Magnus Torset
Simen Sagholen Førrisdal

Kubernetes i praksis

Bacheloroppgave i Drift av datasystemer
Veileder: Jostein Lund
Mai 2020

Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Magnus Torset
Simen Sagholen Førrisdal

Kubernetes i praksis

Bacheloroppgave i Drift av datasystemer
Veileder: Jostein Lund
Mai 2020

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Forstudierapport 3
Innholdsfortegnelse 5
1. Introduksjon 6
2. Bakgrunn for prosjektet 6
3. Prosjektmål 7
4. Interessenter og rammebetingelser 8
5. Kritiske suksessfaktorer 9
6. Risikoanalyse 11
7. Retningslinjer og standarder 14
8. Prosjektorganisering 15
Driftsdokument 17
Innholdsfortegnelse 19
1. Introduksjon 22
2. Konsepter og komponenter i Kubernetes 25
3. Installasjon av Kubernetes 50
4. Oppsett og konfigurering av Kubernetes cluster 53
5 Verktøy for administrering av Kubernetes cluster 58
6. Sette opp tjenester med Kubernetes 94
7. Kilder og videre lesning 146
Sluttrapport 149
1. Sammendrag 151
2. Endringer underveis 151
3. Arbeidsprosessen 151
4. Hva ville vi gjort annerledes 152
5. Videre arbeid 152

Magnus Torset
Simen Sagholen Førrisdal

Kubernetes i praksis

Forstudierapport

Innholdsfortegnelse

1. Introduksjon 6
2. Bakgrunn for prosjektet 6
2.1 Beskrivelse av problemer og behov 6
3. Prosjektmål 7
3.1 Effektmål 7
3.2 Resultatmål 7
3.3 Prosessmål 7
3.4 Prosjektets omfang 7
3.5 Prosjektets milepæler og hovedaktiviteter 8
4. Interessenter og rammebetingelser 8
4.1 Interessentanalyse 8
4.2 Rammebetingelser 9
5. Kritiske suksessfaktorer 9
5.1 Suksessfaktorer 9
5.2 Informasjonsbehov 10
6. Risikoanalyse 11
6.1 Risikovurdering 13
7. Retningslinjer og standarder 14
7.1 Krav til dokumentasjon 14
7.2 Krav til kvalitetsgjennomganger 14
7.3 Endringshåndtering 14
8. Prosjektorganisering 15

1. Introduksjon

Dette dokumentet skal gi en oversikt over prosjektet. Det er kun ment som en veiledning så sluttproduktet vil ikke nødvendigvis være nøyaktig som beskrevet her. Det skal si noe om den planlagte gangen i prosjektet, hvorfor vi gjør prosjektet og hva som kan forventes av det. Rapporten vil først ta for seg bakgrunnen for prosjektet med en forklaring av problemstillingen. Deretter tar vi for oss målene med prosjektet delt inn i effektmål, resultatmål, prosessmål og en avgrensning av omfang. I tillegg tar vi for oss de største milepælene i prosjektet og når vi ser for oss at de skal være ferdig. Videre beskrives rammebetingelsene og kritiske suksessfaktorer for prosjektet samt en interessentanalyse. Til slutt følger retningslinjer og standarder som for eksempel krav til dokumentasjon og hvordan prosjektet er organisert.

2. Bakgrunn for prosjektet

Dette prosjektet gjennomføres for å lære mer om Kubernetes og container teknologi generelt. Vi ønsker å tilegne oss den kunnskapen som er nødvendig for å kunne utrede Kubernetes og lage et dokument som andre kan bruke for å lære seg Kubernetes.

2.1 Beskrivelse av problemer og behov

Den tradisjonelle måten å drifte en tjeneste er å kjøre den direkte på en server eller på en virtuell maskin. En slik løsning byr på flere vanlige problemer ved skalerbarhet, flyttbarhet og konfigurasjon. Når man setter opp en tradisjonell server dimensjoneres den etter antatt trafikk, slik at den kan møte de antatte toppene og litt til. Med tiden vil bedriften vokse, og trafikken øker slik at man blir nødt til å skalere opp serveren. Med virtuelle maskiner er dette enkelt nok, men det vil øke kostnadene og som regel kreve en full omstart av maskinen, noe som betyr nedetid. Det tar ofte lang tid å starte serveren og få tjenesten opp igjen slik at nedetiden blir enda lengre.

En tjeneste som kjører på en tradisjonell server vil også være bundet til selve serveren. Den vil være avhengig av serverens operativsystem, og krever mer konfigurasjon for å fungere som den skal. Alt som påvirker serveren vil også kunne påvirke tjenesten, og vice versa. Dersom man ønsker å flytte tjenesten til en annen server, enten fordi man har fått et tilbud hos en annen skyleverandør eller man vil gjøre endringer i infrastrukturen, vil dette kreve at man tar ned hele tjenesten og setter den opp fra bunnen av på den nye serveren. I noen tilfeller vil det også kreve større endringer i selve tjenesten.

Containere er en relativt ny teknologi som løser mange av disse problemene. En container er et isolert miljø som inneholder kun det tjenesten trenger for å kjøre. De likner virtuelle maskiner, men har ikke noe eget operativsystem. I stedet kjører de på en container plattform, slik som Docker, som kjører på operativsystemet på serveren. Dette gjør dem mindre og enklere, slik at de starter raskere og krever mindre ressurser. Containere er også helt uavhengige av operativsystemet på verts-serveren. Dette betyr at en tjeneste som bruker f.eks PHP og MySQL kan kjøre identisk både på en Linux og en Windows server, og den kan også enkelt flyttes til en annen maskin ved å kopiere containeren til den nye serveren og starte den der.

Containere settes inn i et større system gjennom et orkestreringsverktøy slik som Kubernetes, som er det dette prosjektet skal handle om. I Kubernetes kan man sette opp clustere av flere maskiner som fordeler containere mellom seg for å balansere last. Siden man har et cluster av flere maskiner får man også andre fordeler. Om en maskin i clusteret skulle gå ned, kan en annen ta over kjøringen av containerne den hadde ansvar for. I Kubernetes kjører man gjerne flere identiske containere med samme tjeneste, som garanterer høy oppetid og tilgjengelighet. Kubernetes lar deg enkelt rulle ut mange nye containere på en gang og oppgradere disse én etter én så det ikke blir noen nedetid ved oppgradering. Tjenester kan også enkelt skaleres opp eller ned ved å legge til eller ta vekk containere, enten manuelt eller basert på ressursbruk og trafikk.

I dette prosjektet vil vi sette oss inn i hva Kubernetes er og hvordan det fungerer. Vi vil sette opp et Kubernetes-miljø fra bunnen og teste ulike bruksområder og use-case for Kubernetes. Deretter vil vi skrive en generell utredning som går i detalj om hvordan Kubernetes fungerer og hvordan man kan sette opp og bruke en Kubernetes-installasjon. Siden prosjektet ikke har noe konkret problem som skal løses, men i stedet er et læringsprosjekt, vil det heller ikke være noe konkret design som skal legges frem i designdokumentet. Derfor vil vi forsøke å skrive de relevante delene av designdokumentet inn i driftsdokumentet, og det resulterende dokumentet vil være den endelige dokumentasjonen for prosjektet.

3. Prosjektmål

3.1 Effektmål

- Forenkle opplæring av Kubernetes
- Gi et godt grunnlag for videre opplæring

3.2 Resultatmål

- Sette opp et miljø for testing av Kubernetes
- Sette opp tjenester som drar nytte av Kubernetes
- Fullføres innen 20.05.2020

3.3 Prosessmål

- Opparbeide kunnskap om Kubernetes og hvordan det brukes i praksis
- Kjenne til fordeler og ulemper ved å bruke containere

3.4 Prosjektets omfang

- Prosjektet skal inkludere et Kubernetes miljø
- Prosjektet skal ikke omfatte kostnadsanalyse knyttet til løsningene

3.5 Prosjektets milepæler og hovedaktiviteter

Aktivitet	Start	Frist
Bachelorprosjekt	**Tor 23-01-20**	**Ons 20-05-20**
Forstudierapport	Fre 24-01-20	Fre 21-02-20
Oppsett og testing	Man 24-02-20	--
Driftsdokument	Man 09-03-20	Fre 15-05-20
Sluttrapport	Man 18-05-20	Ons 20-05-20
Vurdering av samarbeid	Man 18-05-20	Ons 20-05-20

4. Interessenter og rammebetingelser

4.1 Interessentanalyse

Det er i hovedsak tre interessenter i dette prosjektet, oppdragsgiver, prosjektgruppen og veileder.

Oppdragsgiver er Stein Meisingseth ved institutt for datateknologi og informatikk(IDI) ved NTNU. De har godkjent problemstillingen vi kom med som noe de ønsker svar på. Det er i tillegg de som skal vurdere resultatet og gjennomføringen av prosjektet. Oppdragsgiver ønsker også å være informert om eventuelle problemer underveis og stiller visse krav til gjennomføring og dokumentasjon av arbeidet. Interessenten er fornøyd om de får svar på problemstillingen på en god måte og prosjektet blir riktig gjennomført etter de kravene de har satt.

Prosjektgruppen består av Magnus Torset og Simen Sagholen Førrisdal. Det er de som skal utføre arbeidet i prosjektet og har ansvar for å følge kravene som oppdragsgiver har satt.

Prosjektgruppen har behov for diverse ressurser som oppdragsgiver skal stille med. Interessenten er fornøyd hvis de lærer seg Kubernetes og leverer gode rapporter som har god læringsverdi.

Veileder er Jostein Lund. Han har ansvar for å veilede prosjektgruppen underveis i prosjektet og bidra med kunnskap og erfaring både i form av teknisk kunnskap og kunnskap om rapportskrivning. Interessenten er fornøyd hvis prosjektgruppen leverer et godt prosjekt som har tilfredsstillende dokumentasjon.

Interessent	**Suksesskriterier**	**Bidrag til prosjektet**
Eksterne - Oppdragsgiver	- Få svar på problemstilling	- Ressurser
Interne - Prosjektgruppen - Veileder	- Lære Kubernetes og lære bort Kubernetes - Et godt gjennomført prosjekt	- Ansvar - Kunnskap, veiledning

4.2 Rammebetingelser

- Prosjektet skal bruke Microsoft Teams til møter og møteinnkalling
- Prosjektet skal bruke Microsoft Teams til å dele dokumenter
- Alle dokumenter skal skrives i Google Docs
- Etter hver skriveøkt skal det lages PDF- og docx-versjoner av dokumentet som legges i Teams for versjonskontroll og deling med veileder
- Diagrammer og tegninger skal lages i Draw.io
- Serverne skal være virtuelle maskiner i vmWare eller Microsoft Azure
- Prosjektet skal være ferdig innen 20.05.2020

5. Kritiske suksessfaktorer

5.1 Suksessfaktorer

En av suksessfaktor for prosjektet er at vi i prosjektgruppen klarer å lære oss Kubernetes og klarer å få et godt bilde av hvordan det brukes i praksis. Det er et stort system med mange nye konsepter og det virker som om det ikke er noen grense for hvor mye i dybden man kan gå. Vi har begrenset med tid så det er viktig at vi ikke henger oss opp i små detaljer over lengre tid, men heller fokuserer på det store bildet.

En annen suksessfaktor er at det vi skriver er forståelig for folk som ikke kan noe om Kubernetes. Spesielt for praktisk anlagte folk kan det være vanskelig å forstå hvordan ting fungerer uten å prøve det ut selv. Det er viktig å finne en balanse mellom å kun forklare det som er nødvendig og å gå i detalj der det trengs. Det kan også være vanskelig for

prosjektgruppen, etter å ha lært om Kubernetes i flere måneder, å sette seg i skoene til noen som ikke vet noe. Det er viktig at grunnprinsippene blir nøye forklart og at vi ikke hopper over noen steg så leseren har en sjanse til å henge med.
En tredje suksessfaktor er at vi får tilgang på det nødvendige arbeidsmiljøet fra oppdragsgiver. Prøving og feiling i praksis vil være en stor del av prosjektet, slik at tidligst mulig tilgang til det nødvendige arbeidsmiljøet kan være kritisk for prosjektets suksess. I tillegg må miljøet ha den nødvendige kapasiteten vi trenger for å gjøre det vi skal. Det er ikke klart hvor omfattende det endelige systemet vil være, og det er viktig at arbeidsmiljøet har de nødvendige ressursene eller har muligheter for oppskalering.

5.2 Informasjonsbehov

Gjennom hele prosjektet skal det være jevn kommunikasjon med både oppdragsgiver og veileder. Oppdragsgiver ønsker informasjon om fremgang og statusrapporter underveis og prosjektgruppen har behov for jevnlig kontakt med veileder for å svare på spørsmål og få tilbakemelding på arbeidet så langt. Spesielt vil prosjektgruppen ha behov for veileder for å gå gjennom rapportene. Forstudierapport kommer relativt tidlig i prosjektet så det er naturlig at det blir flere møter i begynnelsen med spørsmål og gjennomgang av denne. Etter forstudie handler det stort sett om å tilegne seg kunnskap og prøve det ut i praksis så da er det sannsynligvis ikke behov for like mange møter. Når vi begynner å nærme oss slutten er det igjen naturlig at det blir flere møter siden vi da forhåpentligvis driver med ferdigstilling av dokumenter. Vi planlegger å ha faste møter hver 2-3 uke, men gjerne oftere etter behov.

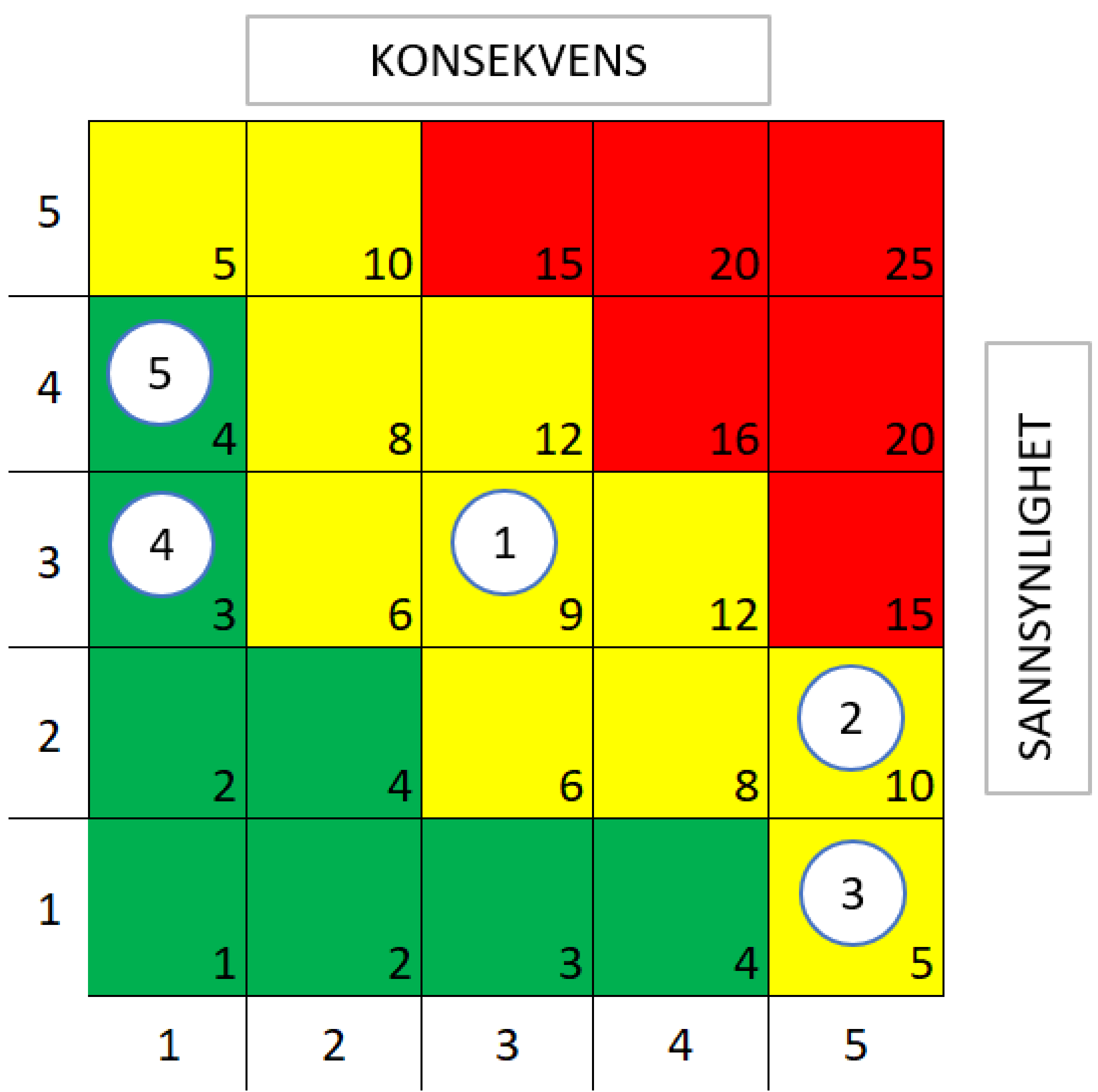
6. Risikoanalyse

Hendelse	Konsekvens	Tiltak	Kommentar
Sykdom i prosjektgruppen	Styrket arbeidspress på de resterende medlemmene. Opphold i arbeidet. Økt tidspress.	De resterende medlemmene må ta på seg flere oppgaver i sykdomsperioden. Prosjektet må omprioriteres for å sikre gjennomførelse	Sykdom er vanskelig å forutsi, og prosjektet strekker seg over et langt tidsrom. Alvorligheten og varigheten av sykdommen vil påvirke prosjektet på forskjellige måter. Siden prosjektgruppen kun består av to personer kan alvorlig eller langvarig sykdom ha stor innvirkning på arbeidet.
Datatap	Enkelte deler eller hele prosjektet må gjøres om igjen	Jevnlig lagring og eventuelt backup av arbeid. Lagring av dokumenter i sky	Siden MS Teams og Google Drive brukes til dokumentasjon og lagring er det liten sannsynlighet for store datatap
Manglende ressurser	Hele eller deler av prosjektet kan ikke gjennomføres	Ta kontakt med veileder og oppdragsgiver for å forsøke å få tilgang til de nødvendige ressursene. Ta i bruk egne ressurser hvis mulig	Oppdragsgiver har de nødvendige ressursene og kravet til ressurser er relativt lite, slik at sannsynligheten for at sannsynligheten for at ressursmangel blir et problem er lav
Manglende kompetanse i prosjektgruppen	Prosjektet blir ikke tilstrekkelig gjennomført	Diskutere situasjonen med veileder. Bruke mer tid på egenopplæring Omformulere problemstillingen.	En del av prosjektet er å opparbeide seg kunnskap om container-teknologi, som er godt dokumentert på internett

Uenigheter i prosjektgruppen	Gruppen klarer ikke samarbeide og oppgaven blir ikke fullført	Ha en åpen dialog Ta opp eventuelle problemer tidlig Involvere veileder som en tredjepart	Siden gruppen består av kun to medlemmer kan større uenigheter få store konsekvenser for prosjektets gjennomførelse

6.1 Risikovurdering

Nr.	Hendelse	S	K	R	Risiko
1	Sykdom i prosjektgruppen	3	3	9	Middels
2	Datatap	2	5	10	Middels
3	Manglende ressurser	1	5	5	Middels
4	Manglende kompetanse i prosjektgruppen	1	3	3	Lav
5	Uenigheter i prosjektgruppen	1	4	4	Lav



7. Retningslinjer og standarder

7.1 Krav til dokumentasjon

Ved slutten av prosjektet skal det leveres

- Forstudierapport
- Driftsrapport
- Sluttrapport
- Vurdering av samarbeid

7.2 Krav til kvalitetsgjennomganger

Rapporter og fremdrift vil bli vurdert under møter med veileder, Jostein Lund, underveis i prosjektet. Endelig kvalitetskontroll og vurdering utføres av Jostein Lund, Stein Meisingseth og en ekstern sensor.

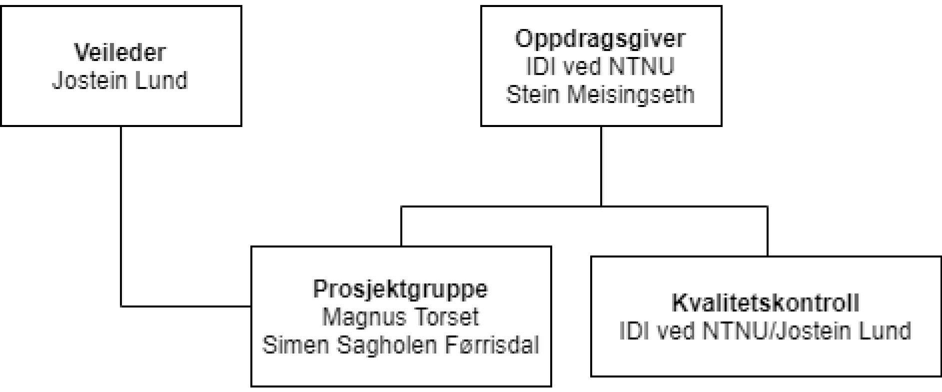
7.3 Endringshåndtering

Endringsønsker kommer fra oppdragsgiver, brukere, prosjektdeltakerne selv og andre interessenter. Endringsønsker er vanlig og skal behandles formelt og forretningsmessig. Framgangsmåten for håndtering av endringsønsker er:

1. Dokumenter endringens innhold
2. Analyser konsekvensene for prosjektet
3. Beregn eventuell kost/nytt
4. Godkjennelse og aksept
5. Logg endringen
6. Juster planene
7. Informer interessentene
8. Gjennomfør endringen

8. Prosjektorganisering

Prosjektgruppen har to medlemmer, Magnus Torset og Simen Sagholen Førrisdal. Siden det bare er to medlemmer ser vi det ikke som nødvendig å utnevne en prosjektleder. Oppdragsgiver er Stein Meisingseth ved IDI. Veileder er Jostein Lund. Kvalitetskontroll utføres av veileder underveis i prosjektet og i tillegg av oppdragsgiver når prosjektet er ferdig.



Magnus Torset
Simen Sagholen Førrisdal

Kubernetes i praksis

Driftsdokument

Innholdsfortegnelse

1. Introduksjon 22
1.1 Dokumentets hensikt og bruk 22
1.2 Definisjoner 22
1.3 Figurliste 24
2. Konsepter og komponenter i Kubernetes 25
2.1 Hva er Kubernetes og containere? 25
2.1.1 Kubernetes 25
2.1.2 Cluster 25
2.1.3 Containere og Pods 26
2.1.4 Container Images 27
2.2 Ressurser 28
2.2.1 Ressursfiler 28
2.2.2 Pod 29
2.2.3 Deployment og ReplicaSet 31
2.2.4 Service 33
2.2.4.1 ClusterIP 33
2.2.4.2 NodePort 34
2.2.4.3 LoadBalancer 35
2.2.4.4 ExternalName 36
2.2.5 Ingress 36
2.2.6 ConfigMap 39
2.2.7 StatefulSet 41
2.2.8 PersistentVolume og PersistentVolumeClaim 42
2.2.8.1 HostPath 42
2.2.8.2 Ekstern filtjener 43
2.2.8.3 PersistentVolumeClaim 43
2.2.9 Secrets 43
2.2.10 DaemonSet 43
2.2.11 Namespace 44
2.2.12 Autentisering 44
2.2.12.1 Role og ClusterRole 45
2.2.12.2 RoleBinding og ClusterRoleBinding 45
2.2.12.3 ServiceAccount 46
2.2.13 Custom Resources og Custom Controller 47
2.3 Nettverk i Kubernetes 48
3. Installasjon av Kubernetes 50
3.1 Forhåndskrav 50
3.2 Skru av swap 50
3.3 Dependencies 50
3.4 Docker 51
3.5 Kubernetes 52
4. Oppsett og konfigurering av Kubernetes cluster 53
4.1 Oppsett av Kubernetes-cluster 53

4.2 Melde inn noder 54
4.3 Oppsett av MetalLB 54
4.4 Oppsett Ingress-Controller 55
4.5 Metrics Server 56
5 Verktøy for administrering av Kubernetes cluster 58
5.1 Helm 58
5.1.1 Installasjon av Helm 58
5.1.2 Charts og repositories 58
5.1.3 Utrulling av charts 59
5.1.4 Konfigurering av charts 59
5.2 Operators 62
5.2.1 Hva er Operators 62
5.2.2 Operators i praksis 62
5.3 Monitorering 64
5.3.1 Prometheus 64
5.3.2 Grafana 64
5.3.3 Installasjon 64
5.3.4 Konfigurasjon 69
5.3.4.1 Deaktivere admin-bruker 69
5.3.4.2 Grafana Dashbord 71
5.3.5 Varsling 76
5.4 Cert-manager 80
5.4.1 Installasjon 80
5.4.2 Bruk 86
5.4.2.1 Sikring gjennom Ingress 86
5.4.2.2 Selvstendige sertifikater 89
6. Sette opp tjenester med Kubernetes 94
6.1 Webserver med autoskalering 94
6.1.1 Bygge Image 94
6.1.2 Utrulling i Clusteret 96
6.1.3 Skalering og Oppgradering 99
6.2 Storage Provisioner og Statefulset 104
6.2.1 Storage Provisioner 104
6.2.2 Oppsett av MySQL med StatefulSet 107
6.2.2.1 ConfigMap og Service 107
6.2.2.2 StatefulSet 108
6.3 Oppsett av utviklingsmiljø 112
6.3.1 GitLab med Registry 112
6.3.1.2 Installasjon 112
6.3.1.3 Konfigurasjon av GitLab 114
Lage administratorbruker 114
Sette opp SSH-nøkkel for en bruker 115
6.3.1.4 Oppsett av Docker Registry 117
6.3.1.5 Bruk av GitLab og Docker Registry 118

Lage grupper 118
Lage prosjekter 119
Bruk av Git 120
Legge til Docker Images 122
6.3.2 Jenkins 124
6.3.2.1 Installasjon 124
6.3.2.2 Konfigurasjon 130
6.3.2.3 Testing 135
6.3.2.4 Testprosjekt 138
7. Kilder og videre lesning 146

1. Introduksjon

Denne oppgaven skal være en innføring i Kubernetes for de som har et godt teknisk grunnlag og ønsker å komme i gang med containere. Den tar for seg installasjon og oppsett, og går i dybden på de komponentene og konseptene som utgjør Kubernetes. Gjennom eksempler vises også praktisk bruk av disse. Oppgaven skal gi det nødvendige grunnlaget for å selv kunne sette opp og drifte et Kubernetes cluster.

