligheten å se på, det ble ikke prioritert.

- *Kartlegge kostnader for bruk av Kubernetes, Docker og monitoreringsverktøy.*
  Som vi beskrev i 3.1.1 i sluttrapporten, så hadde vi opprinnelig en kost/nytte analyse, som vi senere fjernet etter rådføring med styringskomite og veiledere. Dette er rett og slett fordi vi følte den ble for enkel, og hvis vi skulle gjøre den ordentlig hadde vi brukt alt for mye tid på det.

- *Vurdere mulighetene for å implementere Kubernetes i Octopus Deploy.*
  Vi så litt på muligheten med å implementere Kubernetes i Octopus Deploy, men vi fikk ikke tid til å se grundig nok på det til en skikkelig vurdering. Andre deler av prosjektet fikk høyere prioritert.

3.1.3 Prosessmål

- *Økt kompetanse på bruk av Windows Server 2019, Server Core og Nano server.*
  Prosjektgruppen har økt sin kompetanse på Windows Server 2019, Server Core og Nano server. I tillegg har vi utvidet vår kompetanse på Linux.

- *Økt kompetanse og forståelse ved bruk av Docker, Kubernetes og Octopus Deploy på Windows.*
  Prosjektgruppen har styrket kompetanse og forståelse av både Docker og Kubernetes. Vi har virkelig gått i dybden og fått mye ny kunnskap. Vi fikk demonstrert Octopus Deploy av Nikolai Thingnes Leira (NHN). Det ble dessverre ikke anledning til å se nærmere på dette verkøyet.

- *Økt kompetanse og forståelse for hvordan man planlegger implementasjon av en ny løsning i en bedrift.*
  Studentene har fått mye kompetanse rundt planlegging av implementasjon av ny driftsmodell med tilhørende applikasjon. Det har vært tidkrevende å forstå alle delene som må være på plass, og forstå viktigheten av dem alle.

- *Økt kompetanse på bruk av MS Project og prosjektdokumentasjon.*
  Vi har brukt MS Project flittig til å oppdatere Gantt-diagrammet i løpet av prosjektperioden. Prosjektgruppen hadde noe erfaring med bruk av dette verktøyet fra før, men har tilegnet seg en del mer kunnskap om dette verktøyet. Vi har også lært mye om prosjektdokumentasjon, i forhold til timelister, møteinnkalling, møtereferater, bruk av kilder etc.

3.2 Bacheloroppgavens risikoanalyse

I forstudiet ble det gjennomført en risikoanalyse, målet med denne analysen var å avdekke og forebygge uønskede hendelser som negativt kunne påvirke prosjektets utførelse. I vår analyse kom vi frem til to hendelser som ble identifisert som kritiske for prosjektets suksess.

Hendelse	S	K	R	Prioritet
Manglende kunnskap eller informasjon	3	4	12	Tiltak skal implementeres
Testmiljø	2	5	10	Tiltak skal implementeres

Vi kom i forstudiet med tiltak som skulle iverksettes for disse hendelsene. Vi hadde flere gode støttespillere i NHN som vi kunne gå til for informasjon. Det tok litt tid å få på plass vårt testmiljø, men vi hadde alternative testmiljø vi prøvde oss på for å bygge kompetanse mens vi ventet på få på plass miljøet vi ble lovet. Ellers så møtte ikke prosjektgruppen særlig med utfordringer før på slutten, da tiden ble litt knapp. Vi hadde begge en uke der vi var syke hele uken, heldigvis inntraff dette samtidig så det gjorde ingen store skader.

Vi kan si at totalt sett, så har de de risikoreduserende tiltakene ført til at de identifiserte uønskede hendelsene ikke har ført til avgjørende negative konsekvenser for prosjektets resultat.