1.1 Dokumentets hensikt og bruk

Dette dokumentet skal ta for seg hva Kubernetes er, hvordan det installeres og hvordan det brukes. Det er ment til å brukes til opplæring av noen som allerede har et godt teknisk grunnlag, men som nå ønsker å benytte seg av den nye teknologien Kubernetes gir tilgang til. Vi antar at brukeren har en grei kjennskap til Linux og vanlige kommandoer der som curl, wget, hvordan man navigerer osv. Vi kommer først til å ta for oss hva Kubernetes er og de komponentene og konseptene man er nødt til å ha en viss forståelse for for å effektivt bruke Kubernetes. Deretter går vi inn på hvordan det installeres på en bare-metal server. Vi har så en oversikt over nyttige verktøy for administrering og hvordan de installeres og brukes, og til slutt noen eksempler som viser mer i dybden både hvordan verktøyene brukes og hvordan de forskjellige ressursene kan benyttes. Kapittel 2 er ment som en introduksjon til Kubernetes og hvordan det fungerer så det anbefales å lese dette kapittelet først for å få en grei forståelse av Kubernetes før man begynner på eksempler.

1.2 Definisjoner

Node	En node er en maskin som er knyttet til et cluster. Det er stort sett to typer noder. Master og worker.
High Availability (HA)	High Availability betyr å sørge for høy, da helst 100%, oppetid for en applikasjon eller tjeneste. Ofte gjøres dette ved å kjøre flere instanser av tjenesten, slik at den umiddelbart kan erstattes hvis den går ned
Image	Et image er en mal til hvordan en container skal settes opp. Det inneholder aller programmene man trenger for å kjøre den tjenesten man vil, men er utenom det veldig begrenset i forhold til andre ting man vanligvis ville trengt om det var en vanlig maskin. De har ofte også noen forhåndsutfylte konfigurasjonsfiler som kan endres på med miljøvariabler.
Registry/Docker Registry	Et registry er en tjeneste man kan kjøre selv eller leie, som lagrer containere i et format Docker forstår så de kan bli hentet ned og brukt. Et eksempel på et registry er hub.docker.com som er tilgjengelig å bruke for alle, men kun lar deg lage to repositories med en gratis konto. Vi laget derfor vårt eget på docker.torset.me.
swap	swap vil si at det settes av plass på harddisken på en maskin som brukes som minne når det fysiske minnet fylles opp. Det lar

	maskinen bruke mer minne enn den egentlig har
Pod-nettverk	Et Pod-nettverk sørger for intern kommunikasjon mellom Pods i et Kubernetes cluster. Det finnes mange ulike Pod-nettverk, blant annet Calico og Flannel
Bare-Metal	En bare-metal server er en maskin som kjører direkte på den fysiske maskinen, uten noe virtualiserings-lag i mellom. Selv om vi bruker virtuelle maskiner omtaler vi maskinene våre som bare-metal for å skille vår løsning fra de forskjellige sky-løsningene
Replica	En replika er en eksakt kopi. I Kubernetes betyr dette en nøyaktig kopi av en Pod, med den samme programvaren og de samme innstillingene. Det eneste som skiller dem er hostnavnet.
Repository (repo)	Et repository (forkortet repo) er en samling av data. Apt bruker repository om en samling av programmer så når vi snakker om Dockers apt-repository snakker vi om det generelle området hvor Docker lagrer sine installasjonsfiler for apt-pakkesystemet. Helm bruker repository om en samling av charts. Git bruker også repository som en betegnelse på en samling av filer, ofte også kalt et prosjekt.
API-server	API står for application programming interface. Et API er et programmatisk grensesnitt mot en programvare-komponent eller et system, som definerer hvordan andre systemer eller annen programvare kan bruke den. Kubernetes bruker en API-server for å gjøre clusteret tilgjengelig for de ulike komponentene. Dette brukes hver gang man oppretter en ny ressurs, og for å hente filer eller data for ressursbruk.
GPG-nøkkel	En nøkkel som brukes til å verifisere opphavet til en pakke. Det er vanlig å bruke slike nøkler eller lignende metoder for å bekrefte ovenfor de som skal laste ned programvaren din at man er den man utgir seg for å være.

1.3 Figurliste

Figur 2.1: Et typisk cluster med én masternode og tre workere	s.26
Figur 2.2: Fremstilling av Deployment	s.31
Figur 2.3: Service og selector	s.33
Figur 2.4: Oversikt over routing fra internett gjennom Ingress	s.39
Figur 2.5: Pod henter inn konfigurasjon og filer fra ConfigMap	s.40
Figur 2.6: StatefulSet lager Poder med tilhørende PersistentVolume og unik ID	s.42
Figur 2.7: PersistentVolumeClaim oppretter og binder et volum til en Pod	s.43
Figur 2.8: Et DaemonSet vil lage en Pod på hver node	s.44
Figur 2.9: Ressurser i ulike Namespace kan fortsatt kommunisere med hverandre	s.44
Figur 2.10: Oversikt over Role og ClusterRole	s.47
Figur 2.11: Kubernetes bruker flere nettverkslag. Intern- og Pod-nettverket kan kommunisere, men kan ikke nås fra utsiden	s.49
Figur 5.1: Operatoren fungerer som en utvidelse av Kubernetes APIet, og håndterer en bestemt Custom Resource	s.63
Figur 6.1: Lese-forespørsler går gjennom en Service, mens skrive-forespørsler går direkte til master	s.111

2. Konsepter og komponenter i Kubernetes

Kubernetes er et stort system med mange bruksområder, og som består av mange, komplekse deler. I dette kapittelet vil vi gå gjennom de grunnleggende konseptene bak Kubernetes, samt gå i detalj om de viktigste og mest brukte komponentene.

2.1 Hva er Kubernetes og containere?

Her vil gjøre rede for konseptet Kubernetes, i tillegg til å forklare Poder og containere, som er de mest grunnleggende delene av Kubernetes.

2.1.1 Kubernetes

Kubernetes baserer seg på container-teknologi, som har som mål å kjøre applikasjoner isolert og effektivt i små, mobile containere som kan kjøre hvor som helst. Selve containerne kjører i et underliggende container miljø, mens Kubernetes er et verktøy som tilbyr effektiv administrasjon av store og komplekse container-systemer. Det muliggjør automatisk utrulling, skalering og orkestrering av containere, og selve infrastrukturen er mobil og skalerbar. For å få mest mulig ut av Kubernetes, settes det som regel opp i store clustre, hvor flere servere kan dele på den samme lasten.

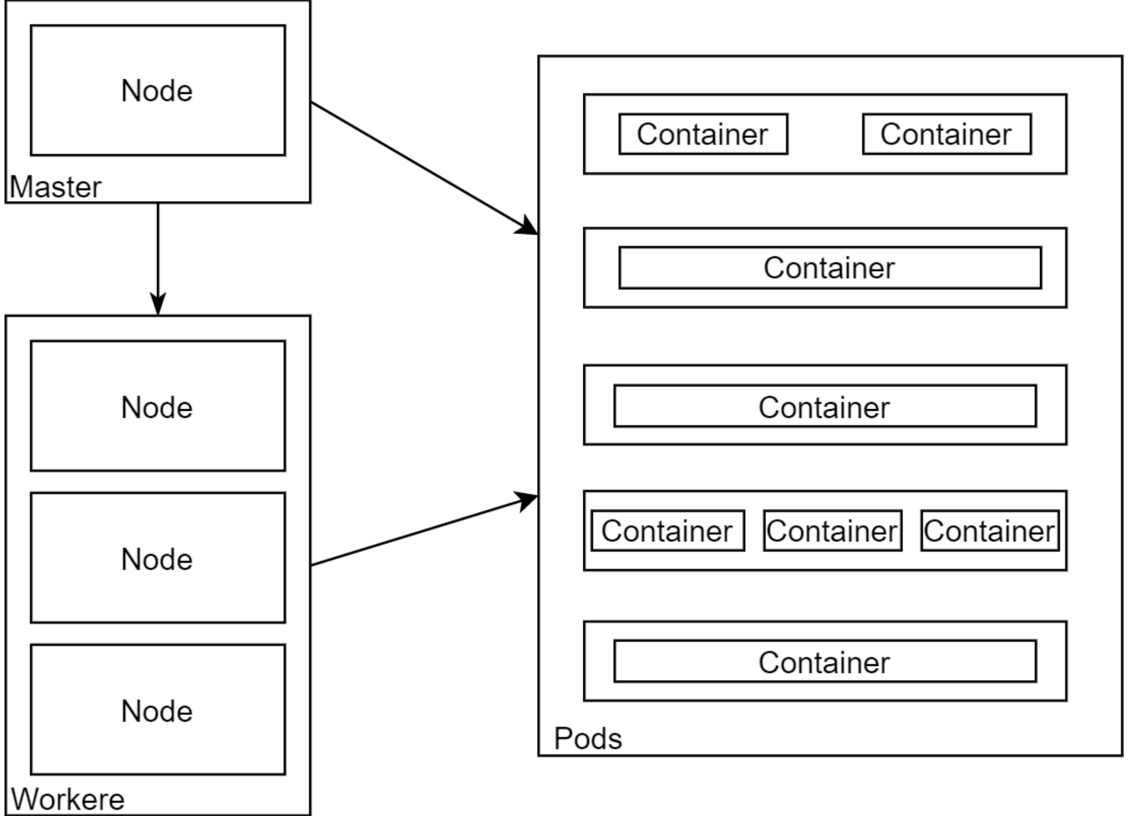
Den store fordelen med containere og Kubernetes i forhold til tradisjonelle løsninger er skalerbarheten, mobiliteten og oppetiden det tilbyr. Med VMer eller fysiske maskiner vil man ofte sette opp maskinen med tanke på applikasjonen man skal kjøre, og applikasjonen vil også være avhengig av at maskinen den kjører på er riktig konfigurert. Hvis man vil skalere opp serveren for å møte økt trafikk må man ta ned hele serveren for å gjøre oppgraderinger, og man vil få betydelig nedetid. Det samme gjelder ved oppdateringer eller flytting av infrastruktur. Da må hele tjenesten tas ned, og man må i verste fall gjøre omfattende konfigurasjon på nytt.

I Kubernetes trenger man kun et cluster i bunnen for å kunne kjøre en hvilken som helst applikasjon. Dette betyr også at man kan kjøre den samme applikasjonen hvor som helst uten å måtte gjøre noen endringer. Ved oppdateringer vil den gamle containeren kjøre helt til den nye er klar, slik at man kan rulle ut en oppdatering helt uten nedetid. Man kan også enkelt skalere opp tjenesten ved å rulle ut nye containere som deler på den samme lasten. Etter hvert som applikasjonen vokser og man møter økende trafikk, kan også clusteret enkelt utvides ved å legge til flere maskiner i clusteret.

På grunn av mobiliteten og skalerbarheten er Kubernetes veldig godt egnet for å kjøre applikasjoner med stor variasjon i last. Så lenge du har mer enn én maskin i clusteret er High Availability allerede satt opp og hvis du går for en skyløsning trenger du ikke bry deg om infrastruktur i det hele tatt og trenger bare tenke på applikasjonene du tenker å kjøre.

2.1.2 Cluster

Et Kubernetes cluster er en samling av maskiner som jobber sammen for å kjøre containere. Clusteret blir styrt fra en master-node. Man kan ha flere master-noder avhengig av hvor høye krav man har til oppetid.[1] Masteren kjører vanligvis ikke andre Pods enn de som brukes for å administrere clusteret som for eksempel clusterets interne DNS og API-serveren. I tillegg til master har man også workere. Det er disse nodene som kjører vanlige tjenester som for eksempel en webserver. Man kan ha så mange workere man vil. Hvis en worker blir tatt ned vil alle tjenestene som kjørte på den noden bli startet på nytt på en annen node.



Figur 2.1: Et typisk cluster med én master node og tre workere.

2.1.3 Containere og Pods

En Pod er en samling av én eller flere containere. I motsetning til virtuelle maskiner har containere ikke sitt eget operativsystem, men låner kernel og andre nødvendige ting fra maskinen de kjører på. De er Linux-baserte og består stort sett kun av et filsystem. Containere er ofte veldig begrenset i forhold til installerte programmer og tjenester. Dette er for å gi raskest mulig oppstart og avslutting og for å holde størrelsen lav. Hver gang en Pod startes på nytt blir den tatt ned og bygget på nytt ut fra et image, og rask oppstart har dermed stor betydning for hvor lang tid dette tar[2]. Siden de bygges helt på nytt betyr dette også at de ikke beholder noen form for filer eller konfigurasjon som ikke er bygget inn i imaget når de startes på nytt. Senere i dokumentet skal vi gå gjennom noen vanlige måter som brukes for å omgå dette. Dette gjør at de blir veldig bærbare og kan startes likt på så mange og forskjellige maskiner du vil.

2.1.4 Container Images

Alle containere baserer seg på et image bygget spesifikt for den oppgaven de skal utføre. Et image er en komprimert mal av containeren. Når vi sier at en container blir bygget mener vi at den lager en ny container basert på denne malen. De deler kjerne med operativsystemet på vertsmaskinen, og trenger derfor ikke noe eget operativsystem. I tillegg inneholder de kun de filene og programmene de trenger for å utføre sin oppgave, slik at de er veldig små og kompakte. Når man spesifiserer et image i Kubernetes, vil Kubernetes forsøke å finne det på Docker Hub. Docker Hub er et slags "marked" for container images, hvor bedrifter og organisasjoner legger ut container-versjoner av sin programvare til offentlig bruk. Når vi snakker om et nginx-image, er dette et image laget av NGINX selv, som inneholder en container-versjon av deres webserver. Det finnes images for de fleste populære løsninger, slik som Apache, MySQL, Node.js og Java, og også hele operativsystemer slik som Debian eller Ubuntu.

Siden containere ikke har noe eget operativsystem kan de heller ikke startes på nytt. I stedet vil de tas helt ned og bygges opp igjen fra bunnen. Det sørger for at containeren er helt uten korrupte eller uvedkommende filer, og garanter at den vil fungere hver gang den starter. Det betyr også at den er stateless, altså at den ikke lagrer noe som helst data om hvilken tilstand den var i forrige gang den kjørte. Det er opp til et orkestreringsverktøy slik som Kubernetes å gi containeren den informasjonen den trenger for å fungere.

2.2 Ressurser

For å administrere Kubernetes og få Pods til å oppføre seg som man vil, trenger man måter å konfigurere både selve containeren og hvordan containeren skal kommunisere med omverdenen på. I Kubernetes er konfigurasjonen delt inn i forskjellige ressursfiler skrevet i YAML-format. For å rulle ut Pods kan man for eksempel lage en Deployment-ressurs, for å legge inn eller endre på en konfigurasjonsfil i Podene kan man lage en ConfigMap-ressurs og for å samle og dele disse Podene med omverdenen kan man lage en Service- og Ingress-ressurs. De forskjellige typene ressurser har hvert sitt mål og bruksområde, mange har overlapp i funksjonalitet, men tilbyr ofte i tillegg mer spesialiserte funksjoner. I kapittel 5 og 6 skal vi se nærmere på hvordan disse ressursene brukes i praksis i litt mer spesifikke situasjoner, men i dette kapittelet handler det mest om å få et inntrykk av hva de forskjellige ressursene er og hva de kan brukes til.

2.2.1 Ressursfiler

Alle ressurser i Kubernetes blir laget ved hjelp av en ressursfil. Disse er i YAML-format og skal inneholde alt Kubernetes trenger å vite om ressursen for å sette den opp på riktig måte. Ressursfiler kan variere fra noen linjer til lange dokumenter avhengig av hvor komplisert oppsettet er og hvor mye ekstra informasjon Kubernetes trenger for å sette opp ressursen så den fungerer med ditt system. En ressursfil kan også inneholde flere ressurser. Man legger til en ressurs ved å kjøre `kubectl apply -f <ressursfil.yaml>`. I noen tilfeller kan man også legge til en ressurs ved hjelp av en kommando. Da vil det bli generert en ressursfil ut fra de parametrene du bruker i kommandoen og standardverdier. La oss se på ressursfilen til en Pod som et eksempel.

2_2_ressurser\pod_nginx_eksempel.yaml

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

Dette er et eksempel på en ressursfil til en Pod som kjører webtjeneren nginx. Alle ressurser inneholder feltene `kind:` og `apiVersion:`. Disse er der for å fortelle Kubernetes hvilken type ressurs det er vi vil legge til og hvilken versjon av Kubernetes APIet den bruker.

Alle ressurser inneholder også et felt for metadata. Dette er for å hjelpe andre ressurser, spesielt Servicer, å samle ressurser som hører sammen. Metadata feltet er nødt til å inneholde et navn og minst én label, men labelen kan være hva du vil. Hvis du for eksempel

vil skille på hvilke Pods som er i produksjonsmiljøet og utviklingsmiljøet kan du bruke labelen `environment: prod` og `environment: dev`. Du kan ha så mange labeler du vil.

Det neste hovedfeltet er spec. Det deles også av alle ressurser, men innholdet er ulikt avhengig av hvilken type ressurs det er du setter opp.

2.2.2 Pod

En Pod er en av de mest grunnleggende ressursene i Kubernetes. Man skal helst sette opp en Pod ved å bruke en annen ressurs som for eksempel en Deployment eller et StafeulSet, men det er mulig å sette de opp for seg selv. En frittstående Pod kan blant annet brukes til å raskt sette opp en Pod som er på innsiden av clusteret sitt nettverk for å kjøre kommandoer mot ressurser som ikke deles med utsiden. Man kan lage en Pod med en ressursfil eller ved å kjøre `kubectl run --generator=run-pod/v1 nginx --image=nginx`. Her spesifiserer vi at vi skal lage en Pod med `--generator=run-pod/v1`, gir den navnet `nginx`, og spesifiserer hva slags image vi vil bruke med `--image=nginx`[2]. Når man kjører en kommando slik som denne vil det automatisk genereres en ressursfil. Vi kan se denne ved å kjøre `kubectl edit pod nginx`.

Vi så litt på ressursfilen til en Pod i kapittelet om ressursfiler. I tillegg til de feltene som er felles for alle ressurser er det også et felt `spec:` som er avhengig av hvilken type ressurs det er du legger til. Vi ser litt nærmere på hvordan det kan se ut for en Pod her:

2_2_ressurser\pod_phpmyadmin_eksempel.yaml

```yaml
spec:
  containers:
    - name: phpmyadmin
      image: phpmyadmin/phpmyadmin
      ports:
        - containerPort: 80
      env:
        - name: PMA_HOST
          value: mysql-0.mysql
        - name: PMA_PORT
          value: "3306"
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-cred
              key: root-password
```

Dette er spec-feltet, for en Pod som kjører phpmyadmin. Siden denne Poden kun inneholder én container er all konfigurasjonen her rettet mot den containeren. Her blir det blant annet definert navnet på containeren, hvilket image den skal bruke og hvilke porter som skal åpnes. Det blir i tillegg satt en del variabler som er spesifikke for vårt miljø. Miljøvariabler er

en ganske vanlig måte å konfigurere containere på. Mange containere er satt opp til å ta imot punkter i konfigurasjonsfiler som miljøvariabler. Disse kan enten skrives rett inn i ressursfilen til Poden som er gjort her eller man kan lage et ConfigMap som vi skal se på senere. Det refereres også til en Secret. En Secret er Kubernetes sin ressurs for å unngå å lagre ting i klartekst. Secrets kommer vi også tilbake til senere.

Containers-objektet er en liste, slik at man kan ha flere containere i samme Pod. Hver container blir et nytt element i listen.

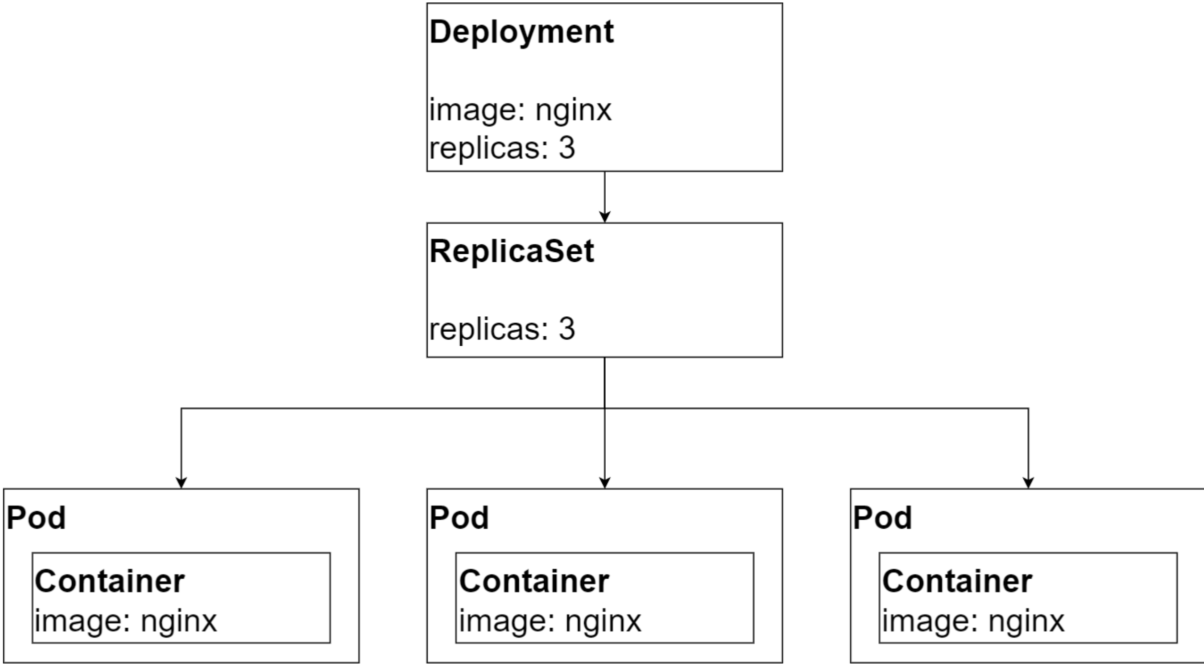
2_2_ressurser\pod_flere_containere_eksempel.yaml

```
spec:
    containers:
      - name: nginx
        image: nginx:latest
        ports:
          - containerPort: 80
      - name: monitoring
        image: monitoring
        ports:
          - containerPort: 5000
```

Når man er ferdig å definere den første containeren bruker man `-` for å si at man skal begynne på en ny container. Det aller vanligste bruksområdet for en Pod med flere containere er når man ønsker tilleggs- eller hjelpe-funksjonalitet til hovedcontaineren. Ofte er dette en “sidevogn” som for eksempel brukes til å overvåke hovedcontaineren, men det kan også være en reverse-proxy for en webserver. I praksis kan man kjøre slike ekstratjenester i en egen Pod, men da vil man kunne støte på problemer ved at Podene ikke stopper og starter samtidig, eller at den ene har en feil mens den andre kjører som før. Ved å putte dem i samme Pod vil de fungere som én enhet, og de vil alltid starte og stoppe samtidig. Dersom den ene ikke kan starte vil heller ikke den andre starte, og hvis den ene crasher vet vi at begge containerne stopper. Vi vil også være sikre på at de behandles på samme måte av Kubernetes.

2.2.3 Deployment og ReplicaSet

En Deployment er den vanligste måten å sette opp nye Pods på. Deployments brukes til å lage et sett med Pods. Et slikt sett kalles et ReplicaSet og lages automatisk når du lager en Deployment. I motsetning til en Pod hvor hver Pod kan inneholde forskjellige containere, inneholder et ReplicaSet flere kopier, som Kubernetes kaller replicas, av den samme Poden[7]. Et ReplicaSet sin oppgave er å holde så mange slike replicas i gang som er spesifisert og å fordele de på de forskjellige nodene i clusteret.



Figur 2.2: Fremstilling av Deployment

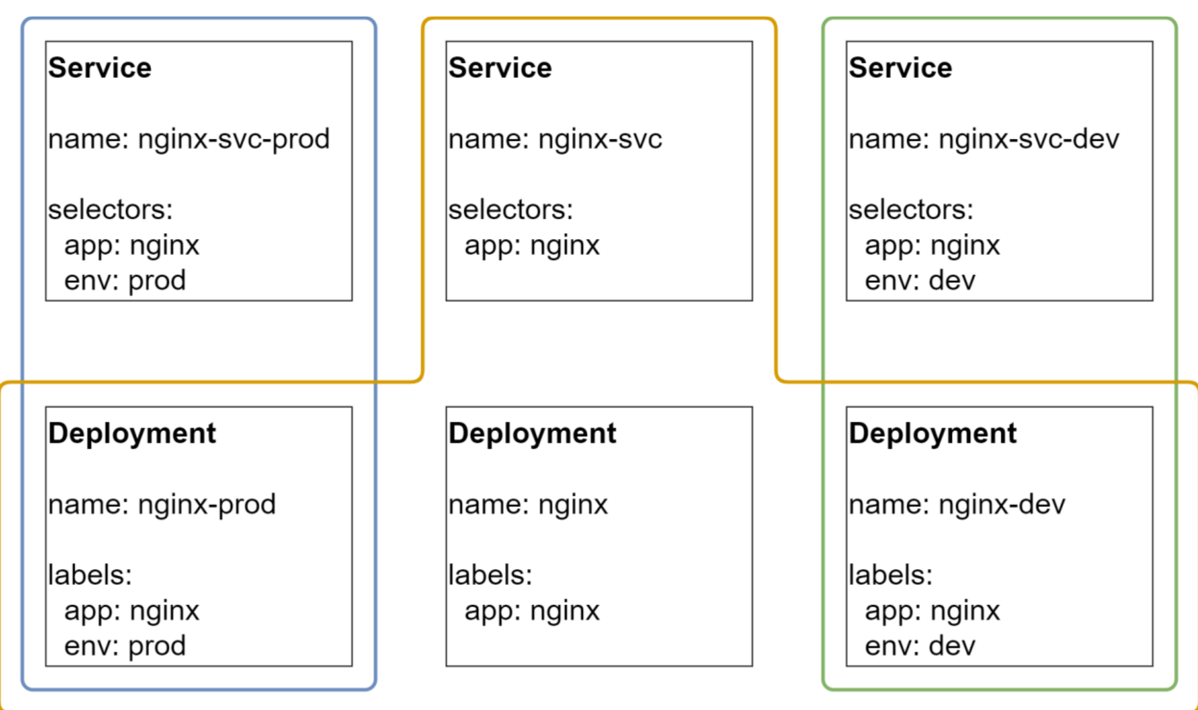
2_2_ressurser\deployment_nginx_eksempel.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: 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
```

Her er et eksempel på en Deployment sin ressursfil. Under det første feltet med `spec:` så ser vi at man kan spesifisere antall replicas og et felt som heter selector. Selector spesifiserer hvilke Pods Deploymenten skal gjelde for. Videre har vi template og under her finner vi malen for hvordan Podene Deploymenten lager skal se ut. Her legger vi inn den samme konfigurasjonen som brukes for å lage ressursfilen til en Pod. En Deployment kan inneholde alt en Pod kan inneholde.

2.2.4 Service

En Service brukes for å samle flere Pods og gi de et felles oppkoblingspunkt ved hjelp av en selector. En selector ser etter Podder med en viss label og samler disse. Det finnes fire typer Servicer. ClusterIP, NodePort, LoadBalancer og ExternalName[3]. Alle Servicetypene utenom ExternalName har innebygd lastbalansering. Som standard vil den kun velge en tilfeldig Pod, men det er også mulig å få den til å velge Podene i rekkefølge, en såkalt round robin. Med litt ekstra oppsett kan man også bruke andre måter for lastbalansering som færrest åpne forbindelser og korteste forventet responstid ved hjelp av noe som heter IPVS proxy modus.



Figur 2.3: Service og selector

2.2.4.1 ClusterIP

ClusterIP er den mest grunnleggende av Service typene. Den gir Servicen en intern IP i clusteret, men eksponerer den ikke utenfor clusterets interne nettverk. Siden Servicen ikke blir eksponert ut av clusteret er man nødt til å gjøre dette ved hjelp av for eksempel en Ingress og Ingress Controller. Å sette opp en ClusterIP Service er veldig enkelt, spesielt om man allerede har en Deployment. Da trenger man bare å kjøre `kubectl expose deployment <deploymentnavn>`. Hvis Deploymenten har definert én eller flere `ContainerPort:` vil disse automatisk bli valgt eller man kan spesifisere hvilken port som blir eksponert ved å legge til `--port=<port>`. Man kan selvfølgelig også sette det opp med en vanlig ressursfil. Den kan for eksempel se slik ut.

```yaml
2_2_ressurser\service_eksempel.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
  labels:
    app: nginx
spec:
  ports:
  - name: http
    port: 80
  selector:
    app: nginx
```

Her ser vi at det velges hvilken port som skal brukes for å nå Podene og det spesifiseres også en selector. Med selectoren `app: nginx` vil denne Servicen gjelde for alle Pods som har labelen `app: nginx`. Om det kun spesifiseres `port:` vil Servicen ikke gjøre noen oversettelse av innkommende og utgående port. Om vi hadde spesifisert `targetPort:` ville Servicen tatt imot forespørsler på porten definert av `port:` og sendt de til Podden på porten definert av `targetPort:`. Vi ser også at type Service ikke ble spesifisert, det kunne den vært, men er ikke nødvendig siden ClusterIP automatisk vil bli valgt om ikke noe annet er oppgitt.

2.2.4.2 NodePort

NodePort gir deg mulighet til å nå en Pod inne i clusteret fra utsiden ved å bruke ip-adressen til noden den kjører på og et portnummer. Om det er flere Poder på samme node vil disse bli lastbalansert. Dette kan være nyttig for å enkelt nå en Pod fra utenfor clusteret eller for å kun nå Podder på spesifikke noder. NodePort blir ofte brukt under testing for å sette opp en Pod uten å måtte ha tilgang til innsiden av clusteret eller sette opp en Ingress. Du må derimot vite hvilken node Poden du vil nå kjører på, noe som lett kan endre seg, så det er ikke anbefalt å bruke NodePort med mindre du vet at Poden alltid vil kjøre på samme node. Med standard oppsett bruker NodePort porter i området 30000-32767, som er høye, ikke-standard portnumre. Dette kan by på problemer, da mange rutere ofte er satt opp for å blokkere HTTP/HTTPS trafikk på andre porter enn 80 og 443, eller blokkerer alle ikke-standard porter generelt. Dette er enda en grunn til at NodePort ikke er egnet som endelig løsning. NodePort kan settes opp ved hjelp av kommandoen `kubectl expose deployment <deploymentnavn> --type=NodePort` eller ved å bruke en ressursfil. Den kan for eksempel se slik ut.

```yaml
2_2_ressurser\service_nodeport_eksempel.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc-np
  labels:
```

```
    app: nginx
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30001
      name: http
  selector:
    app: nginx
```

Vi ser at den er ganske lik ressursfilen til en ClusterIP, men vi har lagt til `type: NodePort` under spec og under ports har vi også lagt til et felt `nodePort: 30001`. Det er denne porten 30001 som vil bli eksponert av nodene. For å nå denne Servicen går vi da til <Node-IP>:30001 og vi vil nå den eller de Podene som kjører på den noden.

2.2.4.3 LoadBalancer

En Service av typen LoadBalancer har kanskje et litt misvisende navn. Den gjør ikke noe mer lastbalansering enn de andre Servicene, men er ment for å koble til en ekstern lastbalanserer. Dette er den eneste av Service typene som gir en ekstern IP, men hvordan den tilegner seg disse er avhengig av hvilket miljø du bruker. I en standard Kubernetes installasjon på bare-metal vil den ikke klare å dele ut IP-adresser siden den ikke vet hvilke som er tilgjengelig. Derfor er man nødt til å bruke eksterne tjenester som MetalLB for at den skal klare å dele ut IP-adresser. Hvis man kjører Kubernetes hos en skyleverandør er det skyleverandøren som står for utdeling av disse IP-adressene. Utenom tildeling av eksterne IP-adresser og åpning av porter på disse fungerer den som en helt vanlig ClusterIP. For å sette opp en Service av typen LoadBalancer kan du som med NodePort og ClusterIP gjøre dette ved hjelp av en kommando, `kubectl expose deployment <deploymentnavn> --type=LoadBalancer`, eller ved å bruke en ressursfil som kan se slik ut.

2_2_ressurser\service_loadbalancer_eksempel.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc-lb
  labels:
    app: nginx
spec:
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 8080
  type: LoadBalancer
```

Vi ser her at `type:` nå er `LoadBalancer`. Vi har også `targetPort:` som er hvilken port Servicen skal bruke for å nå Poden og `port:` som bestemmer hvilken port som skal brukes

på den eksterne IP-adressen. For å sjekke hvilken IP vi har fått tildelt kan vi kjøre kommandoen `kubectl get services`. Vi vil da få en oversikt over alle Servicer i Namespacet og hvilke interne og eksterne IP-er de har.

2.2.4.4 ExternalName

ExternalName er en av de enkleste, men også minst brukte Servicetypene. Den oversetter et Service-navn til et DNS-navn. Dette kan brukes for å lage en Service for en ressurs som er utenfor clusteret som i dette eksempelet hvor en ekstern database blir gitt en Service for at en enkelt skal kunne oppdatere alt som er avhengig av denne databasen om oppsettet endrer seg. Det gjør det også lettere å på et senere tidspunkt migrere databasen til innsiden av clusteret. Da trenger man bare å endre på hva Servicen peker til og siden Podene som bruker Servicen allerede peker til Servicen trenger man ikke konfigurere Podene på nytt.

2_2_ressurser\service_externalname_eksempel.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

Legg merke til at det ikke blir oppgitt noen selector. Dette er fordi den ikke er ment for å samle interne tjenester, men for at interne tjenester skal kunne bruke en Service for å nå eksterne tjenester.

2.2.5 Ingress

En Ingress er måten Kubernetes tar inn nettrafikk fra utsiden av clusteret og fordeler det dit den skal på innsiden av clusteret[4]. For å kunne bruke Ingress ressursen er man nødt til å ha en Ingress Controller. Den kommer ikke standard med Kubernetes så det er noe du må sette opp selv. Vi viser hvordan dette gjøres i kapittel 4.4 Ingress Controller. Ingress Controlleren har en ekstern IP og det er hit man sender trafikken. Ingress Controlleren router all trafikken den mottar på denne IP-adressen og sender den videre til en Service basert på hva som er spesifisert i Ingressene. Siden all trafikk går til den samme IP-adressen bruker man DNS-navn til å bestemme til hvilken Service trafikken går. Det betyr at man må ha en egen DNS-server som oversetter alle hostnavnene man vil bruke til IP-adressen til Ingress Controlleren. Trafikk som kommer på et hostnavn som ikke er spesifisert i en Ingress vil bli avslått. En relativt standard Ingressfil kan se slik ut.

```yaml
# 2_2_ressurser\ingress_eksempel.yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
    name: db-ingress
spec:
    rules:
    - host: database.example.com
      http:
        paths:
        - backend:
                serviceName: phpmyadmin
                servicePort: 80
```

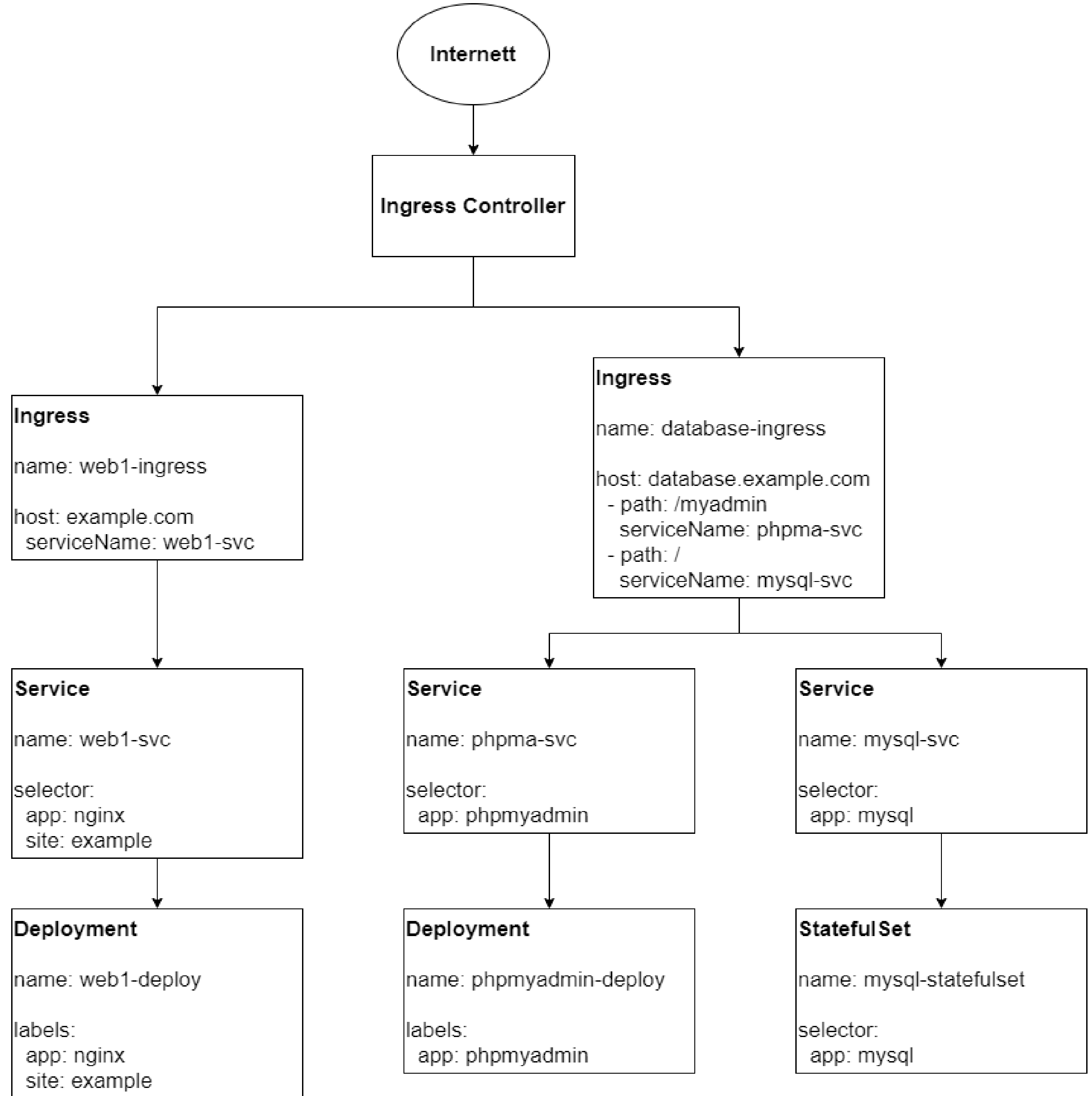
Under `spec:` definerer vi regler for routingen. `host:` sier at denne Ingressen gjelder for all trafikk med URL database.example.com. Under `backend:` definerer vi hvor denne trafikken skal gå internt i clusteret og hvilken port det skal sendes på. Denne Ingressen tar altså all trafikk som kommer inn med URL `database.example.com` og sender det til Servicen `phpmyadmin` på port 80. Man kan også route basert på path, altså alt bak / i URLen. Det kan se slik ut.

```yaml
# 2_2_ressurser\ingress_path_eksempel.yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
    name: db-ingress
spec:
    rules:
    - host: database.example.com
      http:
        paths:
        - path: /myadmin
          backend:
                serviceName: phpmyadmin
                servicePort: 80
```

Her har vi lagt til en regel for path så all trafikk som kommer til `database.example.com/myadmin` blir sendt til `phpmyadmin`. Det er viktig å merke seg at Ingress Controlleren ikke endrer noe på pathen før den sender den videre så med mindre phpmyadmin forventer å motta noe med path `/myadmin` vil dette føre til krøll. Det er mulig å endre denne oppførselen slik at den stripper vekk `/myadmin` før den sender det videre, men oppsettet for dette avhenger av hvilken Ingress Controller du bruker. I Ingress-Nginx kan man gjøre det ved å bruke rewrite-target. Oppsettet for dette ser slik ut.

```yaml
2_2_ressurser\ingress_annotation_eksempel.yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
    name: db-ingress
    annotations:
        nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
    rules:
    - host: database.example.com
      http:
        paths:
        - path: /myadmin(/|$)(.*)
          backend:
              serviceName: phpmyadmin
              servicePort: 80
```

Ingress-Nginx bruker feltet `annotations:` for å legge til funksjonalitet som er utenfor det Ingress ressursen tilbyr. Annotations er et felt som brukes for ikke-identifiserende informasjon om en ressurs i motsetning til Labels som er identifiserende informasjon om en ressurs. Det vil si at du ikke kan bruke Selectorer på en annotations, ellers er de veldig like. Annotations er for det meste brukt for at tredjeparts verktøy skal kunne ha tilgang til ekstra informasjon som ikke er mulig å legge andre steder i ressursfilen. I dette tilfellet spesifiserer vi Annotationen `nginx.ingress.kubernetes.io/rewrite-target: /$2` som tar en variabel $2. Denne variabelen spesifiserer hva det er du vil beholde av pathen. Med $2 vil den droppe alt før den andre skråstreken i URLen. I `path:` legger vi til `(/|$)(.*)` bak pathen vår. Dette er en RegEx streng for å velge alt. Med dette oppsettet kan vi nå gå til `database.example.com/myadmin`, men phpmyadmin vil oppfatte det som om vi gikk til `database.example.com/` og hvis vi går til for eksempel `database.example.com/myadmin/login` vil phpmyadmin tro at pathen er `database.example.com/login`.



Figur 2.4: Oversikt over routing fra internett gjennom Ingress

Her ser vi et diagram over et relativt vanlig oppsett. En pakke blir sendt til Ingress Controlleren på dens eksterne IP. Ingress Controlleren sjekker URLen som kommer inn og bestemmer hvor trafikken skal sendes ved å sammenligne URLen med Ingressene og se om den passer noen av de. Den blir så sendt videre til den Servicen som er oppgitt i Ingressen for så å bli sendt til en av Poddene i Deploymenten Servicen gjelder for. Hvis vi for eksempel går til `example.com` vil DNSen vår først sende oss til Ingress Controlleren, så vil Ingress Controlleren se at det finnes en Ingress som passer forespørselen vår og sende oss til `web1-svc`, derfra velger Servicen en av Poddene i Deploymenten `web1-deploy` og vi er fremme.

2.2.6 ConfigMap

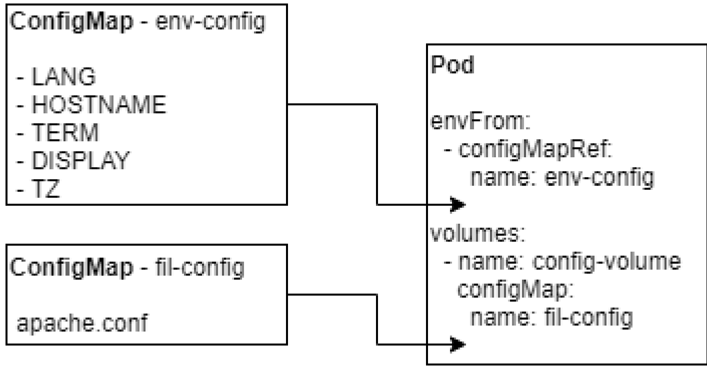
ConfigMaps er en ressurs som brukes for å laste inn ekstra konfigurasjon til en container. Dette kan være enkle miljøvariabler eller hele filer. Hvis vi for eksempel ønsker å erstatte en av konfigurasjonsfilene i en container med vår egen kan vi lage et ConfigMap av filen vi vil

erstatte den med ved å kjøre kommandoen `kubectl create configmap <map-navn> --from-file=<fil-path>`. Man kan også lage et ConfigMap av en hel mappe ved å spesifisere en mappe istedenfor en fil. Det er også mulig å lage en fil med miljøvariabler man ønsker å sette i containeren og lagre dette som et ConfigMap. Da bruker man `kubectl create configmap <map-navn> --from-env-file=<fil-path>`. ConfigMaps er ikke knyttet til spesifikke Pods, men kan importeres til en Pod i ressursfilen til Poden. Dette oppsettet ser slik ut.

2_2_ressurser\configmap_pod_eksempel.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-test
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
      envFrom:
      - configMapRef:
          name: env-config
  volumes:
    - name: config-volume
      configMap:
        name: fil-config
  restartPolicy: Never
```

Her har vi både et ConfigMap med en fil og et ConfigMap med miljøvariabler. ConfigMapet med miljøvariabler legges til med `envFrom:` for så å spesifisere hvilket ConfigMap du vil bruke. For å legge til filen bruker vi `volumeMounts:` og legger til navnet på volumet og hvor det skal mountes. For å lage dette volumet legger vi til `volumes:` og skriver inn navnet på volumet igjen og sier hvilken ConfigMap vi vil bruke. Vi har et eksempel som bruker ConfigMaps i kapittel 6.1 hvor vi setter opp en webserver.



Figur 2.5: Pod henter inn konfigurasjon og filer fra ConfigMap

2.2.7 StatefulSet

Av og til er det nødvendig at en Pod beholder enkelte deler av seg selv mellom restarter og det er ikke mulig å legge dette til i imaget eller med ConfigMaps. Dette kan være om man er nødt til å nå en spesifikk Pod i settet med nettverksnavn eller om man trenger lagring som beholdes om en Pod skulle starte på nytt. For eksempel hvis du kjører en database er det nødvendig at den beholder alle databasefilene sine selv om den blir startet på nytt. Til dette har man StatefulSets[8]. StatefulSets fungerer nesten på samme måte som en Deployment, men med et par viktige forskjeller som gjør at de egner seg bedre for enkelte typer oppgaver.

Når en Deployment lager en Pod vil den få tildelt et navn som ender med en streng tilfeldige tall og bokstaver for å unngå overlapp. I et StatefulSet vil hver Pod få tildelt hvert sitt tall fra 0 og oppover og vil beholde dette når det oppdateres og restartes. Dette gjør det mulig å nå spesifikke Pods med den interne DNSen siden Pod-navnet er hostnavnet til Poden. Det lages også en label med dette Pod-navnet så det er mulig å lage en Service og knytte den til spesifikke Podder i settet. Dette kan være viktig for for eksempel en database der du kun vil at én av Podene skal ha ansvar for skriveoperasjoner, men vil bruke flere Poder for å lastbalansere leseoperasjoner.

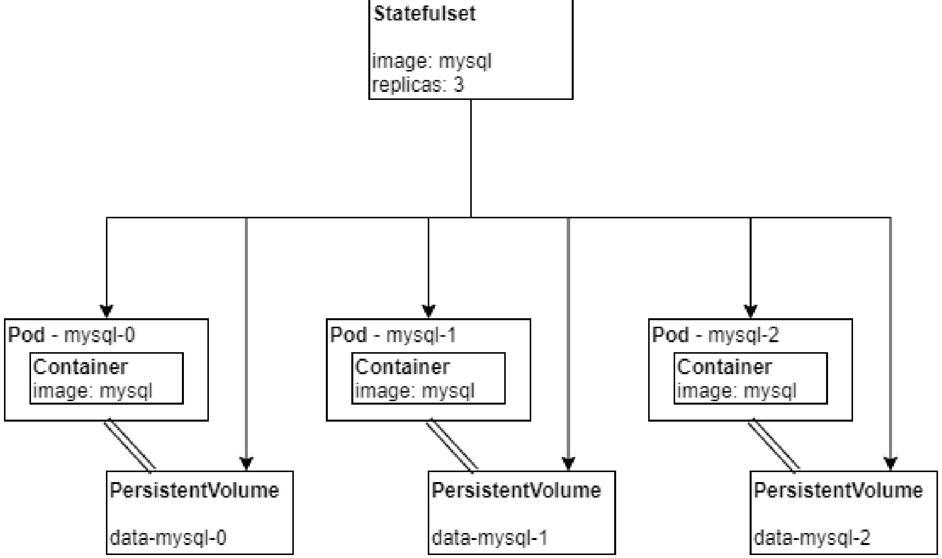
I forhold til lagring vil det bli laget et PersistentVolume for hver Pod i settet og dette volumet vil ikke bli slettet når StatefulSetet slettes.

2_2_ressurser\statefulset_eksempel.yaml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
```

```
volumeClaimTemplates:
- metadata:
    name: www
  spec:
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: "my-storage-class"
    resources:
      requests:
        storage: 1Gi
```

Dette er et eksempel tatt fra Kubernetes sin dokumentasjon om StatefulSets. Oppsettet er veldig likt det for en Deployment med et par unntak. Et StatefulSet er nødt til å ha en Service tilknyttet seg. Denne defineres under `serviceName:`. Vi definerer også `terminationGracePeriodSeconds: 10` for Poden. StatefulSetet vil da vente i 10 sekunder før den begynner å avslutte den neste Podden. Denne bør økes om Poden din bruker lang tid på å avslutte. I tillegg defineres `volumeMounts:` og `volumeClaimTemplates:`. Her settes det opp et PersistentVolume og et PersistentVolumeClaim for Podden. Vi ser nærmere på PersistentVolumes i neste kapittel og vi ser nærmere på bruk av StatefulSet i kapittel 6.2 Storage Provisioner og StatefulSet.



Figur 2.6: StatefulSet lager Poder med tilhørende PersistentVolume og unik ID

2.2.8 PersistentVolume og PersistentVolumeClaim

Et PersistentVolume er ment til å løse problemet med at en Pod ikke beholder data mellom restarts. PersistentVolume tilbyr et sted å lagre data som beholdes selv om Podden bygges på nytt. Det er i hovedsak tre typer PersistentVolume. HostPath, ekstern filtjener og administrert av skytjenesten du bruker [5]. Siden vi kjører på Bare Metal har vi kun mulighet til å snakke om HostPath og ekstern filtjener.

2.2.8.1 HostPath

HostPath er den enkleste måten å sette opp varig lagring på. Det betyr i hovedsak at man deler et område på nodens filsystem med en Pod. Problemet med denne metoden er at det ikke er mulig å flytte Pods mellom noder siden filene kun ligger på én av nodene. Det er

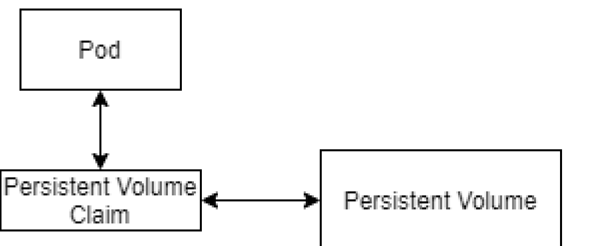
heller ikke mulig å sette opp en HostPath for en Deployment med flere replicas. Dette gjør den svært begrenset i bruk.