3.3 Måloppnåelse i forhold til framdriftsplanen

I starten av oppgaven, var vi nok litt for ivrig på hva vi ønsket å gjennomføre. Vi ville gjerne gjøre alt, og fant fort ut at det klarte vi aldri å gjennomføre. Dette kommer nok av begrenset erfaring med denne typen prosjekt, og at mye av oppgaven til tider var litt diffus. Den orginale framdriftsplanen ble revidert et par ganger, men vi lå alltid godt ann med å levere til tidsfristene vi hadde satt oss selv. Vi hadde noen perioder der vi brukte for lang tid på noen oppgaver, og som kanskje gjorde at vi fikk litt kort tid til dokumentasjonen.

3.4 Hva er det som gikk bra, mindre bra og alternativt hva kunne vi gjort annerledes

3.4.1 Hva gikk bra

Alle hovedmålene i oppgaven ble gjennomført. Vi har tilegnet oss mye kunnskap om verktøy, plattformer, operativsystemer og i tillegg fått vite mye om prosjektdokumentasjon, prosessverktøy og ikke minst hvordan NHN jobber. Det siste punktet er kanskje spesielt nyttig for prosjektgruppen da de skal starte i jobb hos oppgavestiller i August. Prosjektgruppen er stolte av hva de har og vise til, og hva de

har klart å gjennomføre. Vi er også tilfredsstilt med dokumentasjonen underveis og det endelige produktet.

3.4.2 Hva gikk mindre bra

Vi brukte kanskje for mye tid med å stange hodet i veggen med å få Windows containere til å fungere. Det var noen ganger litt vanskelig få hjelp, da NHN ansatte var opptatt, men det er forståelig da de har en jobb å gjøre. Det var kanskje spesielt utvikling som var travle, men når vi først fikk møte med de, så fikk vi god hjelp. Siden containere er lite brukt på NHN var det vanskelig å få svar på ting. Det ble mye samtaler på Github og på e-post til personer i Microsoft etc.

Prosjektgruppen syntes Word Online var et ekstremt dårlig verktøy til å bruke som samhandlingsverktøy og skiftet over til Google Docs når den endelige dokumentasjonen skulle skrives. Word Online ble etter hvert et frustrasjonsmoment.

3.4.3 Hva kunne vi gjort annerledes

Hvis vi skulle ha endret på noe, ville vi ha satt av mer tid til dokumentasjon underveis i prosessen. Vi ble litt oppslukt av det praktiske arbeidet i månedene mars - til slutten av april. Dette gjorde at vi havnet litt bakpå med dokumentasjon. VI er fornøyd med dokumentasjonen vi har levert, men vi kunne fått mer tid til korrektur og generell finpuss. Vi kunne også ha innsnevret oppgaven litt tidligere enn vi gjorde, da vi endte opp med å bruke litt for mye tid på oppgaver som ikke endte opp med å ha merkbar betydning for det endelige resultatet.

4. Konklusjon

Konklusjonen bygger på de utredningene vi gjorde i Designrapporten (Dønnem og Roksvaag, 2019a) og den praktiske jobben i Driftsrapporten (Dønnem og Roksvaag, 2019b).

Vi ser på de begrensningene og mulighetene vi har funnet med Kubernetes. Videre sammenligner vi en container-basert driftsmodell mot en tradisjonell driftsmodell. Før vi avslutsningsvis ser på statusen på containere på en Windows plattform.

4.1 Kubernetes som driftsverktøy

Det er vanlig å å oppfordre utviklere til å skrive programmer som går over flere plattformer, inkludert dedikerte On Premise servere, virtualiserte private skyer og offentlige skyer som AWS og Azure. Tradisjonelt er applikasjoner og verktøyet som støtter dem, nært knyttet til den underliggende infrastrukturen, så det er kostbart å bruke andre distribusjonsmodeller til tross for deres potensielle fordeler. Dette betyr ofte at applikasjoner er avhengig av et bestemt miljø i flere tilfeller.

PaaS forsøker å omgå disse problemene, men ofte på bekostning av å pålegge strenge krav på områder som programmeringsspråk og applikasjonsrammer.