2.2.8.2 Ekstern filtjener

Man kan også bruke en ekstern filtjener til å sette opp et Persistent Volume. Dette kan være for eksempel over NFS, SMB eller iSCSI. Flere av disse protokollene krever at du setter opp en Storage Provisioner. Dette er et lag mellom filtjeneren og Kubernetes som tar seg av blant annet tildeling av ressurser til hver Deployment og hvilke rettigheter Podder skal ha til å bruke volumet. Hvilken provisioner du trenger avhenger av hvilken protokoll du ønsker å bruke. Det er også viktig å passe på at driverne til protokollen du ønsker å bruke er installert på alle nodene i clusteret. Man oppretter også en StorageClass som tildeles provisioneren. StorageClass brukes når du setter opp et PersistentVolumeClaim for å spesifisere hvilken provisioner som skal brukes og hvordan volumet skal behandles.

2.2.8.3 PersistentVolumeClaim

Et PersistentVolumeClaim binder et volum til en Pod. Her legger man inn informasjon om blant annet hvor stor del av volumet Podden skal ha tilgang til, om det er én eller flere Pods som skal ha tilgang til volumet og eventuelt hvilke rettigheter som skal settes. Hvis det ikke finnes noe volum må det opprettes et nytt. Dette volumet forblir gjennom omstart, og det samme volumet brukes neste gang det refereres til. Et volum må enten lages av en administrator eller deles ut automatisk av en Storage Provisioner. En Storage Provisioner krever et eget oppsett som vi viser i kapittel 6.2 Storage Provisioner og StatefulSet. Der ser vi også nærmere på bruk av PersistentVolume og PersistentVolumeClaim.



Figur 2.7: PersistentVolumeClaim oppretter og binder et volum til en Pod

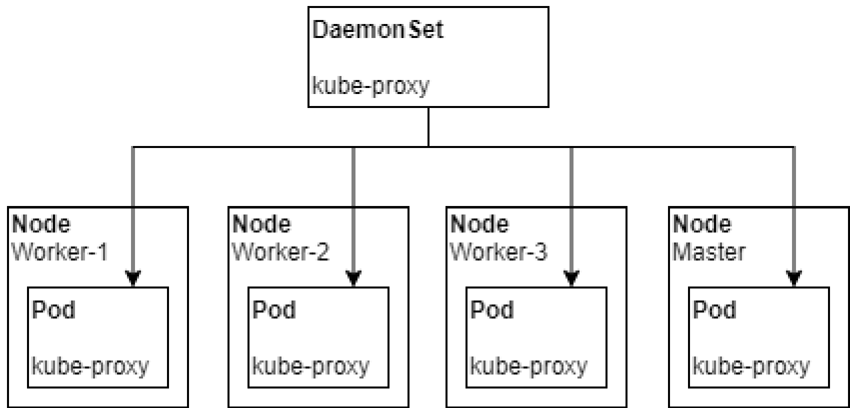
2.2.9 Secrets

Secrets er måten Kubernetes takler passord og andre ting man helst ikke vil ha lagret i klartekst, men man er nødt til å gi til forskjellige Pods. For eksempel innloggingsinfo til en database eller lignende. Som standard er ikke Secrets kryptert, kun base64-kodet, slik at de er vanskelig å lese, men lette å dekode[6]. Det anbefales å sette opp det Kubernetes kaller encryption at rest. Det vil si at Secrets er krypterte når de lagres til disk, men blir dekryptert når de sendes til for eksempel en Pod eller andre steder de skal brukes. Dette skal vanligvis ikke være et problem siden kommunikasjon internt i clusteret skal gå over HTTPS.

2.2.10 DaemonSet

Et DaemonSet er nesten nøyaktig det samme som et ReplicaSet, men istedenfor å passe på at du har et spesifikt antall Pods som kjører passer den på at det kjører en instans av Podden på hver av nodene i clusteret [9]. Dette betyr at du vil ha like mange Pods som du har noder. Dette er mye brukt av tjenester ment for interne oppgaver i Kubernetes, slik som

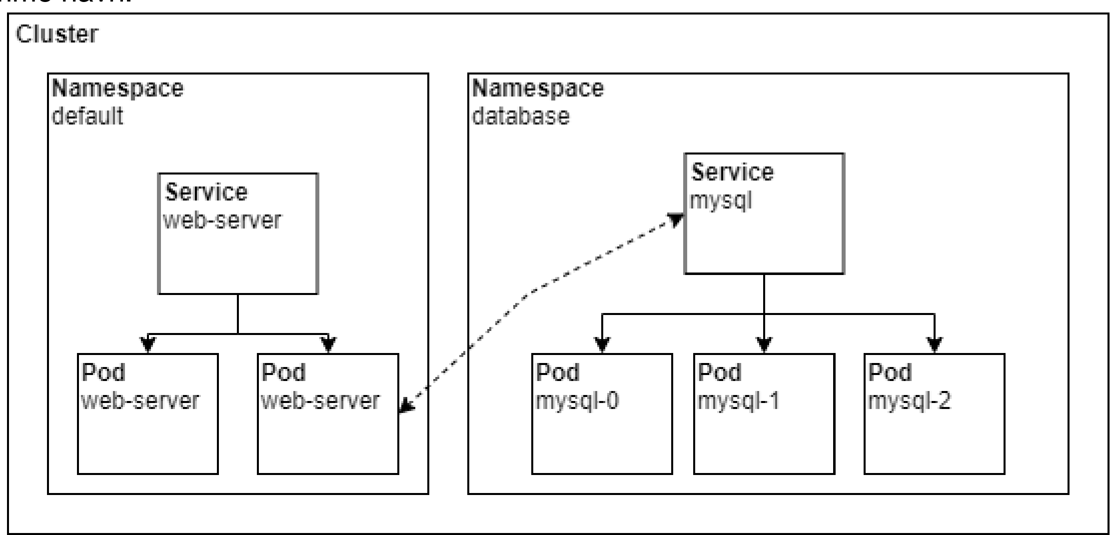
Pod-nettverket og kube-proxy, eller av tjenester som er nødt til å ha en oversikt over tilstanden til alle nodene for overvåkning eller lignende.



Figur 2.8: Et DaemonSet vil lage en Pod på hver node

2.2.11 Namespace

Så godt som alle ressurser tilhører et Namespace med et par unntak som for eksempel enkelte API-ressurser. Et Namespace er en måte å dele opp et cluster på, enten for å ha en bedre oversikt eller av sikkerhetshensyn. Om man har mange brukere kan man sette rettighetsbegrensninger på hvem som har lov til å se og endre ressurser i de forskjellige Namespacene. Man kan også bruke Namespace til å holde clusteret ryddig. Når man kjører for eksempel `kubectl get all` som viser en oversikt over diverse ressurser ser man bare ressurser i det Namespacet man er i for øyeblikket. Ved å legge forskjellige tjenester, spesielt de som har mange forskjellige typer ressurser og mange Deployments, i forskjellige Namespace er det lettere å holde oversikt. Ressurser i forskjellige Namespace kan også ha samme navn.



Figur 2.9: Ressurser i ulike Namespace kan fortsatt kommunisere med hverandre

2.2.12 Autentisering

Autentisering i Kubernetes bruker Role-based access control (RBAC)[10]. Dette er en metode for å begrense tilgang som baserer seg på bruk av roller og tildeling av rettigheter. Det er 5 objekter som inngår i autentiseringen; Role, ClusterRole, RoleBinding, ClusterRoleBinding og ServiceAccount.

2.2.12.1 Role og ClusterRole

Role og ClusterRole går begge ut på å gi brukere, grupper og ServiceAccounter tilgang til et sett med ressurser innad i clusteret. Det som skiller dem er at Role har et Namespace mens ClusterRole ikke har det. De er to separate objekter fordi et objekt i Kubernetes enten må ha et Namespace eller ikke ha det, og kan ikke være begge på en gang. Man bruker Role for å gi tilgang i et enkelt Namespace, og ClusterRole for å gi tilgang som går over flere Namespace, eller fra et Namespace til et annet. ClusterRole kan også gi rettigheter til ressurser som gjelder hele clusteret. Roles og ClusterRoles er bygget opp slik:

2_2_ressurser\role_eksempel.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Som alle andre ressursfiler har de apiVersion, kind og metadata. Eksempelet viser en Role, og vi må derfor ha med Namespace. Hvis vi i stedet ville lage en ClusterRole, bytter vi bare kind til ClusterRole og tar vekk Namespace. Det er rules-delen som bestemmer rettighetene til rollen. `apiGroups` bestemmer hvilken API-gruppe rollen tilhører. Her betyr `""` kjerne-gruppen, altså `api/v1`. De andre API-gruppene finnes under `apis/<gruppenavn>/<versjon>`, og de kan velges ved å skrive `apiGroups: <gruppnavn>/<versjon>`. De ulike API-gruppene har tilgang til ulike ressurser. `resources` bestemmer hva slags ressurser vi skal ha tilgang til, i dette tilfellet Poder. `verbs` bestemmer hva som kan gjøres med de ressursene man har tilgang til. `get`, `watch` og `list` betyr at man har tilgang til å hente data om Poder, overvåke en Pod og liste alle Poder.

2.2.12.2 RoleBinding og ClusterRoleBinding

For å binde roller til brukere og grupper brukes rolle-bindinger. Igjen har vi to forskjellige, en med Namespace og en uten. En RoleBinding kan referere til en Role i samme Namespace for å gi rettigheter i dette Namespacet, eller en ClusterRole for å binde ClusterRolen til et Namespace. ClusterRoleBinding binder en ClusterRole til alle Namespace i clusteret. Begge består av en liste med subjects som er en liste over alle brukerne eller gruppene rollen skal bindes til, og en roleRef som referer til rollen vi vil binde.

2_2_ressurser\rolebinding_eksempel.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
```

```
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

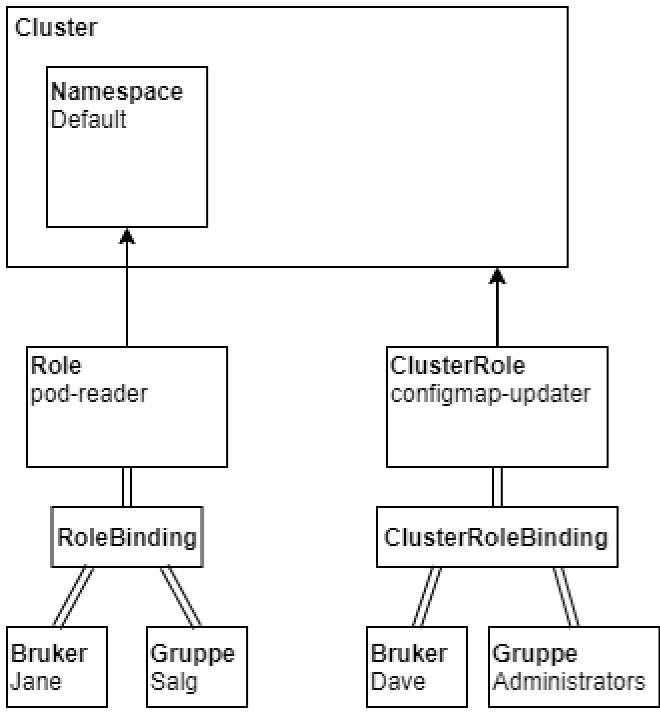
Her ser vi en RoleBinding som binder rollen `pod-reader` til brukeren `jane` i Namespacet `default`. Her må vi allerede ha laget en rolle `pod-reader` som ligger i samme Namespace. For å lage en ClusterRoleBinding bytter vi bare `kind:` til ClusterRoleBinding og tar bort `namespace`. Vi må også endre `kind:` under `roleRef:` til ClusterRole

2.2.12.3 ServiceAccount

En ServiceAccount er den tredje typen som kan brukes som `subject` i en rolle-binding. De brukes av Poder for å kommunisere med Kubernetes APIet, og opprettes som regel automatisk for de Podene som trenger det. Det hender likevel at man må lage en ServiceAccount manuelt for å gi en Pod de tilgangene den trenger. De lages enkelt med `kubectl create serviceaccount <navn>`. Vi kan sette Podene til å bruke denne ServiceAccounten ved å spesifisere den i Spec: feltet i en Pod-beskrivelse. For å gi ServIceAccounten rettigheter må den bindes til en rolle slik som vist tidligere:

2_2_ressurser\serviceaccount_eksempel.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
spec:
  replicas: 3
  template:
    metadata:
    # ...
    spec:
      serviceAccountName: my-serviceaccount
      containers:
      - name: nginx
        image: nginx:1.14.2
```



Figur 2.10: Oversikt over Role og ClusterRole

Bildet viser en RoleBinding som binder en bruker og en gruppe til en rolle i Default Namespace, og en ClusterRoleBinding som binder en annen bruker og en annen gruppe til en ClusterRole som har rettigheter i hele clusteret. Hvis du ønsker å se nærmere på hvordan Role, RoleBinding og Service Account brukes i praksis er de brukt i kapittel 6.3.2 om oppsett av Jenkins.

2.2.13 Custom Resources og Custom Controller

Kubernetes lar deg lage egendefinerte ressurser utover de som allerede er inkludert. Disse kan, sammen med en Custom Controller, gjøre alt som lar seg gjøre med API-kall til Kubernetes. En Custom Resource har ansvar for å lagre informasjon på en måte som både er leselig for mennesker og som en Custom Controller kan bruke til å gjøre API-kall. Det er også mulig å bruke en Custom Resource uten en Custom Controller, men da vil den fungere mer som et ConfigMap med spesialisert lagring av en type data. Custom Resource og Controller gjør Kubernetes mer modulært. Man kan lage eller laste ned egne ressurser for spesielle behov.

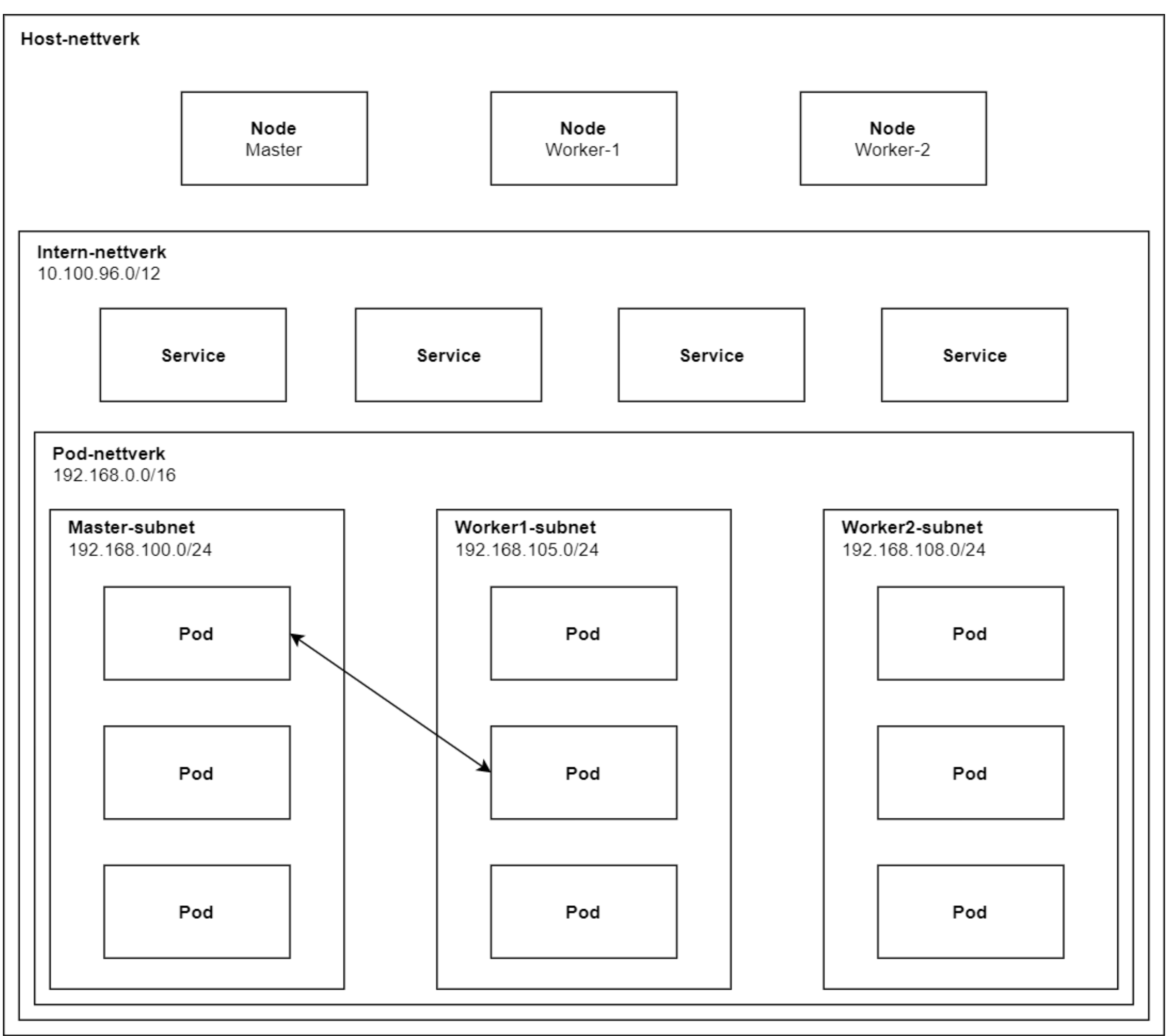
Mange tjenester ment for å forenkle administrering av Kubernetes legger ved egne Custom Resources og Controllere som kan automatisere diverse oppgaver. De er spesielt egnet for oppgaver der store deler av prosessen er den samme, men det er et par variabler som må endres avhengig av situasjon. Man kan for eksempel lage en Custom Resource som genererer TLS-sertifikater. Her er store deler av prosessen lik, men man er nødt til å oppgi host-navn og lignende avhengig av situasjon. En av fordelene med en Custom Resource er at den er mulig å bruke med kubectl. Du kan for eksempel kjøre `kubectl get <custom resource-navn>` for å få en oversikt over alle ressursene av den typen, de kan legges til

med en vanlig YAML-ressursfil ved å bruke `kubectl apply` og du kan få mer informasjon om dem med `kubectl describe`. Vi ser nærmere på hvordan en bruker en Custom Resource i kapittel 5.4 når vi setter opp cert-manager for automatisk tildeling av TLS-sertifikater.

2.3 Nettverk i Kubernetes

Et Kubernetes cluster bruker flere nettverkslag for å styre kommunikasjonen mellom de ulike tjenestene. Ytterst har vi host-nettverket. Dette er det vanlige nettverket maskinene dine er koblet til, og vil variere med hver enkelts nettverksoppsett. Når vi snakker om adressen til master eller en node er det dette nettverket vi snakker om. Clusteret har også et intern-nettverk, hvor hver Service får sin egen IP-adresse. Når vi snakker om ClusterIP er det som regel det interne nettverket det er snakk om. Dette nettverket settes opp automatisk under cluster-oppsettet, og bruker som regel adresseområdet 10.96.0.0/12.

Det siste nettverket vi har er Pod-nettverket. Det har som oppgave å styre kommunikasjon mellom Poder, og gjør det mulig for Poder på ulike noder å kommunisere. Dette gjøres ved at hver node får en del av Pod-nettverkets adresseområde til sine Poder. Da kan de kommunisere uten adresse-konflikter eller behov for ekstra ruting. Pod-nettverket må installeres som et tillegg, og det finnes flere ulike alternativer. Hvilke funksjoner det tilbyr og hvilket adresseområde det bruker kommer an på hvilket man velger. Vi vil bruke Calico, som er et av de mest brukte Pod-nettverkene, og det eneste som støttes og testes av Kubernetes teamet. Calico bruker adresseområdet 192.168.0.0/16 som standard.
Intern-nettverket og Pod-nettverket kan fritt kommunisere seg imellom, men de kan ikke nås fra utsiden. Når vi vil nå en tjeneste må vi gjøre Servicen til tjenesten tilgjengelig fra utsiden, og Servicen vil rute oss videre til riktig Pod



Figur 2.11: Kubernetes bruker flere nettverkslag. Intern- og Pod-nettverket kan kommunisere med hverandre, men kan ikke nås fra utsiden

3. Installasjon av Kubernetes

Det følgende oppsettet gjennomføres på alle maskinene som skal være en del av clusteret. Alle maskinene trenger den samme programvaren og det samme miljøet, mens fordeling av roller først skjer under oppsett av selve clusteret. Installasjonen i delkapittel 3.3 til 3.5 er gjort gjennom ett enkelt script som ligger vedlagt i `3_installasjon_av_kubernetes/docker-kube-install.sh`. Denne gjennomgangen går mer i detalj, og er ikke nøyaktig lik scriptet

3.1 Forhåndskrav

Maskinene som brukes må være Linux maskiner, våre kjører Debian. Windows maskiner kan også brukes som noder, men de kan ikke være master. Vi vil ikke gå gjennom bruk av Windows maskiner i dette dokumentet. Maskinene må i tillegg ha minimum ha 2 prosessorkjerner og 2 GB minne, men det anbefales å ha mer. [13][14] Våre maskiner har 4 prosessorkjerner og 16 GB minne.
Før installasjonen sørger vi for at maskinene som skal brukes har statiske IP-adresser og unike maskinnavn. Vi velger å oppdatere maskinnavnene til k8s-master, k8s-worker1, k8s-worker2 og k8s-worker3. Disse er beskrivende for maskinens rolle og er lette å huske senere. I tillegg installerer vi openssh-server på alle serverne. Dette er praktisk både under installasjon og ved senere bruk.

3.2 Skru av swap

Kubernetes kan ikke kjøre med swap skrudd på, og vi må derfor slå det av. For å gjøre dette kjører vi kommandoen:

```
sudo swapoff -a
```

Vi må også åpne filen `/etc/fstab` og kommentere ut en linje for å forhindre swap i å mountes ved oppstart. Linjen ser ut som følger:

```
UUID=4b229be9-b04d-47f2-80b0-e70e40a65d09 none   swap   sw    0       0
```

3.3 Dependencies

Før vi går videre installerer vi en rekke nødvendige pakker. Noen er nødvendige for Docker, noen for Kubernetes og noen for begge:

```
sudo apt install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg2 \
    software-properties-common \
```

```
    iptables \
    arptables \
    ebtables
```

For å unngå problemer på nyere OS versjoner må vi endre `iptables`, `arptables` og `ebtables` til å bruke legacy version:

```
sudo update-alternatives --set iptables /usr/sbin/iptables-legacy
sudo update-alternatives --set ip6tables /usr/sbin/ip6tables-legacy
sudo update-alternatives --set arptables /usr/sbin/arptables-legacy
sudo update-alternatives --set ebtables /usr/sbin/ebtables-legacy
```

3.4 Docker

Nå er alt klart, og vi kan gå i gang med selve installasjonen. Det første vi gjør er å installere Docker. Vi legger til og bekrefter Dockers offisielle GPG nøkkel:

```
sudo curl -fsSL https://download.docker.com/linux/debian/gpg |
    sudo apt-key add -
sudo apt-key fingerprint 0EBFCD88
```

For å finne de riktige pakkene må vi også sette opp Dockers apt-repository. Her kan man velge mellom `nightly`, `test` og `stable`. `nightly` og `test` er nyere versjoner som ofte er mer ustabile, så vi velger `stable`:

```
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/debian \
    $(lsb_release -cs) \
    stable"
```

Når repositoryet er satt opp, kan vi oppdatere apt og installere docker. Vi velger å installere spesifikke versjoner for å forhindre problemer ved uventede oppdateringer. De nyeste kompatible versjonene kan finnes på [Kubernetes sine sider]:

```
apt-get update && apt-get install -y \
  containerd.io=1.2.10-3 \
  docker-ce=5:19.03.4~3-0~debian-$(lsb_release -cs) \
  docker-ce-cli=5:19.03.4~3-0~debian-$(lsb_release -cs)
```

Følgende kommando konfigurerer logging og driver for Docker daemonen. Dette krever også at daemonen startes på nytt:

```
cat > /etc/docker/daemon.json <<EOF
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF
mkdir -p /etc/systemd/system/docker.service.d
sudo systemctl daemon-reload
sudo systemctl restart docker
```

Til slutt må alle brukerne som skal bruke Docker uten sudo tilgang legges til i `docker` gruppen:

```
sudo usermod -aG docker <brukernavn>
```

3.5 Kubernetes

Neste steg er å installere Kubernetes. Kubernetes har også en GPG nøkkel som vi legger til:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |
    sudo apt-key add -
```

Vi legger til Kubernetes repositoryet i filen `/etc/apt/sources.list.d/kubernetes.list`:

```
echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" |
    sudo tee -a /etc/apt/sources.list.d/kubernetes.list
```

Nå kan vi installere `kubectl`, `kubeadm` og `kubelet`. Versjonen låses, igjen for å forhindre uventede oppdateringer:

```
sudo apt update && sudo apt install -y kubectl kubeadm kubelet
sudo apt-mark hold kubelet kubeadm kubectl
```

4. Oppsett og konfigurering av Kubernetes cluster

Når alle maskinene har installert Docker og Kubernetes kan vi begynne å sette opp selve clusteret. Denne installasjonen skal kun gjøres på master-noden. Her setter vi opp selve clusteret og installerer noen tjenester vi mener er nødvendige for å ha et komplett bare-metal cluster. Igjen har vi gjort dette gjennom vedlagte scripts. De ulike scriptene er oppgitt i starten av de delkapitlene hvor de er brukt.

4.1 Oppsett av Kubernetes-cluster

Script ligger vedlagt under `4_1_kubernetes-cluster/kube-master.sh`

`kubeadm` brukes til å sette opp og konfigurere clusteret. Her settes også adresseområdet for Pod-nettverket. Dette kan være nesten hva som helst, men kommer som regel an på hvilket Pod-nettverk man bruker. Vi bruker Pod-nettverket Calico, som har standard adresseområde `192.168.0.0/16`. [15] Denne kommandoen genererer også en kommando for å melde noder inn i clusteret, som vi kopierer til senere:

```
sudo kubeadm init --pod-network-cidr=192.168.0.0/16
```

For at en vanlig bruker skal kunne bruke clusteret må konfigurasjonsfilene kopieres til brukerens hjemmemappe. Hvis man vil la flere brukere bruke clusteret, må dette gjøres for hver bruker:

```
mkdir -p $HOME/.kube
sudo cp /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Vi må også sette opp Calico, som er Pod-nettverket vårt. Flere steder bruker vi `curl -O` for å laste ned ressursfilen før vi legger den til. Når vi har ressursfilen lokalt er det enklere å se hva som installeres i tillegg til at det blir enklere å dokumentere eventuelle endringer som gjøres. URL-en kan brukes direkte i `kubectl apply` kommandoen:

```
curl -O https://docs.projectcalico.org/manifests/calico.yaml
kubectl apply -f calico.yaml
```

For å fullføre oppsettet av clusteret vil vi også sette opp Kubernetes Dashboard, som gir et grafisk grensesnitt mot clusteret:

```
curl -O https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-
beta8/aio/deploy/recommended.yaml
kubectl apply -f recommended.yaml
```

Kubernetes Dashboard trenger en ServiceAccount, som vi setter i filen `serviceaccount.yaml`:

4_1_kubernetes-cluster/serviceaccount.yaml

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
    name: admin-user
    namespace: kubernetes-dashboard
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
    name: admin-user
roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

Ressursen legges til med `kubectl apply -f serviceaccount.yaml`

4.2 Melde inn noder

For å melde nodene inn i clusteret brukes et token. Kommandoen vi kopierte tidligere inneholder allerede dette tokenet, og er klar til bruk. IP-adressen vi bruker her vil være IPen til master noden etterfulgt av portnummeret `6443`, som er porten Kubernetes bruker. Vår maskin har adressen `10.100.39.23`.

```
kubeadm join 10.100.39.23:6443 --token <token> \
      --discovery-token-ca-cert-hash <hash>
```

Kommandoen kjøres som root på hver av nodene. Når dette er gjort kan vi liste ut alle nodene på masteren:

```
simen@k8s-master:~$ kubectl get nodes
NAME          STATUS    ROLES    AGE   VERSION
k8s-master    Ready     master   13d   v1.17.3
k8s-worker1   Ready     <none>   13d   v1.17.3
k8s-worker2   Ready     <none>   13d   v1.17.3
k8s-worker3   Ready     <none>   13d   v1.17.3
```

4.3 Oppsett av MetalLB

Ingress Controlleren trenger en ekstern IP-adresse, noe som i en skyplattform vil håndteres av skyleverandøren. Siden vi har en bare-metal installasjon trenger vi en egen tjeneste for dette, slik som MetalLB. For å sette opp MetalLB trenger vi kun laste ned og legge til en ressursfil:

```
curl -O
https://raw.githubusercontent.com/google/metallb/v0.8.3/manifests/metall
b.yaml
kubectl apply -f metallb.yaml
```

Vi gir MetalLB et sett med IP-adresser gjennom et ConfigMap i filen `metallb-configmap.yaml`:

4_3_metallb/metallb-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 10.100.39.35-10.100.39.40
```

Her får MetalLB adresseområdet 10.100.39.35-10.100.39.40 som kan deles ut til Servicer i clusteret. Vi legger til ressursen med `kubectl apply -f metallb-configmap.yaml`

4.4 Oppsett Ingress-Controller

Script ligger vedlagt under *`4_4_ingress-controller/ingress-controller.sh`*

Når MetalLB er satt opp kan vi fortsette til Ingress Controlleren. Den trenger to ressursfiler, en for selve Poden og kjernefunksjonaliteten og en for den Service typen vi vil ha:

```
curl -O https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-
0.30.0/deploy/static/mandatory.yaml
kubectl apply -f mandatory.yaml
```

```
curl -O https://raw.githubusercontent.com/kubernetes/ingress-
nginx/nginx-0.30.0/deploy/static/provider/cloud-generic.yaml
kubectl apply -f cloud-generic.yaml
```

For å se at Ingress Controlleren er satt opp riktig kan vi liste Servicen vi satt opp. Under `EXTERNAL-IP` skal vi se at den har blitt tildelt en av adressene vi tildelte MetalLB. Det kan også stå `<pending>`. Da kjører vi bare kommandoen igjen helt til den har fått en IP-adresse:

```
simen@k8s-master:~$ kubectl get svc -n ingress-nginx
NAME            TYPE           CLUSTER-IP    EXTERNAL-IP
ingress-nginx   LoadBalancer   10.96.38.3    10.100.39.35
```

4.5 Metrics Server

Script ligger vedlagt under *`4_5_metrics-server/metrics-server.sh`*

Metrics Server er ansvarlig for å samle inn data om Poder og noders ressursbruk, som brukes av auto-skalerere for å rulle ut eller ta ned Pods. Vi bruker git for å hente filene vi trenger:

```
git clone https://github.com/kubernetes-sigs/metrics-server.git
```

Kubectl kan brukes for å legge til alle ressursfilene i en mappe på en gang

```
kubectl apply -f metrics-server/deploy/kubernetes
```

For at metrics serveren skal fungere må vi gjøre en endring i Deployment ressursen dens. Med `kubectl edit` kan vi se en ressurs i yaml format. Dette er ikke en faktisk fil, bare en representasjon av ressursen. Vi bruker følgende kommando:

```
kubectl edit deployment metrics-server -n kube-system
```

Ressursen vil åpnes i en editor:

```
template:
  metadata:
    creationTimestamp: null
    labels:
      k8s-app: metrics-server
    name: metrics-server
  spec:
    containers:
    - args:
      - --cert-dir=/tmp
      - --secure-port=4443
      command:
      - /metrics-server
```

```
      - --kubelet-preferred-address-types=InternalIP,Hostname,ExternalIP
      - --kubelet-insecure-tls
      image: k8s.gcr.io/metrics-server-amd64:v0.3.6
      imagePullPolicy: IfNotPresent
      name: metrics-server
      ports:
      - containerPort: 4443
        name: main-port
        protocol: TCP
```

Endringen vi må gjøre er å legge inn `command:` seksjonen slik som vist over. Dette vil endre hvordan metrics-serveren kommuniserer med Kubelet. Når vi lagrer filen vil endringene bli utført automatisk.

5 Verktøy for administrering av Kubernetes cluster

Etter hvert som et Kubernetes cluster vokser vil det bli flere og flere ting som trenger administrering og overvåkning, og det har blitt utviklet flere verktøy og tjenester for å gjøre nettopp dette enklere. Dette kapittelet tar for seg installasjon og bruk av noen av de mest populære Kubernetes-verktøyene.

5.1 Helm

Helm er et verktøy for å raskt deploye alt du trenger for å kjøre en tjeneste og fungerer litt som et pakkesystem for Kubernetes. Det forenkler utrullingen av tjenester ved å ha en kommando for utrulling av alle ressurser en tjeneste trenger og en samling av ferdigskrevne ressurser med standardverdier.

5.1.1 Installasjon av Helm

Den enkleste måten å installere Helm til Linux på er å bruke en binærfil. For å gjøre dette må vi først laste ned ønsket versjon. For å laste ned versjon 3.1.2 som er siste versjon kjører vi `wget https://get.helm.sh/helm-v3.1.2-linux-amd64.tar.gz`. Denne filen er pakket inn som en tarball, for å pakke den ut kjører vi `tar -zxvf helm-v3.1.2-linux-amd64.tar.gz`. Så må vi legge binærfilen et sted hvor den kan kjøres fra. Dette gjøres med kommandoen `mv linux-amd64/helm /usr/local/bin/helm`.

5.1.2 Charts og repositories

For å deploye en tjeneste bruker man det Helm kaller charts. Et chart er en samling av alle ressursfilene man trenger for å deploye tjenesten. Disse ressursfilene er laget på en sånn måte at man kan endre på konfigurasjonen uten å måtte endre på en og en ressursfil eller laste ned hele chartet. Vi ser nærmere på hvordan dette fungerer senere.

Som standard kommer Helm uten noen repositorier. Et repository er en samling med charts som Helm leter gjennom. Man burde kun legge til repositorier fra kilder man stoler på. Vi kan legge til Helms stable repository ved å kjøre `helm repo add stable`. Stable er en kurert samling av charts som vedlikeholdes av Helm. Man kan finne charts og repositorier ved å gå til https://hub.helm.sh.

5.1.3 Utrulling av charts

Å rulle ut et chart med Helm er veldig enkelt. Kommandoen man bruker ser slik ut: `helm install <navn på utrulling> <repo>/<chart>`. Hvis vi for eksempel ønsker å rulle ut en nginx webtjener trenger vi bare å legge til et repo med et chart for nginx som for eksempel Bitnami med `helm repo add bitnami https://charts.bitnami.com/bitnami` for så å installere med `helm install nginx bitnami/nginx`. Hvis vi nå kjører `helm list` kan vi se blant annet status for utrulling og applikasjonsversjon.

```
magnus@k8s-master:/opt/kubernetes/mysql$ helm list
NAME    NAMESPACE       REVISION        UPDATED                                 STATUS          CHART           APP VERSION
nginx   helm            1               2020-03-25 16:04:21.159419101 +0100 CET deployed        nginx-5.1.11    1.16.1
```

Hvis vi kjører `kubectl get all` ser vi at det har blitt laget en Deployment med tilhørende ReplicaSet og Pod og en Service for nginx.

```
magnus@k8s-master:/opt/kubernetes/mysql$ kubectl get all
NAME                            READY   STATUS    RESTARTS   AGE
pod/nginx-66c9bdbbdb-98624      1/1     Running   0          3m42s

NAME            TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)                      AGE
service/nginx   LoadBalancer   10.105.54.5    10.100.39.37    80:30720/TCP,443:30526/TCP   3m42s

NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx   1/1     1            1           3m42s

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-66c9bdbbdb   1         1         1       3m42s
```

5.1.4 Konfigurering av charts

Selv om Helm bruker helt vanlige ressursfiler, er de satt opp på en annerledes måte. For at du skal slippe å laste ned et chart med mange forskjellige ressursfiler og mange verdier som må endres til det samme er ressursfilene satt opp for å ta imot variabler som ligger i en annen fil og som kan endres fra kommandolinjen under installasjon.

5_1_helm\helm_chart_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ template "openvpn.fullname" . }}
  labels:
    app: {{ template "openvpn.name" . }}
    chart: {{ template "openvpn.chart" . }}
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
spec:
  replicas: {{ .Values.replicaCount }}
{{- if .Values.updateStrategy }}
  strategy:
{{ toYaml .Values.updateStrategy | indent 4 }}
{{- end }}
```

```
  selector:
    matchLabels:
      app: {{ template "openvpn.name" . }}
      release: {{ .Release.Name }}
```

Dette er et utdrag fra en typisk Deployment-fil i et openvpn chart. Vi ser at det ikke blir fylt ut noe informasjon om oppsettet i selve filen. Det brukes i stedet variabler som hentes fra andre steder. Vi ser det hentes fra ".Values". Her henter den fra filen values.yaml som inneholder standardverdier det er ment skal kunne endres på. Det hentes også fra ".Release". Dette er informasjon om selve denne versjonen av chartet som versjonsnummer, navn og lignende og det hentes fra "template" som er generell informasjon om chartet og komponentene i chartet.

5_1_helm\helm_chart_values.yaml

```
# Default values for openvpn.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.
replicaCount: 1

updateStrategy: {}
  # type: RollingUpdate
  # rollingUpdate:
  #   maxSurge: 1
  #   maxUnavailable: 0

image:
  repository: jfelten/openvpn-docker
  tag: 1.1.0
  pullPolicy: IfNotPresent
```

Dette er et utdrag fra values-filen. Her ser vi at det settes verdier for blant annet hvilket image som skal brukes, hvor mange replicas som skal lages og hvordan den skal takle oppdateringer. Dette er standardverdier som kan endres på.

Det er tre måter å endre disse variablene på. Den første og mest tungvinte er å laste ned hele chartet og endre på standardverdiene i values-filen for så å pakke inn chartet på nytt. Dette gjøres ved å først laste ned chartet ved å kjøre `helm pull --untar <repository>/<chart>` for så å gå inn i values-filen og endre verdiene der. Her kan man også endre på selve ressursfilene. Når man har gjort endringene er man nødt til å pakke inn igjen chartet ved å kjøre `helm package <mappe med chart>`. Denne metoden er spesielt nyttig om du må endre på noe som ikke er gitt en verdi i values som for eksempel apiVersion.

De to andre måtene er ved å sette flagg under installasjon. Med flagget `--set` kan man overskrive verdier i selve kommandoen. Hvis vi vil øke antall replicas og sette den til å hente det nyeste imaget til openvpn chartet i eksempelet over kan det for eksempel se slik ut: `helm`

`install openvpn stable/openvpn --set replicaCount=2,image.tag=latest`. Man bruker komma for å legge til flere felter man vil overskrive. Legg også merke til at man bruker likhetstegn for å sette en verdi istedenfor kolon som brukes i YAML.

Om man har veldig mange verdier man vil overskrive kan man lage en egen fil og legge denne inn i kommandoen når man installerer med flagget `--values`. Denne filen skal være formatert som en YAML-fil og ha samme oppsettet som values.yaml. Hvis vi vil gjøre samme endringer som vi gjorde med `--set` kan vi bruke kommandoen `helm install openvpn stable/openvpn --values mineverdier.yaml` og mineverdier.yaml vil da se slik ut:

5_1_helm\mineverdier.yaml

```
replicaCount: 3
image:
  tag: latest
```

5.2 Operators

Å sette opp en stateful applikasjon i Kubernetes kan fort bli en stor og kompleks oppgave. I database eksempelet senere ser vi at det er behov for et StatefulSet, flere Servicer, init-containere osv. I en større applikasjon som er ment for produksjon blir dette fort en stor og tungvint jobb å sette opp og følge med på. Som en løsning på dette finnes Kubernetes Operators.

5.2.1 Hva er Operators

En Operator er programvare som er skrevet for å pakke inn en kompleks applikasjon og integrere den inn i Kubernetes rammeverket. [18] Alt av konfigurering, utrulling, oppgradering, feilhåndtering osv. håndteres av Operatoren, og påkalles automatisk når det trengs. Hver Operator skrives for en bestemt applikasjon, og forbindes med en eller flere egendefinerte ressurstyper. De fungerer som en utvidelse til Kubernetes APIet, og lytter etter alt som har med sin ressurstype å gjøre. Operators kan bli store og komplekse applikasjoner i seg selv, og er ment for å skrives én gang for å gjøre jobben triviell ved senere bruk. Mange populære tjenester slik som PostgreSQL, MySQL og Prometheus har Operators for enkel og effektiv bruk av sine applikasjoner.

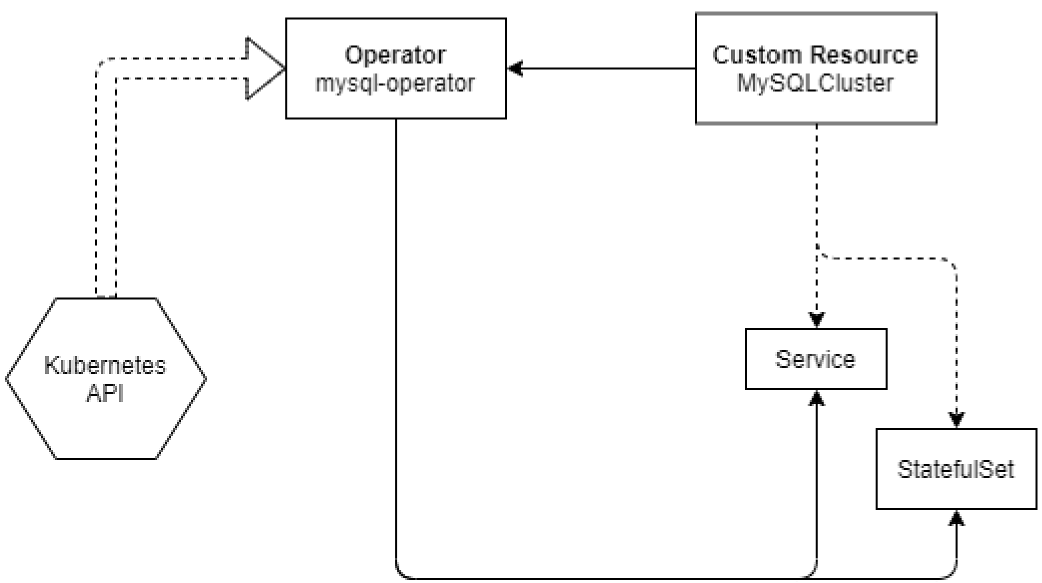
Operators ligner på mange måter på Helm Charts, men det er noen viktige forskjeller. Helm henter og ruller ut alle de nødvendige ressursene for en applikasjon, men har ikke, og kan heller ikke håndtere, informasjon om hvordan disse fungerer sammen for å lage en fungerende applikasjon. Operators brukes til stateful applikasjoner, som har spesifikke krav til hvordan de skal starte, skalere og avslutte. En Operator har denne informasjonen, og kan sette ressursene i sammenheng for å drifte applikasjonen best mulig. Helm og Operators er altså ikke alternativer til hverandre, men brukes ofte sammen. Mange Helm Charts inkluderer en Operator som tar seg av oppsett og drifting av applikasjonen.

5.2.2 Operators i praksis

Som et eksempel ser vi for oss at vi har i oppgave å sette opp flere MySQL databaser. Det vil være relativt små forskjeller mellom hver database, slik som antall Poder, lagringsplass og databaseversjon. Likevel blir vi nødt til å sette opp StatefulSet, Servicer og alt som følger med for hver av dem, selv om de er nærmest identiske. Vi kan tenke oss at disse kunne vært en egen ressurstype kalt MySQLCluster [19]:

```yaml
apiVersion: cr.mysqloperator.grtl.github.com/v1
kind: MySQLCluster
metadata:
  name: "my-cluster"
spec:
  secret: "my-secret"
  fromBackup: "my-backup-2017-12-14-01-22"
  port: 3306
  replicas: 2
  storage: "1Gi"
  image: "mysql:latest"
```

MySQLCluster er ikke noen vanlig ressurstype i Kubernetes, og Kubernetes vet derfor ikke hva som skal gjøres med denne ressursen. For at den skal kunne tas i bruk må vi rulle ut en Operator som utvider Kubernetes APIet. Ved å kjøre `kubectl run mysql-operator --image=grtl/mysql-operator:latest` kan vi rulle ut MySQL Operatoren fra GRTL, som er satt opp for nettopp denne ressursen. Når vi aktiverer definisjonen ovenfor vil Operatoren fange opp dette og lage en MySQLCluster ressurs. Denne vil igjen lage et StatefulSet, Service og alt annet som trengs for at databasen skal fungere.



Figur 5.1: Operatoren fungerer som en utvidelse av Kubernetes APIet, og håndterer en bestemt Custom Resource

5.3 Monitorering

Monitorering av clusteret kan være en kjekk ting å sette opp. Det kan gi deg et kjapt overblikk over hvor mye ressurser som blir brukt av clusteret og Pods og gi deg varsler når for eksempel en node eller Pod bruker veldig mye ressurser. For å gjøre dette trenger vi to ting; en scraper for å hente ut data fra kubernetes og en måte å visualisere dataene på. Til dette velger vi å bruke Prometheus og Grafana, men det er mange alternativ. Vi valgte Prometheus og Grafana siden de ser ut til å være veldig utbredt og har god støtte i tillegg til at de er open source. Siden vi ønsker å lagre statistikken vi henter inn krever dette at det er satt opp en Storage Provisioner eller en annen type varig lagring som vist i kapittel 6.2.1 Storage Provisioner.

5.3.1 Prometheus

Prometheus er en tjeneste som henter inn data om clusteret og lagrer det i en database sammen med et tidsstempel. Dette kan være alt fra hvor mye CPU en node bruker til hvor mange Pods som kjører. Du kan da bruke disse dataene til å lage grafer og tabeller for å lett få et overblikk over hvordan tilstanden til clusteret er. Prometheus inneholder også det de kaller en alertmanager. Den lar deg sette opp varsler for å få beskjed når for eksempel ressursbruken til en Pod eller node går over en satt grense. For å hente ut data og sette opp varslinger bruker Prometheus sitt eget språk, PromQL. Det er relativt likt SQL i struktur og syntaks, men er spesialtilpasset til å brukes i databaser med tidsstempel.

5.3.2 Grafana

Grafana er et verktøy for å visualisere data. I samarbeid med Prometheus har du mulighet til å lage dashboards som kan gi deg et kjapt overblikk over tilstanden til clusteret. Grafana er ekstremt konfigurerbart noe som både er bra og dårlig. Det tar litt ekstra tid å få satt opp ting akkurat som du vil, men du er sjeldent begrenset av hva du kan gjøre.

5.3.3 Installasjon

Det er flere måter å installere Prometheus og Grafana på. Vi har valgt å installere ved hjelp av Helm for å gjøre det enklest mulig. [16] Vi er først nødt til å definere noen standardverdier som skal overskrives med våre egne. Siden det er ganske mange er det lettest å lage en fil og sende denne til installasjons kommandoen. Vi kan se alle verdiene vi kan overskrive ved å kjøre `helm show values stable/prometheus-operator`. Denne er ekstremt lang så det er mulig toppen blir kuttet av, derfor laster vi den ned til en fil ved å kjøre `helm show values stable/prometheus-operator > default.yaml`. Nå kan vi også søke i denne filen som gjør den lettere å arbeide med. Det første vi må endre på er hvilken storageClass som skal brukes for å lage et Persistent Volume. Dette må endres både for Prometheus og AlertManager. Vi lager en ny fil som vi kaller promValues.yaml og kopierer inn storage-innstillingene for både prometheus og alertmanager fra default.yaml. De ser slik ut

5_3_monitorering\default.yaml

```
    storage: {}
    # volumeClaimTemplate:
    #   spec:
    #     storageClassName: gluster
    #     accessModes: ["ReadWriteOnce"]
    #     resources:
    #       requests:
    #         storage: 50Gi
    #   selector: {}
```

Vi fjerner kommentartegnene og legger til overobjektene så treet blir riktig. Vi må også fjerne {} fra `storage:` og `selector: {}`. I tillegg setter vi `storageClassName:` til Storage Provisioneren vår. Siden vi har begrenset med lagring på filtjeneren vår skrur vi også ned lagringskapasiteten fra 50 til 10 GiB. Det er viktig å passe på at det blir riktig antall mellomrom. Noen ganger, spesielt når man kopierer eller fjerner kommentarer, kan det bli fjernet mellomrom og siden YAML baserer objektstrukturen på mellomrom er det viktig å passe på. Det kan heller ikke brukes tabuleringer. Noen tekstprogram som Notepad++ erstatter fire mellomrom med en tabulering automatisk så dette må endres tilbake. Da promValues.yaml se slik ut.

5_3_monitorering\promValues.yaml

```
prometheus:
  prometheusSpec:
    storageSpec:
      volumeClaimTemplate:
        spec:
          storageClassName: nfs-storage
          accessModes: ["ReadWriteOnce"]
          resources:
            requests:
              storage: 10Gi

alertmanager:
  alertmanagerSpec:
    storage:
      volumeClaimTemplate:
        spec:
          storageClassName: nfs-storage
          accessModes: ["ReadWriteOnce"]
          resources:
            requests:
              storage: 10Gi
```

Det neste vi må gjøre er å legge inn innstillingene for Ingress. Dette må settes opp for Grafana, Prometheus og alertManager. Vi kopier Ingressoppsettet fra default.yaml og legger det inn i promValues.yaml. Ingressoppsettet ser slik ut i default.yaml:

5_3_monitorering\default.yaml

```
  ingress:
    enabled: false
    annotations: {}
    labels: {}

    ## Hostnames.
    ## Must be provided if Ingress is enabled.
    ##
    # hosts:
    #   - prometheus.domain.com
    hosts: []

    ## Paths to use for ingress rules - one path should match the
prometheusSpec.routePrefix
    ##
    paths: []
    # - /

    ## TLS configuration for Prometheus Ingress
    ## Secret must be manually created in the namespace
    ##
    tls: []
      # - secretName: prometheus-general-tls
      #   hosts:
      #     - prometheus.example.com
```

Vi setter `enabled:` til `true` og siden vi ikke bruker TLS så kan den biten kan fjernes. Vi legger til hostnamet vårt i firkantparantesen til hosts. Siden vi lager et nytt subdomene for hver tjeneste legger vi kun til / i path. Standardverdien for routePrefix er allerede / så vi trenger ikke gjøre noe der. Dette må gjøres på alle tjenestene vi ønsker å ha tilgjengelig fra utsiden og noen av dem ser litt annerledes ut. Grafana har som standardverdi at det prøver å lage et TLS-sertifikat med ACME, dette må endres til false. promValues.yaml ser nå slik ut:

5_3_monitorering\promValues.yaml

```
prometheus:
  prometheusSpec:
    storageSpec:
      volumeClaimTemplate:
        spec:
          storageClassName: nfs-storage
          accessModes: ["ReadWriteOnce"]
```