I vår oppgave har vi sett at Kubernetes eliminerer infrastruktur låsing ved å gi kjernevirksomhet for containere uten å pålegge begrensninger. Kubernetes oppnår dette gjennom en kombinasjon av funksjoner innenfor Kubernetes-plattformen, inkludert Pods og Services.

Vi har sett at containere tillater at applikasjoner dekomponeres i mindre deler med tydelig separasjon. Abstraksjonslaget som er gitt for et individuelt container image, gjør det mulig å revurdere hvordan distribuerte applikasjoner er bygget. Denne modulære tilnærmingen muliggjør raskere utvikling av mindre, mer fokuserte team som hver er ansvarlig for bestemte containere. Det tillater oss også å isolere avhengigheter og gjøre større bruk av mindre komponenter.

Vi har sett at dette ikke oppnås med containere alene; det krever et system for å integrere og orkestrere disse modulære delene. Kubernetes oppnår dette delvis ved bruk av Pods. Containerne deler ressurser, for eksempel filsystemer, kernel namespaces og en IP-adresse. Ved å la containere samles på denne måten, fjerner Kubernetes fristelsen til å ta i bruk for mye funksjonalitet til et enkelt container image.

Konseptet med en Service i Kubernetes brukes til å gruppere sammen en samling pods som utfører en lignende funksjon. Services kan enkelt konfigureres for horisontal skalering og lastbalansering.

Vi har sett at distribusjonskontrolleren i Kubernetes forenkler en rekke komplekse administrasjonsoppgaver. Som for eksempel:

- Skalerbarhet. Programvaren kan distribueres for første gang på en skala ut over Pods, og distribusjonene kan skaleres inn eller ut når som helst.
- Versjonskontroll. Oppdatering av distribuerte deployments til nyere versjoner er enkel jobb, og det er like lett å rulle tilbake til en tidligere distribusjon hvis den gjeldende versjonen ikke er stabil.

Vi har også funnet ut at Kubernetes forenkler noen spesifikke distribusjonsoperasjoner som er spesielt verdifulle for utviklere av moderne applikasjoner. Disse inkluderer følgende:

- Horistontal autoskalering. Kubernetes skalerer automatisk Pods basert på bruk av spesifiserte ressurser (innenfor definerte grenser).
- Rullende oppdateringer. Oppdateringer til en Kubernetes-distribusjon blir orkestrert på "rullende måte" over distribusjonens Pods. Disse rullende oppdateringene blir orkestrert mens du jobber med valgfrie forhåndsdefinerte grenser på antall Pods som kan være utilgjengelige og antall reservedeler som kan eksistere midlertidig.

Vi har vist at Kubernetes er et godt verktøy for å drifte applikasjoner hvis applikasjonene er bygget for å kjøre som containere. Kubernetes er et mer avansert driftsverktøy enn det som brukes i dag på Drift 2. Det tilbys dog et webbasert grensesnitt der man kan gjøre endringer på deployments hvis man ikke er komfortabel med å bruke et kommandolinjeverktøy.

Kubernetes har vært stabilt under hele oppgaven, og vi har ikke opplevd feil, verktøyet gir god granulering for å gi tilgang til spesifikke deployments i Kubernetes. Automatisk skalering har overgått våre forventninger, og det fungerer meget godt. Vi har hatt erfaring med å bruke Kubernetes på forskjellige plattformer, i Azure ved bruk av AKS Engine, og en simulert On Premise løsning i Azure. Det er helt klart at Kubernetes er bygget for å kjøre i skyen. Dokumentasjonen som ligger ute nevner også skyen som den beste plattformen per nå for å kjøre Kubernetes. On Premise løsningen fungerte dårligere, og det var mye mer manuelt arbeid.