```
          resources:
            requests:
              storage: 10Gi
  ingress:
    enabled: true
    annotations: {}
    labels: {}
    hosts: [prometheus.example.com]
    paths: [/]

alertmanager:
  alertmanagerSpec:
    storage:
      volumeClaimTemplate:
        spec:
          storageClassName: nfs-storage
          accessModes: ["ReadWriteOnce"]
          resources:
            requests:
              storage: 10Gi
  ingress:
    enabled: true
    annotations: {}
    labels: {}
    hosts: [alertmanager.example.com]
    paths: [/]

grafana:
  ingress:
    enabled: true
    annotations:
      kubernetes.io/ingress.class: nginx
      kubernetes.io/tls-acme: "false"
    labels: {}
    hosts: [grafana.example.com]
    path: /
```

Det var alle standardverdiene vi var nødt til å endre. Nå er det bare å kjøre `helm install monitor stable/prometheus-operator --values promValues.yaml`. På grunn av en bug i enkelte versjoner av Helm er det mulig du får errormeldingene:

```
manifest_sorter.go:192: info: skipping unknown hook: "crd-install"
manifest_sorter.go:192: info: skipping unknown hook: "crd-install"
manifest_sorter.go:192: info: skipping unknown hook: "crd-install"
```

```
manifest_sorter.go:192: info: skipping unknown hook: "crd-install"
manifest_sorter.go:192: info: skipping unknown hook: "crd-install"
manifest_sorter.go:192: info: skipping unknown hook: "crd-install"
```

Dette betyr at Helm ikke klarte å installere en Custom Resource Definition. For å fikse dette må du avinstallere chartet igjen med `helm uninstall monitor` og legge til definisjonene selv med

```
kubectl apply -f https://raw.githubusercontent.com/coreos/prometheus-operator/release-0.37/example/prometheus-operator-crd/monitoring.coreos.com_alertmanagers.yaml

kubectl apply -f https://raw.githubusercontent.com/coreos/prometheus-operator/release-0.37/example/prometheus-operator-crd/monitoring.coreos.com_podmonitors.yaml

kubectl apply -f https://raw.githubusercontent.com/coreos/prometheus-operator/release-0.37/example/prometheus-operator-crd/monitoring.coreos.com_prometheuses.yaml

kubectl apply -f https://raw.githubusercontent.com/coreos/prometheus-operator/release-0.37/example/prometheus-operator-crd/monitoring.coreos.com_prometheusrules.yaml

kubectl apply -f https://raw.githubusercontent.com/coreos/prometheus-operator/release-0.37/example/prometheus-operator-crd/monitoring.coreos.com_servicemonitors.yaml

kubectl apply -f https://raw.githubusercontent.com/coreos/prometheus-operator/release-0.37/example/prometheus-operator-crd/monitoring.coreos.com_thanosrulers.yaml
```

Før vi installerer på nytt er vi nødt til å si til Prometheus at den ikke skal prøve å lage disse definisjonene ved å legge til disse linjene i promValues.yaml

```
prometheusOperator:
  createCustomResource: false
```

Vi kan nå kjøre installasjonen igjen med `helm install monitor stable/prometheus-operator --values promValues.yaml`.

Hvis vi kjører `kubectl get all` nå kan vi se at Helm har rullet ut tre Deploymenter, to StatefulSets og ett DaemonSet.

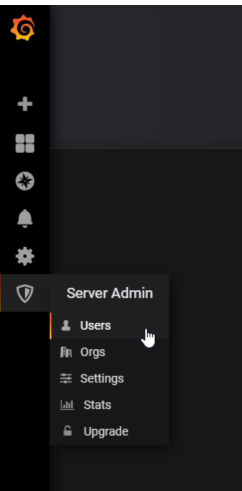
```
NAME                                                        READY   STATUS    RESTARTS   AGE
pod/alertmanager-monitor-prometheus-operato-alertmanager-0   2/2     Running   0          21s
pod/monitor-grafana-7f4ddbff8c-lhqt6                         3/3     Running   0          26s
pod/monitor-kube-state-metrics-756787548f-m986d              1/1     Running   0          26s
pod/monitor-prometheus-node-exporter-7gjkh                   1/1     Running   0          26s
pod/monitor-prometheus-node-exporter-9p9gh                   1/1     Running   0          26s
pod/monitor-prometheus-node-exporter-stjgk                   1/1     Running   0          26s
pod/monitor-prometheus-node-exporter-v2hvv                   1/1     Running   0          26s
pod/monitor-prometheus-operato-operator-69b95c8b-xfq22       2/2     Running   0          26s
pod/prometheus-monitor-prometheus-operato-prometheus-0       3/3     Running   0          11s

NAME                                              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
service/alertmanager-operated                     ClusterIP   None             <none>        9093/TCP,9094/TCP,9094/UDP   21s
service/monitor-grafana                           ClusterIP   10.100.55.51     <none>        80/TCP                       26s
service/monitor-kube-state-metrics                ClusterIP   10.104.182.32    <none>        8080/TCP                     26s
service/monitor-prometheus-node-exporter          ClusterIP   10.111.158.181   <none>        9100/TCP                     26s
service/monitor-prometheus-operato-alertmanager   ClusterIP   10.108.165.215   <none>        9093/TCP                     26s
service/monitor-prometheus-operato-operator       ClusterIP   10.103.229.155   <none>        8080/TCP,443/TCP             26s
service/monitor-prometheus-operato-prometheus     ClusterIP   10.102.142.10    <none>        9090/TCP                     26s
service/prometheus-operated                       ClusterIP   None             <none>        9090/TCP                     11s

NAME                                              DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
daemonset.apps/monitor-prometheus-node-exporter   4         4         4       4            4           <none>          26s

NAME                                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/monitor-grafana                       1/1     1            1           26s
deployment.apps/monitor-kube-state-metrics            1/1     1            1           26s
deployment.apps/monitor-prometheus-operato-operator   1/1     1            1           26s

NAME                                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/monitor-grafana-7f4ddbff8c                  1         1         1       26s
replicaset.apps/monitor-kube-state-metrics-756787548f       1         1         1       26s
replicaset.apps/monitor-prometheus-operato-operator-69b95c8b   1      1         1       26s

NAME                                                                  READY   AGE
statefulset.apps/alertmanager-monitor-prometheus-operato-alertmanager   1/1     21s
statefulset.apps/prometheus-monitor-prometheus-operato-prometheus       1/1     11s
```

Hvis DNSen din er satt opp til å sende de hostnavnene vi satt opp til Ingress Controlleren kan du nå nå de med en nettleser. Alertmanager og prometheus har som standard ikke passord, men på Grafana må du bruke brukernavn: admin og passord: prom-operator. Standardpassordet er mulig å endre på under installasjon ved å legge inn `adminPassword: <passord>` under `grafana:` i promValues.yaml

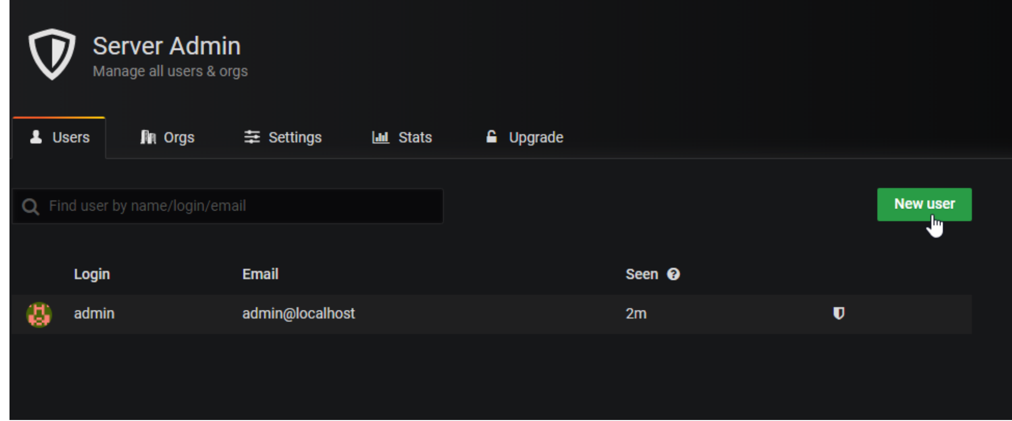
5.3.4 Konfigurasjon

5.3.4.1 Deaktivere admin-bruker

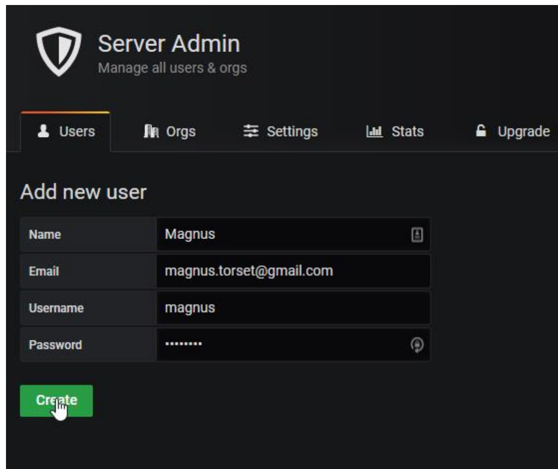
Det første vi burde gjøre er å deaktivere administratorbrukeren i Grafana og lage en ny bruker til oss selv og eventuelt andre som skal ha tilgang. Dette gjøres ved gå til grafana.example.com og logge inn med administratorbrukeren. Deretter går vi til “Server Admin” i menyen til venstre og velger “Users”.



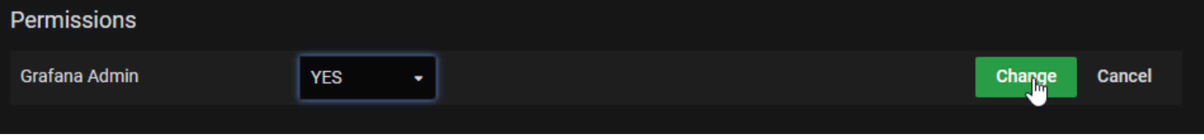
Legg til en ny bruker ved å klikke på “New user”.



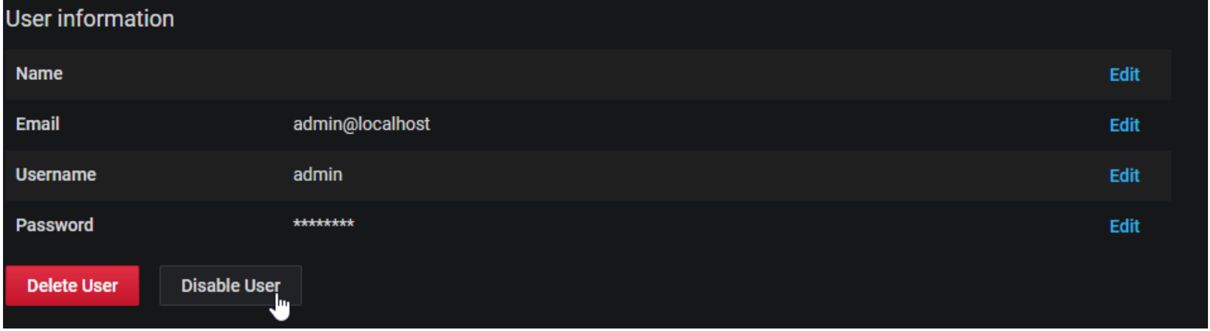
Fyll inn navn, epost, brukernavn og passord og klikk “Create”.



For å gi brukeren administratorrettigheter klikker du på brukeren, går til “Permissions”, “Grafana Admin” og klikker “Change”. Endre fra “NO” til “YES” i dropdown-menyen og klikk “Change”.

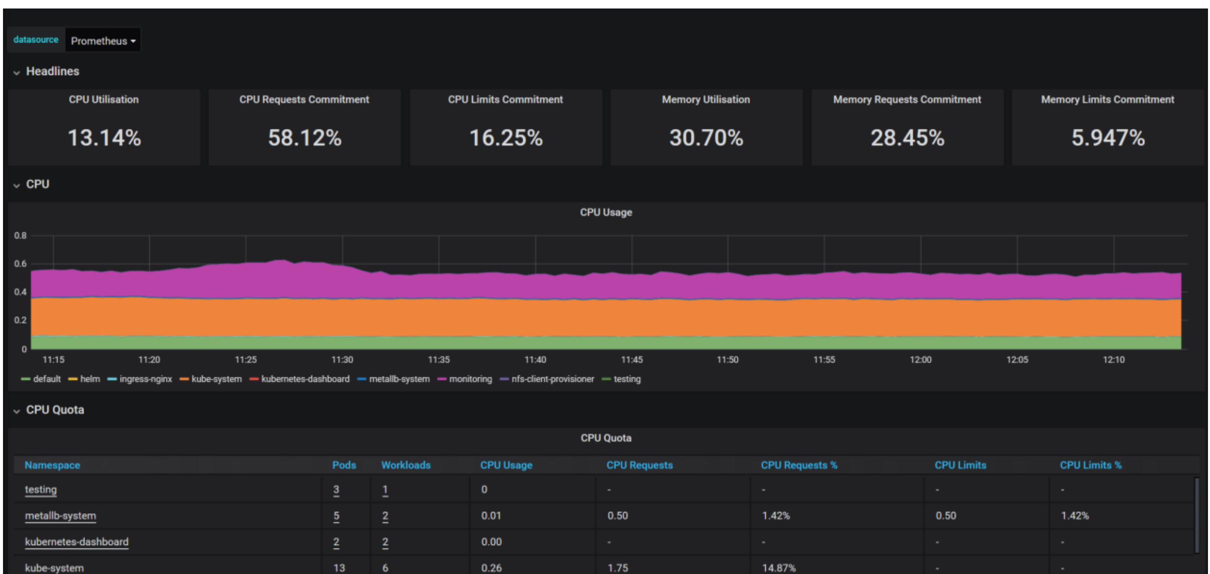


For å deaktivere administratoren klikker vi på admin i listen over brukere og velger “Disable User”. Du kan nå logge inn med brukeren du lagde.



5.3.4.2 Grafana Dashbord

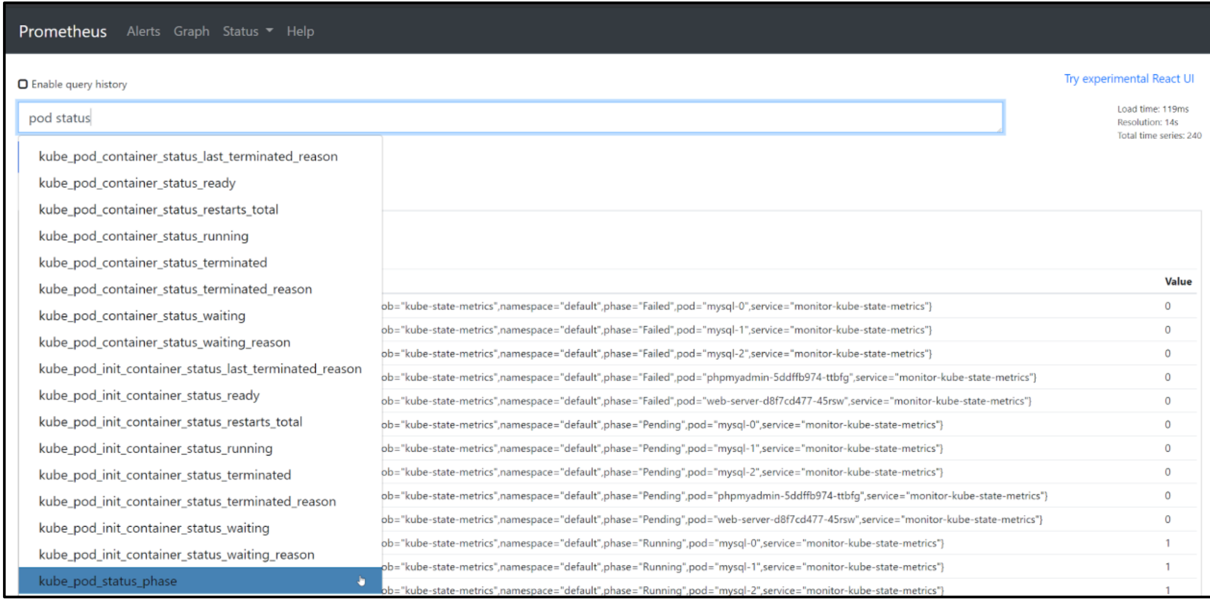
Når du installerer Grafana gjennom det Helm-chartet vi gjorde kommer det med noen ferdiglagde dashbord. Disse kan være nyttige og er ofte gode utgangspunkt for å lage egne dashbord, men det beste er jo selvfølgelig å lage dashbord som er tilpasset ditt oppsett med den informasjonen du trenger å vite.



Et eksempel på ett av de innebygde dashbordene.

Vi ønsker å lage et nytt dashbord med paneler som viser diverse informasjon om Podder som hvor mange som kjører, om det er noen som ikke kjører, ressursbruk for hele clusteret og ressursbruk for enkelte Deployments.

For å teste ut og finne spørringer bruker vi Prometheus sin web app. Med vårt oppsett finnes den på prometheus.example.com. Ved å begynne å skrive i query-feltet får man opp forslag til spørringer basert på den informasjonen Prometheus har tilgjengelig. I vårt tilfelle vil vi vite status til de Podene vi har og hvor mange Poder det er totalt. Derfor begynner vi med å skrive “Pod status” og ser at det dukker opp en metric som heter `kube_pod_status_phase`. Om vi klikker på den vil den automatisk bli fylt inn.

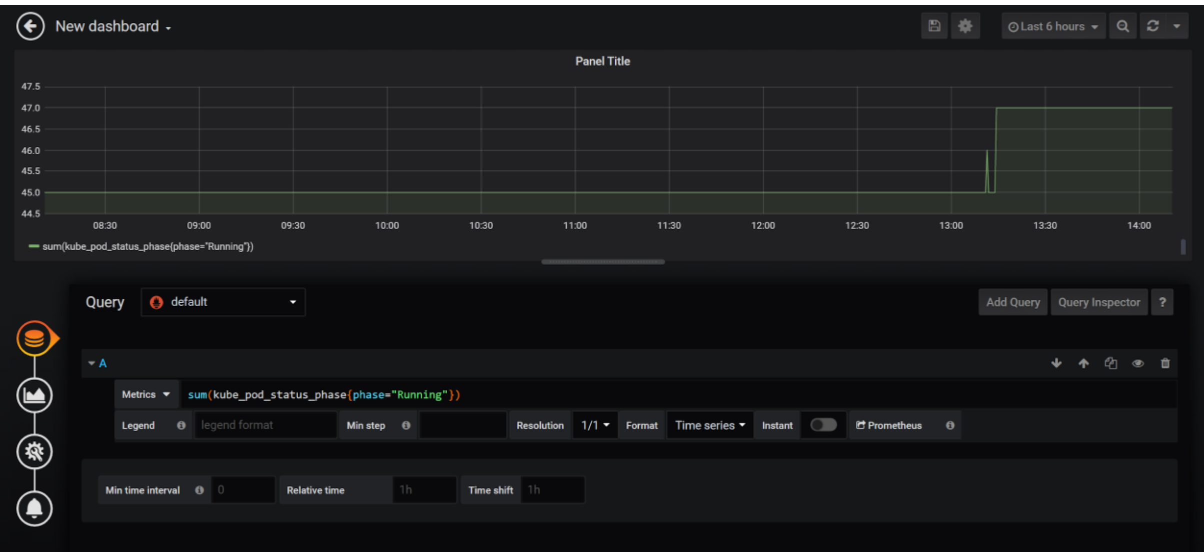


Om vi trykker "Execute" nå vil vi få opp en tabell med én rad for hver fase for hver Pod som har en verdi 0 eller 1 avhengig av om Podden er i den fasen. Det vil si at hver Pod har flere rader, én for hver fase.

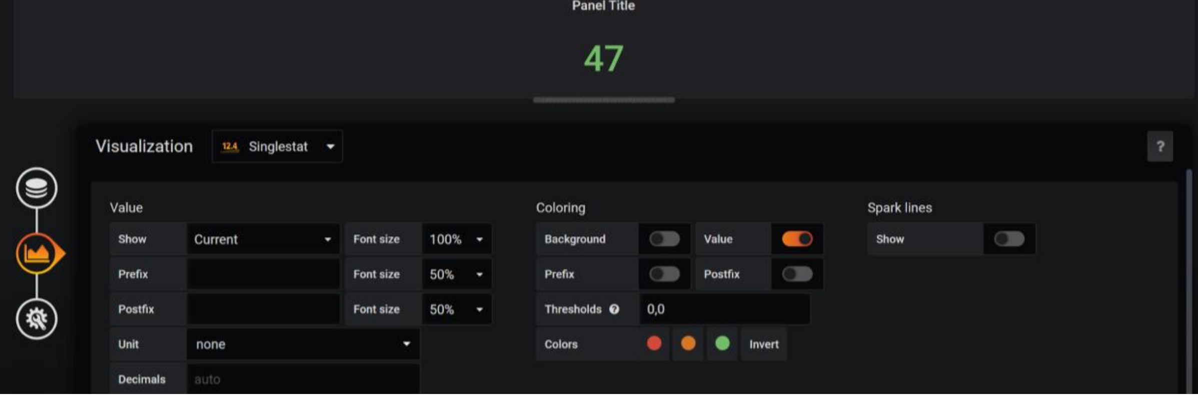
Element	Value
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Failed",pod="mysql-0",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Failed",pod="mysql-1",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Failed",pod="mysql-2",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Failed",pod="phpmyadmin-5ddffb974-ttbfg",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Failed",pod="web-server-d8f7cd477-45rsw",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Pending",pod="mysql-0",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Pending",pod="mysql-1",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Pending",pod="mysql-2",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Pending",pod="phpmyadmin-5ddffb974-ttbfg",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Pending",pod="web-server-d8f7cd477-45rsw",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Running",pod="mysql-0",service="monitor-kube-state-metrics"}	1
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Running",pod="mysql-1",service="monitor-kube-state-metrics"}	1
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Running",pod="mysql-2",service="monitor-kube-state-metrics"}	1
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Running",pod="phpmyadmin-5ddffb974-ttbfg",service="monitor-kube-state-metrics"}	1
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Running",pod="web-server-d8f7cd477-45rsw",service="monitor-kube-state-metrics"}	1
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Succeeded",pod="mysql-0",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Succeeded",pod="mysql-1",service="monitor-kube-state-metrics"}	0
kube_pod_status_phase{endpoint="http",instance="192.168.100.222:8080",job="kube-state-metrics",namespace="default",phase="Succeeded",pod="mysql-2",service="monitor-kube-state-metrics"}	0

For å begrense søket vårt til å bare finne ut om de kjører eller ikke, bruker vi krøllparanteser. Krøllparanteser fungerer litt som `WHERE` i SQL og andre språk. Spørringen vår blir da `kube_pod_status_phase{phase="Running"}`. Siden vi nå kun viser om Poden har status running eller ikke kan vi summere opp verdien i alle radene med `sum(kube_pod_status_phase{phase="Running"})` og finne antallet Podder som har status running. Vi kan også bruke `count(kube_pod_status_phase{phase="Running"})` for å finne totalt antall Pods i clusteret.

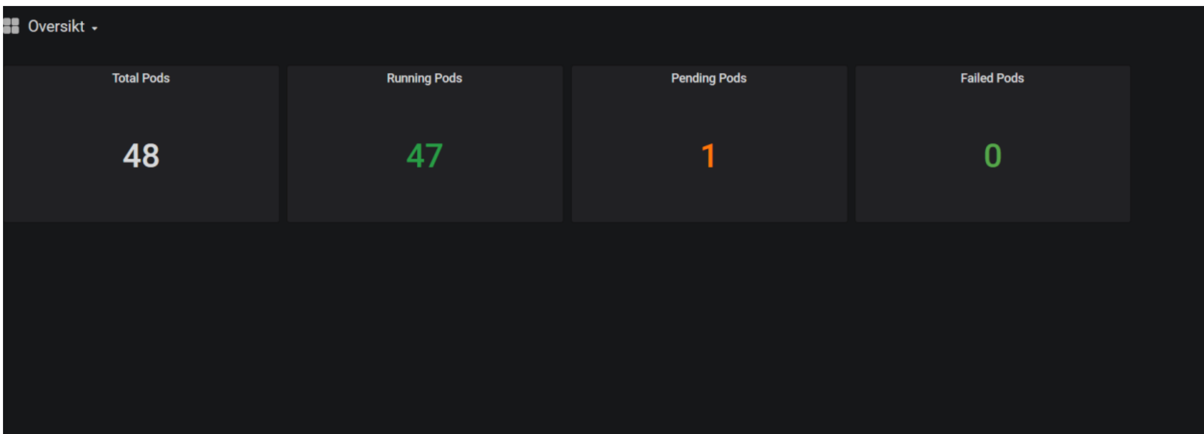
For å vise frem denne informasjonen med Grafana må vi først lage et nytt dashbord. For å lage et nytt dashbord går du til "Create" i menyen til venstre og velger Dashboard. Da vil du få opp et nytt tomt dashbord med et panel. I panelet kan du velge om du vil legge inn spørring eller visualiseringsmetode først. Vi lager panelet som viser antall kjørende Podder først så vi velger "Add Query". Under "Metrics" fyller vi inn spørringen vi testet i Prometheus, `sum(kube_pod_status_phase{phase="Running"})`



Vi kan nå gå videre til visualiseringsfanen til venstre. Her kan du velge hvordan stistikken skal vises frem. Siden vi bare har et enkelt tall å vise bruker vi Singlestat. Det er viktig å endre “Value”, “Show” til “Current”, ellers handler det mest om å stilsette panelet. Siden dette panelet viser antall Pods som kjører riktig kan vi for eksempel gjøre teksten grønn.



I den siste fanen “General” setter vi en tittel på panelet og klikker lagre-knappen oppe til høyre. Vi lager nye paneler og gjør det samme med spørringene `sum(kube_pod_status_phase{phase="Pending"})`, `sum(kube_pod_status_phase{phase="Failed"})` og `count(kube_pod_status_phase{phase="Running"})`. Dashbordet vårt ser nå slik ut.

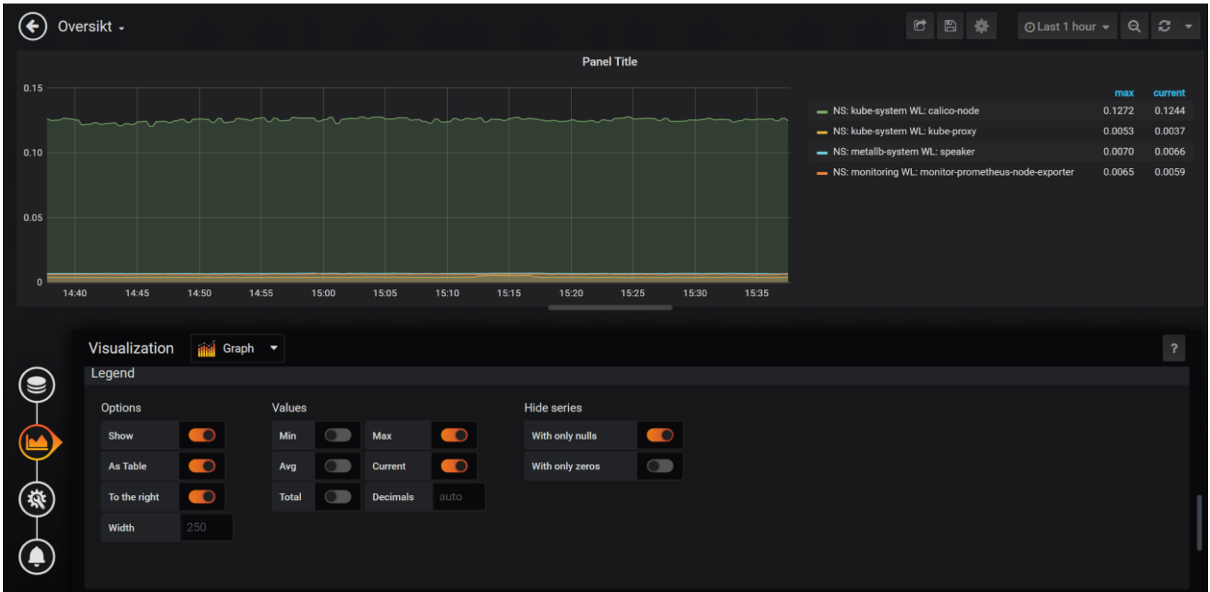


La oss legge til noen grafer som viser ressursbruk. Vi lager tre grafer, en for DaemonSet, en for StatefulSet og en for Deployments. Denne spørringen er litt komplisert

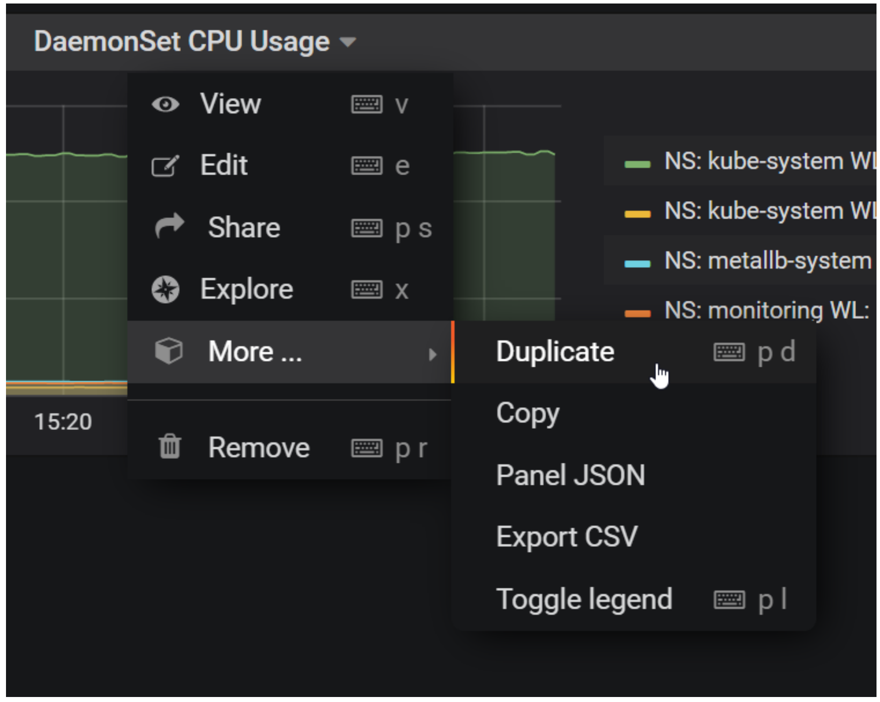
```
sum(
 node_namespace_pod_container:container_cpu_usage_seconds_total:sum_rate
* on(namespace,pod)
  group_left(workload, workload_type)
mixin_pod_workload{workload_type="daemonset"}
) by (namespace, workload, workload_type)
```

Den bruker metricen `node_namespace_pod_container:container_cpu_usage_seconds_total:sum_rate` som viser CPU-bruk og summerer opp CPU-bruken til alle Poddene i et DaemonSet ved å gruppere dem etter hvilket DeamonSet de er med i.

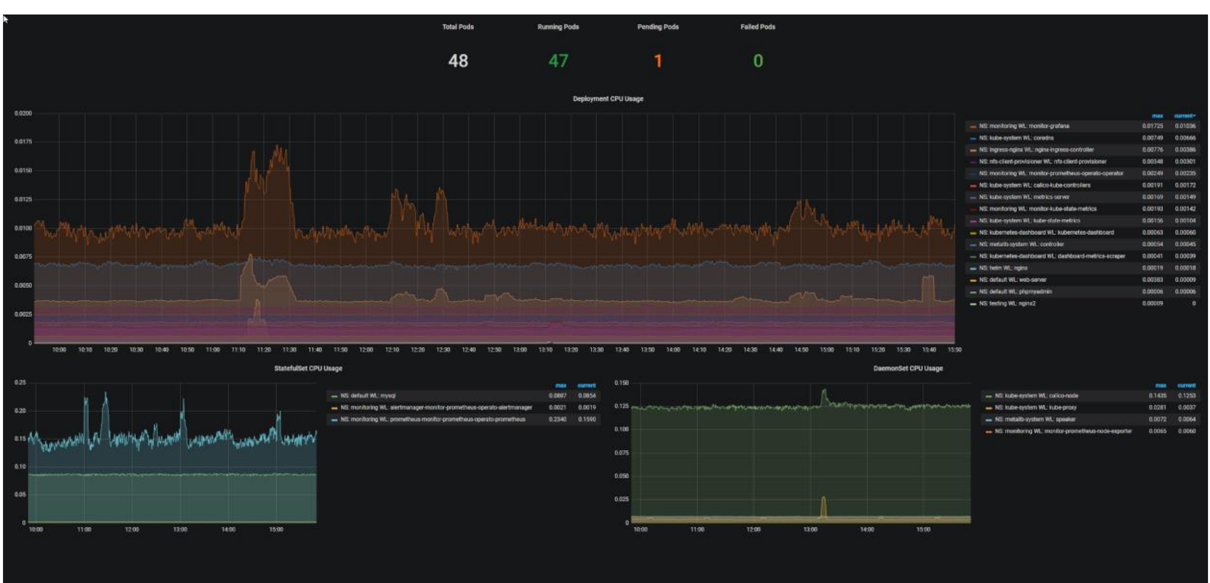
Vi legger denne spørringen inn i et nytt panel i Grafana og setter “Legend” til å være “NS: {{namespace}} WL: {{workload}}” Dette vil gi de forskjellige punktene i grafen leselige navn. Under “Visualization” synes jeg det er greit å vise graf-forklaringen som tabell på høyresiden med maks og nåværende verdier. Dette gjøres under “Legend”.



Siden vi skal ha tre av omtrent samme grafen kan vi gå tilbake til dashbordet vårt og klikke på dropdown-menyen til DaemonSet-grafen og duplisere den to ganger.

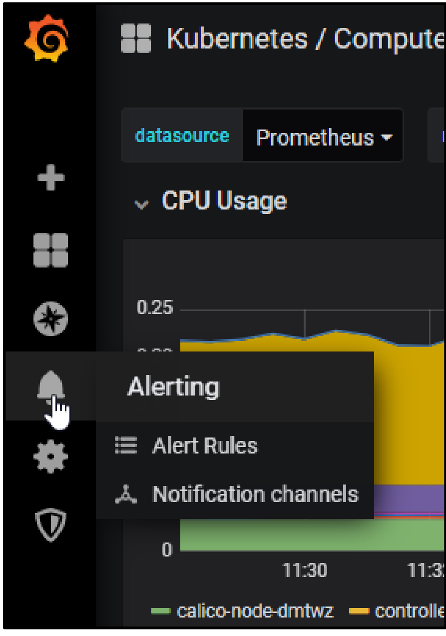


Vi kan nå gå i dropdown-menyen til de grafene vi nettopp laget og klikke “Edit”. Deretter endrer vi spørringen fra `workload_type="daemonset"` til `workload_type="statefulset"` for grafen som skal vise StatefulSet og til `workload_type="deployment"` på Deploymentgrafen. Vi må også endre tittel under “General”-fanen. Etter litt justering av størrelser og omrokkering på ting ser dashbordet vårt slik ut.



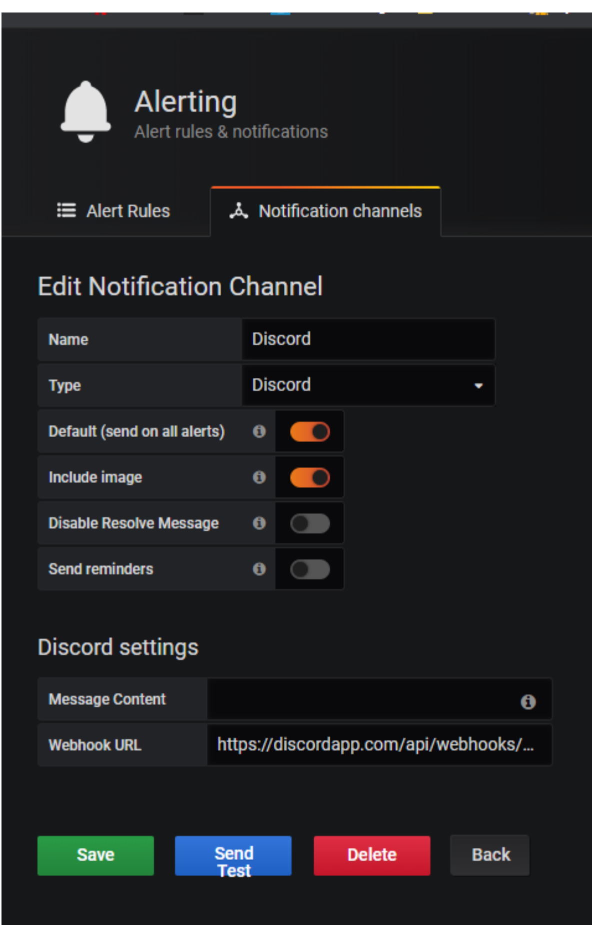
5.3.5 Varsling

Både Grafana og AlertManager har mulighet til å sende varsler når et gitt kriterium inntreffer. Vi velger å bruke Grafana siden det er enklest å sette opp og mere intuitivt i bruk. For å sette opp varslinger må du først sette opp en varslingsmetode. Grafana har mange innebygde måter å kontakte forskjellige tjenester som Microsoft Teams, Slack, Epost og Discord på. For å sette opp en varslingsmetode går du til "Alerting"-menyen til venstre og velger "Notification channels"

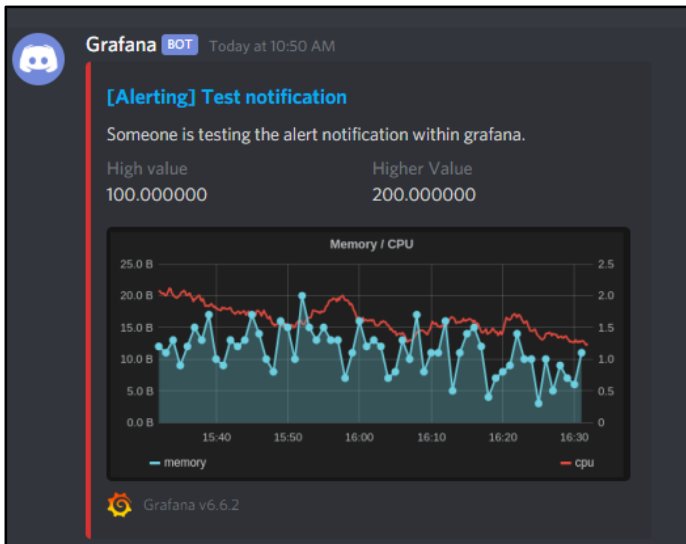


Vi setter opp varsling i Discord siden vi ikke har tilgang til å lage en Webhook i Microsoft Teams, men oppsettet er veldig likt. Under "Notification channels velger vi "New channel" og velger Discord fra dropdown-menyen under "Type". Vi gir kanalen et navn og setter den som Default. Det vil si at alle varsler kommer til å bli sendt gjennom denne kanalen. For å få tak i en Webhook URL høyreklikker vi på kanalen vi vil sende varsler til i Discord og velger "Edit

Channel" og "Webhooks". Vi klikker så "Create Webhook". Her gir vi den et navn og kopierer URLen. Vi limer så inn URLen i Grafana under "Webhook URL".



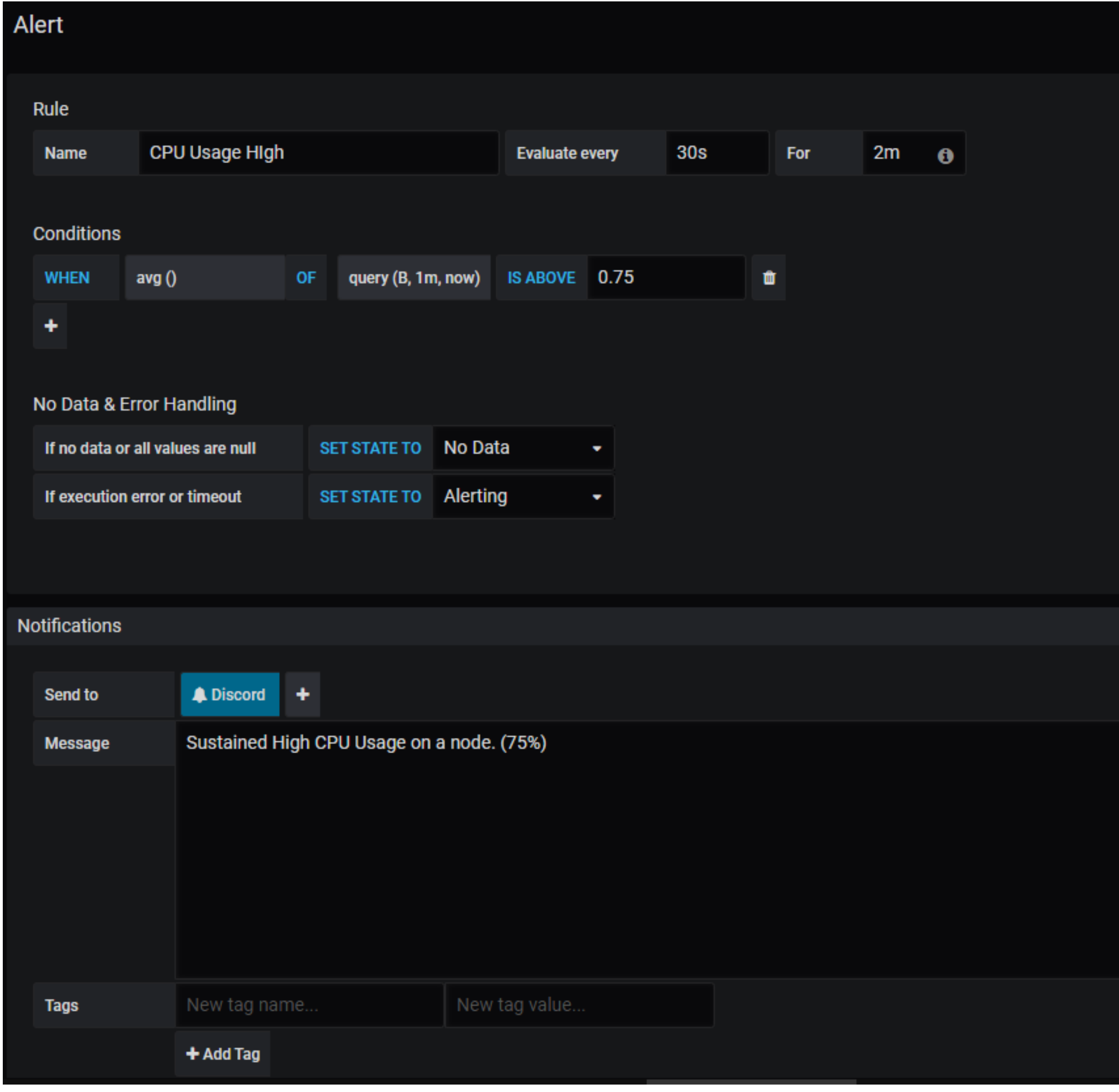
Vi kan klikke "Send Test" for å sjekke om koblingen funket. Da vil det dukke opp en testmelding i kanalen.



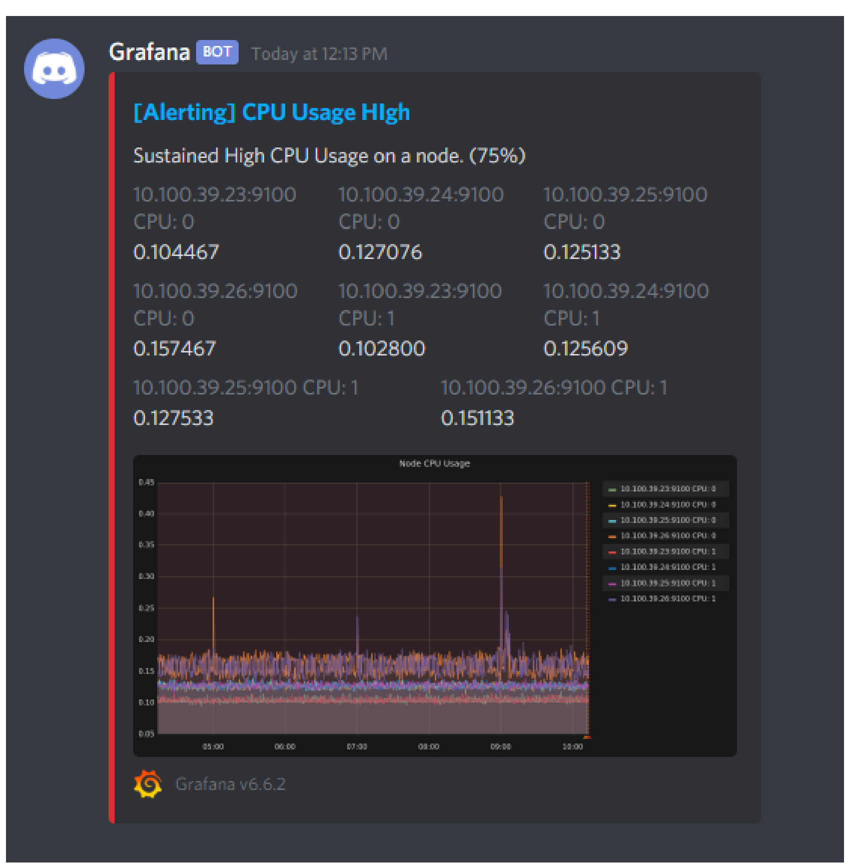
Ulempen med å bruke Grafana istedenfor AlertManager er at du er litt mer begrenset i forhold til hva du kan sende varslinger om. Det er kun enkelte av panelene som støtter

varslinger, og du kan kun ha én varsling per panel. Dette er imidlertid mulig å omgå ved å bruke OR for å ha flere kriterier per panel, men teksten som sendes sammen med meldingen vil bli den samme.

For å sette opp en varsling på et panel går vi til panelet vi ønsker å varsle om. Jeg har satt opp et enkelt panel som viser CPU-bruk per kjerne per node som en graf med spørringen `1 - (rate(node_cpu_seconds_total{mode="idle"}[1m]))`. Vi velger "Alert" i menyen til venstre og klikker "Create alert". Vi fyller inn et navn, hvor ofte den skal sjekke og hvor lenge kriteriet må være oppfyllt for at den skal sende en varsling med "Name", "Evaluate every" og "For". Under "Conditions" setter du kriteriet for at den skal sende et varsel. Jeg satt kriteriet til at den skal varlse om snittbruken det siste minuttet er høyere enn 75% med `WHEN avg() OF query(B, 1m, now) IS ABOVE 0.75`. Query-funksjonen leses som query(<hvilket query>, <fra>, <til>) så query(B, 1m, now) ser på query B fra ett minutt siden til nå.



For å teste varslingen satt jeg grensen for når den skal rapportere til over 0.1, altså 10% bruk og etter evalueringstiden var omme kom det en melding på Discord.



Grafana
BOT
Today at 12:13 PM
[Alerting] CPU Usage HIgh
Sustained High CPU Usage on a node. (75%)
10.100.39.23:9100
CPU: 0
0.104467
10.100.39.24:9100
CPU: 0
0.127076
10.100.39.25:9100
CPU: 0
0.125133
10.100.39.26:9100
CPU: 0
0.157467
10.100.39.23:9100
CPU: 1
0.102800
10.100.39.24:9100
CPU: 1
0.125609
10.100.39.25:9100 CPU: 1
0.127533
10.100.39.26:9100 CPU: 1
0.151133
Node CPU Usage
Grafana v6.6.2

5.4 Cert-manager

Cert-manager er et verktøy for automatisk generering og fornyelse av TLS-sertifikater. Cert-manager støtter Automated Certificate Management Environment, eller ACME, som gjør det enkelt og automatisk å lage sertifikater. Det er også mulig å bruke verifisering over DNS gjennom ACME. Dette er veldig viktig for oss siden vi ikke har en offentlig IP-adresse å bruke til å skaffe oss sertifikater på den vanlige måten som er det de kaller HTTP-verifisering. HTTP-verifisering går ut på at ACME starter en liten web-server med en streng spesifikk til deg og sjekker at den eksisterer der du påstår ved å prøve å hente den fra web-serveren over internett. Cert-manager kan levere sertifikater fra blant annet Let's Encrypt, HashiCorp Vault, Venafi eller signere det selv. Den lar deg også bruke flere leverandører samtidig med en egen Custom Resource som gjør det enkelt å bytte mellom testmiljøer og produksjonsmiljøer hos for eksempel Let's Encrypt. Dette er veldig nyttig siden Let's Encrypt begrenser antall API-kall på sertifikatleveranser og under testing og oppsett kan dette fort bli brukt opp hvis det er noe feil i innstillingene dine. Siden vi er nødt til å eie domenet for at vi skal kunne få tildelt et sertifikat bruker vi et av våre egne domener i denne delen.

5.4.1 Installasjon

Vi velger å installere cert-manager ved hjelp av Helm. For å slippe å lage ressurser som Issuer og Certificate på nytt om vi er nødt til å reinstallere cert-manager blir disse installert manuelt med ressursfiler før vi installerer resten av cert-manager. Dette gjøres ved å kjøre kommandoen `kubectl apply --validate=false -f https://github.com/jetstack/cert-manager/releases/download/v0.14.1/cert-manager.crds.yaml`. Denne kommandoen installerer flere Custom Resource Definitions som brukes til å sette opp blant annet sertifikatutsteder og metode for autentisering. De brukes også til å lagre informasjon om sertifikatene på en lesbar måte. Siden de ikke er en del av Helm chartet blir de heller ikke påvirket hvis vi kjører `helm uninstall`. Det neste vi må gjøre er å lage et nytt Namespace. Dette gjøres med kommandoen `kubectl create namespace cert-manager`. Så må vi legge repositoryet til chartet i Helm med kommandoen `helm repo add jetstack https://charts.jetstack.io` for så å oppdatere den lokale cachen av repositoryer med `helm repo update`.

Før vi installerer chartet legger vi til en ressurs, Issuer, som er konfigurasjon av sertifikatutstederen. Vi vil at denne skal brukes som default og det må defineres under installasjon så vi lager den nå.