Vi mener at Kubernetes markerer et gjennombrudd for DevOps, fordi det tillater driftsteam å holde tritt med kravene til moderne programvareutvikling. Kubernetes tillater oss å utnytte maksimal nytte fra containere og bygge og deploye applikasjoner som kan kjøre hvor som helst, uavhengig av skysspesifikke krav. Dette er tydelig den effektive modellen for applikasjonsutvikling og drift mange har ventet på.

4.2. Tradisjonell driftsmetode vs container-basert

Når vi skal forsøke å sammenligne tradisjonell og container-basert driftsmetode blir det naturlig å se tilbake på det vi skrev i forstudierapporten (Dønnem og Roksvaag, 2019b) om potensielle fordeler med en container-basert metode. Der har vi trukket frem fire elementer som vi mente var potensielle fordeler. Nå som vi har gjennomført det meste av prosjektet har vi et godt grunnlag til å vurdere om disse fordelene er reelle eller ikke.

Forenkle utrulling, konfigurering og oppdatering av applikasjoner
Med å dele applikasjoner opp i mindre, isolerte containere som kan kommunisere med hverandre blir utrulling, konfigurering og oppdatering mye enklere da dette kan gjøres på en del av applikasjonen uavhengig av de andre delene. Vi nevnte også rullende oppdatering i kapittel 4.1 slik at du sømløst kan oppdatere deler av, eller hele, applikasjonen uten at brukere merker noen forskjell. Med Kubernetes overvåkes containerens helse under utrulling og hvis det oppdages feil, blir den automatisk rullet tilbake. Nedetid på applikasjoner ved oppdateringer kan bli en del av fortiden. Vi fikk mye erfaring med dette i vårt prosjekt når vi konverterte HelseID. Det ble da hyppig gjort endringer i konfigurasjon av distribusjoner og oppdateringer ble gjort flere ganger om dagen.

Øke effektivisering av hardware
Dette er en av de letteste fordelene å se med en container-basert driftsmetode. I stedet for å ha en virtuell maskin med sitt eget operativsystem for hver applikasjon eller tjeneste kan flere containere dele på samme OS og nettverkskobling. Hver container inneholder alt som trengs for å kjøre en spesifikk applikasjon eller tjeneste og ikke noe annet. Dette gjør dem ekstremt lettvektige og de tar opp færre ressurser i forhold til å ha et helt OS i grunn. Dette reduserer maskinvare- og datasenterkostnader.

Dette merket vi godt når vi fikk konvertert hovedkomponentet til HelseID, HelseID STS, og endte opp med et image på under 100mb og fikk rullet ut dette i Kubernetes som en fullverdig STS tjeneste.

Forbedre skalerbarhet
Dette var den mest oppsiktsvekkende fordelen for oss, både på container- og nodenivå. Med Kubernetes kunne vi konfigurere automatisk skalering basert på metrics som CPU-bruk. Når terskel for CPU-bruk ble overgått ville opp til 40 nye containere bli rullet ut på under 2 minutt hvis vi hadde tilgjengelige ressurser til å kjøre alle 40.

Det er når vi ikke har tilgjengelige ressurser vi må skille mellom container-basert driftsmetode on-premise og skyløsning. Ettersom vårt miljø var i Azure, en skyløsning, kunne vi konfigurere Kubernetes til å også automatisk skalere på nodenivå. Hvis vi da ikke hadde nok tilgjengelige ressurser til å kjøre opp alle 40 containere fra det tidligere eksempelet ville Kubernetes automatisk rullet ut nok nye VM-er og legge de til i clusteret for å kunne kjøre opp alle 40 containere. Alt dette skjedde på under 5 minutter.

Dette er ikke noe som kan reproduseres med en tradisjonell driftsmetode, uten ekstreme kostnader.

Med en on-premise løsning får du fortsatt de samme fordelene med skalering på containernivå, men fordelene med skalering på nodenivå er ikke like åpenbare.

Gjøre utvikling og drift (DevOps) med effektivt og agilt

I kapittel 4.1 snakket vi om at Kubernetes markerer et gjennombrudd for DevOps men dette gjelder ikke bare for Kubernetes. Det finnes alternativer til Kubernetes for orkestrering av containere og mange av de samme fordelene med Kubernetes gjelder for disse.

Siden alt som er nødvendig for å kjøre er pakket inn i containeren, kjører den på samme måte uansett hvordan maskin det gjøres på. For utviklere betyr dette at det ikke blir brukt tid på å sette opp miljøer, feilsøking av miljøspesifikke problemer etc. Containere sikrer konsistente miljøer fra utvikling til produksjon. De er konfigurert til å opprettholde alle konfigurasjoner og avhengigheter internt; du kan bruke samme container fra utvikling til produksjon og sørge for at det ikke er noen uoverensstemmelser eller manuelle inngrep.

For driftere har vi allerede gått gjennom mange av fordelene. Rask og enkel distribusjon, automatiske verktøy for skalering, forenklet oppdatering og disse er helt klart reelle fordeler etter vår erfaring med dette prosjektet.

Ulemper med container-basert driftsmetode

Det er så klart også ulemper, eller nærmere sagt utfordringer, med en container-basert driftsmetode. For de fleste kommer dette til å være ny teknologi og driftsteam må lære seg en ny måte å jobbe på samt lære seg flere nye verktøy. Nå er heldigvis dette ikke noe som er veldig uvanlig i denne bransjen, da ny teknologi introduseres hyppig men det er noe som må tas med i betraktningen.

For å få det meste ut av det burde det brukes i sammenheng med en skyløsning, som igjen kan bli en del ekstra arbeid og noen ting kan ikke kjøres i skyen på grunn av lovverk.

I tillegg kan det bli veldig mye arbeid med å konvertere eksisterende applikasjoner til å kjøres som containere, spesielt applikasjoner som har absolutte avhengigheter i Windows.

Men heldigvis er ikke dette et tilfelle av “enten eller”. Man kan benytte en tradisjonell driftsmetode på eksisterende applikasjoner som det blir for mye arbeid med å konvertere samtidig som man benytter en container-basert driftsmetode ved utvikling og drift av nye applikasjoner.

Alt dette fører til vår konklusjon som er at fordelene med en container-basert driftsmetode helt klart er reelle og noe som burde sterkt vurderes å implementere i NHN, spesielt for nye utviklingsprosjekt i fremtiden.

4.3 Status containere og Windows

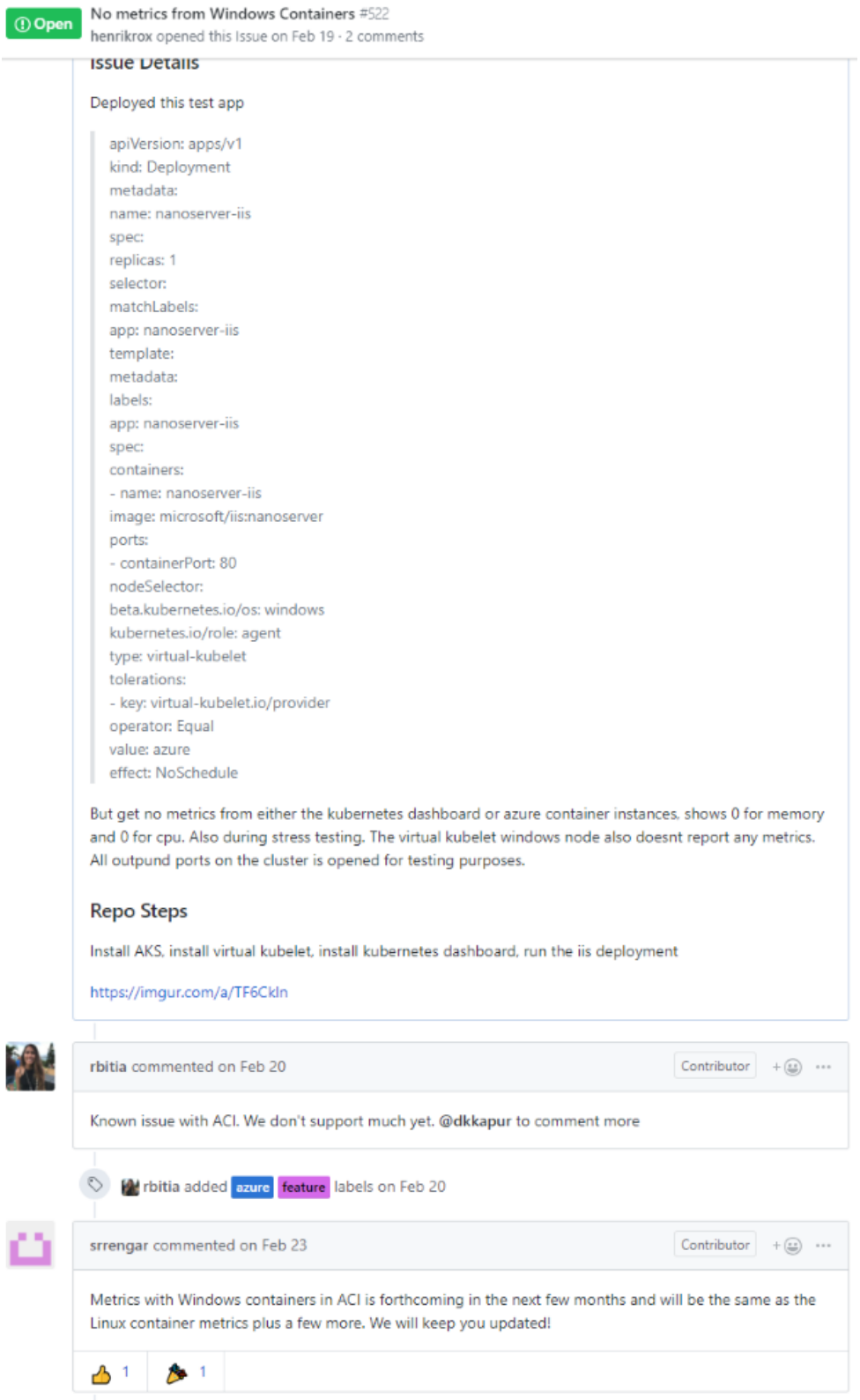
I forstudie skrev vi at vi ønsket å se på Windows som plattformen til å ta i bruk containere på. Etter intensiv prøving og mye feiling, kom vi frem til at dette ikke er en god fullverdig plattform enda. Vi har ikke dokumentert dette så mye i rapporten, og vi fant ut at vi har god nok tid til å dokumentere alt vi prøvde, som ikke gikk. Da hadde det fort blitt 500 sider.

Grunnen til at Windows ikke er en fullgod plattform for containere er mange, og vi skal her nevne de vi mener er viktigst.

- Azure sin egen containerløsning (AKS) støtter ikke Windows
  Azure tilbyr bruk av AKS i Azure til å ta i bruk Kubernetes. AKS støtter per dags dato bare Linux, og det er ikke mulig å bruke Windows hoster, og derfor heller ikke Windows containere. Microsoft har sagt mulighetene for Windows containere i Azure er på vei, og det ble i vår gjort tilgjengelig et søknadsskjema for å bli med i en preview kalt *Azure Kubernetes Service -- Windows Containers Preview* (Microsoft, 2019)

  Det er teknisk sett mulig å bruke Windows containere i AKS ved bruk av open source program kalt Virtual Kubelet. Problemet igjen med Virtual Kubelet er at det ikke støtter noe metrics enda. Som igjen fører til at automatisk skalering ikke vil fungere. Automatisk skalering er jo noe av det som gjør Kubernetes til et så bra verktøy, og uten dette mister Kubernetes mye av poenget. Vi var i

samtale med utviklerne av Virtual Kubelet på Github og fikk følgende som svar



Open **No metrics from Windows Containers** #522
henrikrox opened this Issue on Feb 19 · 2 comments

Issue Details

Deployed this test app

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nanoserver-iis
spec:
replicas: 1
selector:
matchLabels:
app: nanoserver-iis
template:
metadata:
labels:
app: nanoserver-iis
spec:
containers:
- name: nanoserver-iis
image: microsoft/iis:nanoserver
ports:
- containerPort: 80
nodeSelector:
beta.kubernetes.io/os: windows
kubernetes.io/role: agent
type: virtual-kubelet
tolerations:
- key: virtual-kubelet.io/provider
operator: Equal
value: azure
effect: NoSchedule
```

But get no metrics from either the kubernetes dashboard or azure container instances. shows 0 for memory and 0 for cpu. Also during stress testing. The virtual kubelet windows node also doesnt report any metrics. All outpund ports on the cluster is opened for testing purposes.

Repo Steps

Install AKS, install virtual kubelet, install kubernetes dashboard, run the iis deployment

https://imgur.com/a/TF6CkIn

rbitia commented on Feb 20 Contributor

Known issue with ACI. We don't support much yet. **@dkkapur** to comment more

rbitia added azure feature labels on Feb 20

srrengar commented on Feb 23 Contributor

Metrics with Windows containers in ACI is forthcoming in the next few months and will be the same as the Linux container metrics plus a few more. We will keep you updated!

1 1

“Metrics with containers in ACI is forthcoming in the new few months and will be the same as the linux container metrics plus a few more. We will keep you updated”

Konklusjonen blir derfor at AKS ikke er en fullverdig god løsning på Windows enda.

- AKS Engine med Windows containere
  Vi har et hybrid cluster kjørende med linux hoster og windows hoster. Her har vi dog samme problem, at vi ikke får metrics data fra windows hostene. Først trodde vi dette var på grunn av monitorerings verktøyene vi brukte, men når vi innså at 4-5 monitoreringsverktøy ikke viste metrics for hostene så forstod vi at dette var et vedvarende problem. Vi fikk heller ikke noe data inn i Azure sin innebygde container monitorerings løsning, ei heller i Kubernetes dashboardet. Derfor var heller ikke AKS Engine en god løsning for windows containere. Til Linux dog, så var AKS Engine veldig bra.

Avslutsningsvis vil vi si at Windows containere i Kubernetes er der, men bare så vidt. Vi vet at med Windows Server 2019, så har Microsoft lagt forholdene til rette for at containerdrift kan bli realisert med Windows containere også. Problemet er bare at vi ikke er der helt enda. Vi er dog helt sikker på at om et års tid, vil ting være annerledes.

5. Videre arbeid

Hvis NHN skulle ta i bruk bachelorgruppens arbeid, vil vi anbefale at det gjøres en omfattende kost/nytte analyse. Det anbefales videre at man utforsker muligheten ved å integrere Kubernetes med allerede eksisterende verktøy, her er spesielt Octopus Deploy sentralt. For monitorering vil vi anbefale å utforske mulighetene ved å ta i bruk Splunk til logging, alarmering og dashboards.

6. Kilder

Microsoft (2019): *"Azure Kubernetes Service -- Windows Containers Preview"*
Tilgjengelig fra:
https://forms.office.com/Pages/ResponsePage.aspx?id=v4j5cvGGr0GRqy180BHbR1zIfQJ4OeJCrEAguCCqD5dUOUJCUU1CMkFMV0I4UUtDUENCOTUxWEg1SC4u
(Hentet 01.05.2019)

Dønnem og Roksvaag (2019a): *Designrapport*

Dønnem og Roksvaag (2019b): *Driftsrapport*

Dønnem og Roksvaag (2019c): *Forstudierapport*