```yaml
                                              5_4_cert-manager\staging-acme.yaml
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
  acme:
    email: magnus.torset@gmail.com
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
```

```
      name: example-issuer-account-key
    solvers:
    - dns01:
        cloudflare:
          email: magnus.torset@gmail.com
          apiTokenSecretRef:
            name: cloudflare-api-token-secret
            key: api-token
```

Vi ser av `apiVersion:` at dette er en Custom Resource. Det vil si at det ikke er en ressurs som er innebygd i Kubernetes, men som er lagt til av en annen tjeneste. I `kind:` defineres det at dette er en `ClusterIssuer`. Det vil si at den har tilgang til hele clusteret og er ikke begrenset til ett Namespace som `Issuer` er. Under `acme:` fyller vi inn nødvendig informasjon for å få tildelt et sertifikat som hvilken autentiseringsserver som skal brukes. Vi bruker Let's Encrypt sin staging-server. Staging er et testmiljø så den vil ikke gi oss et gyldig sertifikat, men den gir oss et sertifikat som viser at oppsettet vårt er riktig. Hvis vi får tildelt et sertifikat fra staging-serveren er det bare å bytte til produksjonsserveren for å få et gyldig sertifikat. Dette er nyttig fordi Let's Encrypt har begrensninger på hvor mange API-kall man kan gjøre mot produksjonsserveren, men betydelig mindre begrensninger på staging-serveren så man bruker staging-serveren til å passe på at man har riktig oppsett for så å bytte over til produksjonsserveren. Det neste vi setter opp i ressursfilen er `privateKeySecretRef:`. Denne vil bli generert automatisk og er bare lagring av kredentialer for utstederen. `solvers:` er måten utstederen skal bekrefte at det er du som eier domenet og at det eksisterer. Her har du to alternativ, http01 og dns01. http01 setter opp en webserver som utstederen prøver å nå på det domenenavnet du oppgir. Dette krever at det er mulig å nå serveren din fra internett på port 80 som ikke er mulig for oss. Vi velger derfor å bruke dns01. dns01 bruker API-et til DNS-leverandøren din for å bekrefte at det er du som eier domenet ved å prøve å lage et TXT-record i samme sone som domenet du vil ha sertifikat for. Dette gjør det mulig å automatisk dele ut og fornye sertifikater signert av en ekstern utsteder til bruk i interne tjenester. Ikke alle DNS-leverandører er støttet. Cert-manager har en liste over støttede leverandører i dokumentasjonen sin. Vi bruker Cloudflare som er en av de støttede leverandørene. Det siste vi gjør er å lage en referanse til en Secret som inneholder en API-Token til DNSen vår så ACME kan lage et TXT-record. Denne lages ved å legge til denne Secreten:

5_4_cert-manager\cf-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: cloudflare-api-token-secret
type: Opaque
stringData:
  api-token: <API-token>
```

Nå skal Issueren vår være på plass så nå kan vi installere cert-manager med Helm. Dette gjør vi ved å kjøre denne kommandoen:

```
helm install \
  cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --set ingressShim.defaultIssuerName="letsencrypt-staging" \
  --set ingressShim.defaultIssuerKind="ClusterIssuer" \
  --set ingressShim.defaultACMEChallengeType="dns01" \
  --set ingressShim.defaultACMEDNS01ChallengeProvider="cloudflare" \
  --set 'extraArgs={--dns01-recursive-
nameservers=1.1.1.1:53\,1.0.0.1:53}' \
  --version v0.14.1
```

Vi kjører `helm install` og gir installasjonen et navn. Dette kan være hva som helst, vi valgte å bare kalle den cert-manager. Alle ressurser som lages av Helm-chartet vil få dette som en prefix. Vi velger repo og chart som er `jetstack/cert-manager`. Vi setter hvilket Namespace vi vil cert-manager skal ligge i med `--namespace` og setter noen verdier for konfigurasjonen av cert-manager med `--set`. `ingressShim.defaultIssuerName` setter Issueren vi nettop lagde til default. `ingressShim.defaultIssuerKind` sier hvilken type Issuer det er, om det er en ClusterIssuer som gjelder i alle Namespace eller en vanlig Issuer som kun gjelder for ett Namespace. `ingressShim.defaultACMEChallengeType` forteller cert-manager hvilken type challenge som skal brukes for å få tak i sertifikater. `ingressShim.defaultACMEDNS01ChallengeProvider` sier hvilken DNS-tilbyder vi bruker. `'extraArgs={--dns01-recursive-nameservers=1.1.1.1:53\,1.0.0.1:53}'` sier at vi skal bruke disse navneserverne i challengen. Normalt ville den bare brukt navneserverne i resolv.conf, men dette ga oss problemer så vi er nødt til å spesifisere de her. Til slutt sier `--version` hvilken versjon vi ønsker å installere.

For å sjekke at installasjonen fungerer lager vi en kjapp webserver med `kubectl apply -f https://k8s.io/examples/application/deployment.yaml`. Dette er en av eksemplene fra Kubernetes sin dokumentasjon som setter opp en Deployment av Nginx med to Podder. Vi lager en Service til Deploymenten med `kubectl expose deployment.apps/nginx-deployment`. Nå må vi lage en Ingress.

5_4_cert-manager\test-ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/ingress.class: ingress-nginx
    cert-manager.io/cluster-issuer: "letsencrypt-staging"
spec:
  tls:
    - hosts:
      - nginx.k.torset.me
      secretName: test-tls
  rules:
  - host: nginx.k.torset.me
    http:
      paths:
      - path: /
        backend:
          serviceName: nginx-deployment
          servicePort: 80
status:
  loadBalancer: {}
```

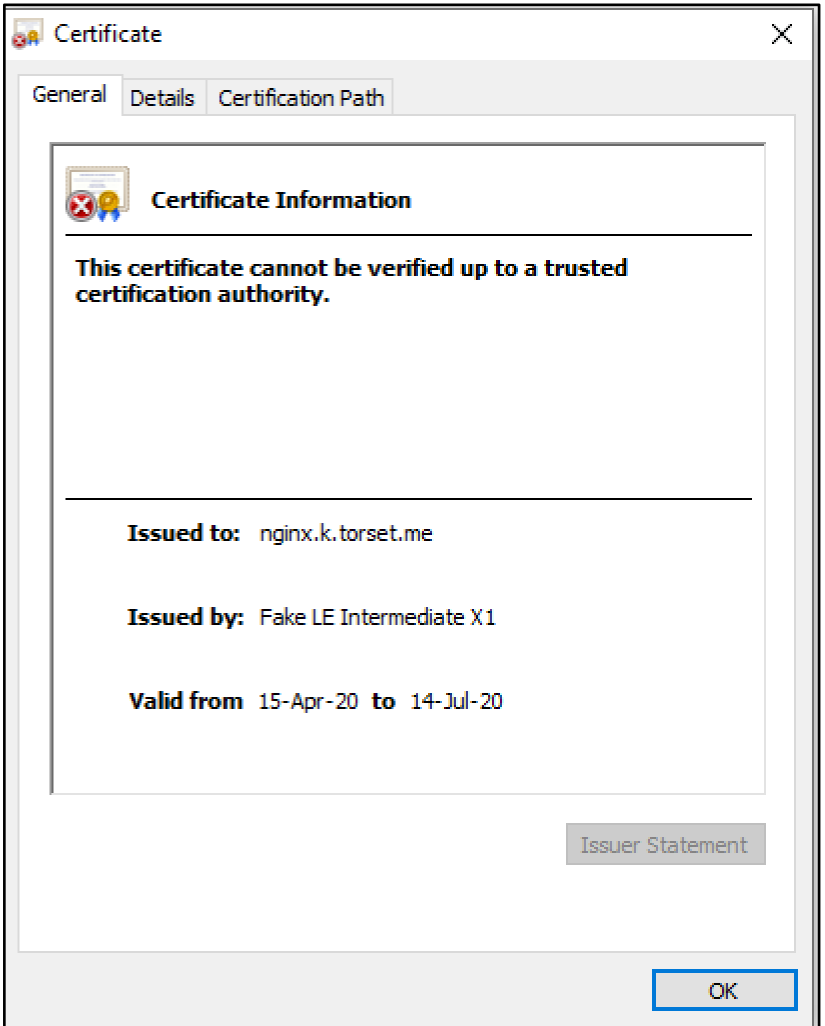
Denne Ingressen er satt opp som normalt, men vi legger i tillegg til feltet `tls:` med `hosts:` og `secretName:` og annotationen `cert-manager.io/cluster-issuer: "letsencrypt-staging"`. secretName er det du vil Secreten med sertfikatet skal hete. Den trenger ikke lages på forhånd. Annotationen sier at det skal genereres et sertifikat og hvilken issuer som skal brukes. Vi må også legge til et DNS-record for hostnavnet vi har tenkt å bruke hos DNS-tilbyderen vår siden ACME sjekker om det eksisterer, selv om vi kun bruker intern DNS. Nå trenger vi bare legge til denne med `kubectl apply -f ingress.yaml` og vente på å få tildelt et sertifikat. Dette kan ta opptil 15 minutter. For å se hvordan sertifikatgenereringen ligger an kan vi kjøre `kubectl get certificates`. Da vil vi få en liste over sertifikater i det Namespacet. Hvis vi kjører `kubectl describe certificate <sertifikatnavn>` kan vi få info om sertifikatet og om det er klart.

```
magnus@k8s-master:/opt/kubernetes/nginx-deployment$ kubectl describe certificate test-tls
Name:         test-tls
Namespace:    testing
Labels:       <none>
Annotations:  <none>
API Version:  cert-manager.io/v1alpha3
Kind:         Certificate
Metadata:
  Creation Timestamp:  2020-04-15T11:06:34Z
  Generation:          1
  Owner References:
    API Version:           extensions/v1beta1
    Block Owner Deletion:  true
    Controller:            true
    Kind:                  Ingress
    Name:                  test-ingress
    UID:                   0512a11d-8455-4fdc-b441-5f438726a16a
  Resource Version:        13694265
  Self Link:               /apis/cert-manager.io/v1alpha3/namespaces/testing/certificates/test-tls
  UID:                     5e6b1c18-7733-421b-aaa7-f465883c947b
Spec:
  Dns Names:
    nginx.k.torset.me
  Issuer Ref:
    Group:      cert-manager.io
    Kind:       ClusterIssuer
    Name:       letsencrypt-staging
  Secret Name:  test-tls
Status:
  Conditions:
    Last Transition Time:  2020-04-15T11:08:10Z
    Message:               Certificate is up to date and has not expired
    Reason:                Ready
    Status:                True
    Type:                  Ready
  Not After:               2020-07-14T10:08:09Z
Events:
  Type    Reason     Age   From          Message
  ----    ------     ----  ----          -------
  Normal  Requested  19m   cert-manager  Created new CertificateRequest resource "test-tls-1108908235"
  Normal  Issued     18m   cert-manager  Certificate issued successfully
```

Under Events ser vi at det lages en ressurs CertificateRequest. Vi kan se denne ved å kjøre `kubectl get certificaterequest` eller `kubectl get cr`. Også her kan vi bruke describe for å få status med `kubectl describe cr <certificaterequest-navn>`.

```
magnus@k8s-master:/opt/kubernetes/nginx-deployment$ kubectl describe cr test-tls-1108908235
Name:         test-tls-1108908235
Namespace:    testing
Labels:       <none>
Annotations:  cert-manager.io/certificate-name: test-tls
              cert-manager.io/private-key-secret-name: test-tls
API Version:  cert-manager.io/v1alpha3
Kind:         CertificateRequest
Metadata:
  Creation Timestamp:  2020-04-15T11:06:34Z
  Generation:          1
  Owner References:
    API Version:           cert-manager.io/v1alpha2
    Block Owner Deletion:  true
    Controller:            true
    Kind:                  Certificate
    Name:                  test-tls
    UID:                   5e6b1c18-7733-421b-aaa7-f465883c947b
  Resource Version:        13694262
  Self Link:               /apis/cert-manager.io/v1alpha3/namespaces/testing/certificaterequests/test-tls-1108908235
  UID:                     47035e84-38aa-482a-8c28-e9f418d4fbb3
Spec:
  Csr:  LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQ2l6Q0NBWE1DQVFBd0Z6RVZNQk1HQTFVRUNoTU1ZMlZ5ZEMxdFlXNWhaMlZ5TUlJQklqQU5CZ2txaGtpR
tNG94MFdzVGNTMmljdEFNMEl5MVBHd2VPa3kzalZJUzlNY2E0cWhaMG5oOCtQajg5b2FtWFBSeUFNNFpzMUkKNHRLN3NHSlRIMURjNFVPeEdSZ0xOalhjdjhITGl2K3RhVi9FRmNHYWlY
4wbGxiRHdOUTZNT0RtQ0JoTzdMCmpJQ1RuWnBiZ0owakxXZnNzWjAxcWRVUEw1WERHVGxjb1BhcHpEMmJtS2NNYi9zR0lnM3IvWUVFUElZWk5JYXcKTmVtanl5YmFFb3E2S3IvaTlWRTh
lZIUkVFRlRBVGdoRnVaMmx1ZUM1ckxuUnZjbk5sZEM1dFpUQU5CZ2txCmhraUc5dzBCQVFzRkFBT0NBUUVBT2VROU43WVkxd25GU1pESk5hcS9VS3hyeVBHei9ocGhob0NzYlUrSHZvZj
OFFreWtxU3EvcFVxRWlMVWpWSzE1QlFMekhJRU43K2hPczIzU3h5SW1GS0kzUTBpQlZnYk5uMVZkUHY1YUQwCmxSR0JmdWRpbkRsc0ovTm9FVElWTy80dWJLU2g3VDJSWUhOeTQ1ZVBPZ
4cEl0SU5PU0lGcWI5VytxRml1TQpsTUJrMVIyMW5MajZ3RU1LdWJPMEdpblgvUytnQjhLTmw1aEhxV1dyeXc9PQotLS0tLUVORCBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0K
  Issuer Ref:
    Group:  cert-manager.io
    Kind:   ClusterIssuer
    Name:   letsencrypt-staging
Status:
  Certificate:  LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUZQRENDQkNTZ0F3SUJBZ0lUQVBwZ3E3L2F0QnpxWnJDYURnRlJTNHNZclRBTkJna3Foa2lHOXcwQkFRc0YKQ
0TURsYUZ3MHlNREEzTVRReE1EQTRNRGxhTUJ3eEdqQVlCZ05WQkFNVEVXNW5hVzU0TG1zdWRHOXljMlYwCkxtMWxNSUlCSWpBTkJna3Foa2lHOXcwQkFRRUZBQU9DQVE4QU1JSUJDZ0tD
t5M2pWSVM5TWNhNHFoWjBuaAo4K1BqODlvYW1YUFJ5QU00WnMxSTR0SzdzR0pUSDFEYzRVT3hHUmdMTmpYY3Y4SExpdit0YVYvRUZjR2FpWE9BCmpaOWY3dnBHVk5lY0JJUzhtNmxFSlZ
lpwYmdKMGpMV2Zzc1owMXFkVVBMNVhER1RsY29QYXB6RDJibUtjTQpiL3NHSWczci9ZRUVQSVlaTklhd05lbWp5eWJhRW9xNktyL2k5VkU4Qnc5cFJYZXNicER3YThPWlllNHh6d0ZmCk
RkJ3TUJCZ2dyQmdFRkJRY0RBakFNQmdOVkhSTUJBZjhFQWpBQU1CMEdBMVVkRGdRV0JCVGtWUVFxNU9ZWgpYaXdPTTZmN3FzbTRHdlRaTmpBZkJnTlZIU01FR0RBV2dCVEF6QU5HdVZnZ
5dlkzTndMbk4wWnkxcGJuUXQKZURFdWJHVjBjMlZ1WTNKNWNIUXViM0puTURNR0NDc0dBUVVGQnpBQ2hpZG9kSFJ3T2k4dlkyVnlkQzV6ZEdjdAphVzUwTFhneExteGxkSE5sYm1OeWVY
JnWm5nUXdCQWdFd053WUxLd1lCQkFHQzN4TUJBUUV3S0RBbUJnZ3IKQmdFRkJRY0NBUllhYUhSMGNEb3ZMMk53Y3k1clpYUnpaVzVqY25sd2RDNXZjbWN3Z2dFRkJnb3JCZ0VGQWRaNQp
jlobHJMQUFBRUF3QklNRVlDSVFDN0d5enF3S2dxQU1kaFBJYm9LTkFoRnRzQWJZWk4yTEZyQUVRZ0UwM1EKU1FJaEFQVnFuLytGMU5lV2MwekxIeWE1a0FEVFNic2RaMU02TXVPcmhZVF
TUFSekJGQWlFQTFvYVJFV2pGCmhFd3lFTU96K2RHZFRjZktUdDdCLzVyUGcrOW91eVdxTDFJQ0lFYTl0a2lRUXh5cVFmRTZneFNwTFlYYmhYKzUKUTRwdGtpQzVuUG9FeVo0eU1BMEdDU
keU40ZlpyMzd5SHErZ0JLd3dNUHVNMSsxT0czNUtJWDQ4cmVhdDF4CkgvMTYyN1o2cDZoN3FsWmN3bVpFYktvNzVlZXliemNnY3RNT0twTCtpTDlOaGVZVktxRCtJbnBwYTFUaU1WMUUK
NIZ28zZWhvZTBKWnlNYjVmMDVSZ2pVeG9Da0hrWktMcTkrMmlVUzMweUFKUmNNZmp0VzNWK1NNNEFOa3FwCmJHWElsVktweThEOVNYZzdiYURoYUFnR2gvTTJ6enprZXI4Q3JTbUkwT08
VRFLS0tLS0KTUlJRXF6Q0NBcE9nQXdJQkFnSVJBSXZoS2c1WlJPMDhWR1F4OEpkaFQrVXdEUVlKS29aSWh2Y05BUUVMQlFBdwpHakVZTUJZR0ExVUVBd3dQUm1GclpTQk1SU0JTYjI5ME
clpTQk1SU0JKYm5SbGNtMWxaR2xoZEdVZ1dERXcKZ2dFaU1BMEdDU3FHU0liM0RRRUJBUVVBQTRJQkR3QXdnZ0VLQW9JQkFRRHRXS3lTRG43cldaYzVnZ2p6M1pCMAo4ak80eHRpM3V6S
ORHpJZWMwT0lhZmxXcUFyMjltN0o4K05OdEFwRU44blpGbmYzYmhlaFpXN0F4bVMxbTAKWm5Tc2RIdzBGdytiZ2l4UGcyTVE5azlvZWZGZXFhKzdLcWRsejViYnJVWVYydm9seGhERnRu
JmTzJKZTU2CmRoTWZ6Q2dPS1hlSjM0MFdoVzNUakQxenFMWlhlYUN5VU5SbmZPbVdaVjhuRWh0SE9GYlVDVTdyL0trak1aTzkKQWdNQkFBR2pnZU13Z2VBd0RnWURWUjBQQVFIL0JBUUR
205V2c2TUhvR0NDc0dBUVVGQndFQkJHNHdiREEwCkJnZ3JCZ0VGQlFjd0FZWW9hSFIwY0RvdkwyOWpjM0F1YzNSbkxYSnZiM1F0ZURFdWJHVjBjMlZ1WTNKNWNIUXUKYjNKbkx6QTBCZ2
ekFmQmdOVkhTTUVHREFXZ0JUQkpuU2tpa1NnNXZvZ0tOaGNJNXBGaUJoNTREQU5CZ2txCmhraUc5dzBCQVFzRkFBT0NBZ0VBQllTdTRJbCtmSTBNWVU0Mk9UbUVqKzFIcVE1RHZ5QWV5Q
zaWVUc2FDOQpBRnZDU2dOSEp5V1NWdFd2QjVYRHhzcWF3bDFLekh6endyMTMyYkYycnRHdGF6U3FWcUs5RTA3c0dITUNmK3pwCkRRVkRWVkd0cVpQSHdYM0txVXRlZkU2MjFiOFJJNlZD
JOV0hTOXBZR0dRS0ppWTJ4bVZDOWgxMkg5OVh5Zgp6V0U5dmI1ektQM01WRzZuZVgxaFNkbzdQRUFiOWZxUmhIa3FWc3FVdkpsSVJtdlh2VktUd05DUDNlQ2pSQ0NJClBUQXZqVis0bmk
jlsdE5XVWlZWkhKMW1tTzk3alNZLzZZZmRPVUg2NmlSdFF0RGtIQlJka05Cc01iRCtFbQoyVGdCbGR0SE5TSkJmQjNwbTlGYmxnT2NKMEZTV2NVRFdKN3ZPMCtOVFhsZ3JSb2ZSVDZwVn
N1JGUTB1eXQKbjVaNU1xa1lobE1JM0oxdFBSVHAxbkV0OWZ5R3NwQk9PMDVnaTE0OFFhc3ArM04rc3ZxS29tb1FnbE5vQXhVPQotLS0tLUVORCBDRVJUSUZJQ0FURS0tLS0tCg==
  Conditions:
    Last Transition Time:  2020-04-15T11:08:10Z
    Message:               Certificate fetched from issuer successfully
    Reason:                Issued
    Status:                True
    Type:                  Ready
Events:
  Type    Reason              Age   From          Message
  ----    ------              ----  ----          -------
  Normal  OrderCreated        22m   cert-manager  Created Order resource testing/test-tls-1108908235-3149095411
  Normal  CertificateIssued   21m   cert-manager  Certificate fetched from issuer successfully
```

Her ser vi hele sertifikatet og under Events ser vi at sertifikatet ble hentet. Hvis vi går til nginx.k.torset.me vil vi fortsatt få advarsel om at sertifikatet ikke er sikkert siden vi bruker staging-miljøet til Let’s Encrypt. Hvordan man ser sertifikatet avhenger av hvilken nettleser man bruker, men hvis man i Google Chrome klikker på varseltrekanten med “Not secure” på adresselinjen og klikker Certificate ser vi at vi er blitt tildelt et sertifikat fra “Fake LE Intermediate X1” som er gyldig i tre måneder.



5.4.2 Bruk

5.4.2.1 Sikring gjennom Ingress

Til nå har vi brukt staging-miljøet til Let's Encrypt. Dette lar oss generere sertifikater og sjekke om innstillingene våre er riktige, men sertifikatene er ikke gyldige. For å få gyldige sertifikater må vi bruke produksjonsmiljøet deres. Dette gjør vi ved å legge til en ny issuer.

5_4_cert-manager\prod-acme.yaml

```yaml
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    email: magnus.torset@gmail.com
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: example-issuer-account-key
```

```
    solvers:
    - dns01:
        cloudflare:
          email: magnus.torset@gmail.com
          apiTokenSecretRef:
            name: cloudflare-api-token-secret
            key: api-token
```

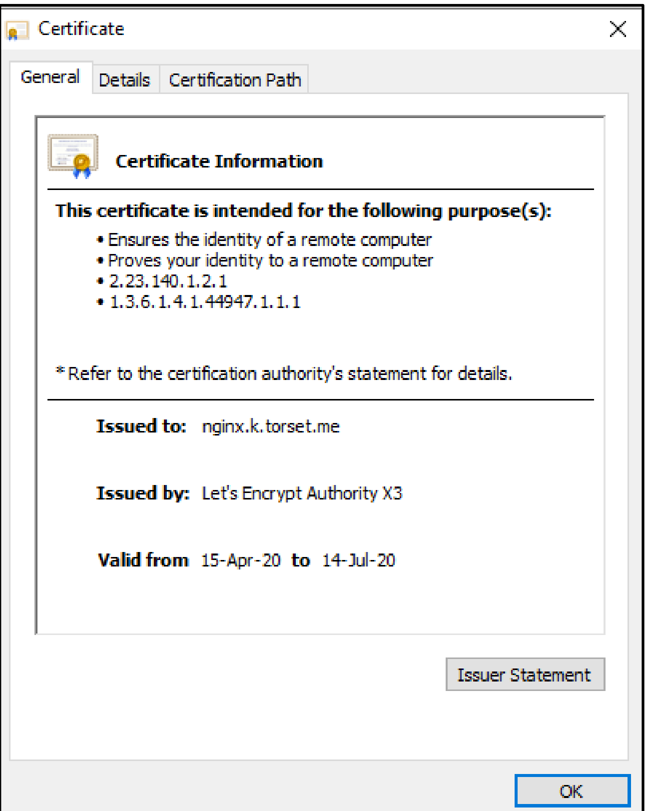
Denne er helt lik staging-issueren vi la til under installasjon bortsett fra navnet og server-linjen. Denne er nå endret fra staging-miljøet til produksjonsmiljøet. Vi kan også bruke den samme Secreten for API-nøkkel til cloudflare. Vi lager denne filen og legger den til med `kubectl apply -f <filnavn>`.

Nå kan vi endre på Ingressen til nginx-serveren vår med `kubectl edit ingress test-ingress`. Under `annotations:` endrer vi `cert-manager.io/cluster-issuer:` fra `letsencrypt-staging` til `letsencrypt-prod`. Den redigerte Ingressen skal nå se slik ut :

```
# Please edit the object below. Lines beginning with a '#' will be
ignored,
# and an empty file will abort the edit. If an error occurs while saving
this file will be
# reopened with the relevant failures.
#
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"extensions/v1beta1","kind":"Ingress","metadata":{"annotat
ions":{"cert-manager.io/cluster-issuer":"letsencrypt-
staging","kubernetes.io/ingress.class":"ingress-nginx"},"name":"test-
ingress","namespace":"testing"$
    kubernetes.io/ingress.class: ingress-nginx
  creationTimestamp: "2020-04-15T11:06:34Z"
  generation: 1
  name: test-ingress
  namespace: testing
  resourceVersion: "13693818"
  selfLink: /apis/extensions/v1beta1/namespaces/testing/ingresses/test-
ingress
  uid: 0512a11d-8455-4fdc-b441-5f438726a16a
spec:
  rules:
```

```yaml
  - host: nginx.k.torset.me
    http:
      paths:
      - backend:
          serviceName: nginx-deployment
          servicePort: 80
        path: /
  tls:
  - hosts:
    - nginx.k.torset.me
    secretName: test-tls
status:
  loadBalancer: {}
```

Så lagrer vi og lukker editoren. Ingressen vil nå bli oppdatert og den vil lage en ny CertificateRequest som vi kan se med `kubectl describe cr <CertificateRequst-navn>`. Når sertifikatet er klart kan vi igjen gå til nginx.k.torset.me og vil nå ikke lenger bli møtt med advarsel om ugyldig sertifikat og det vil være en hengelås istedenfor en advarsel på adresselinjen. Hvis vi igjen sjekker sertifikatet vil vi se at det er gyldig og utstedt fra Let's Encrypt Authority X3.



5.4.2.2 Selvstendige sertifikater

Noen ganger er det nødvendig å lage et sertifikat uten å gjøre det gjennom en Ingress. Dette gjøres ved at vi lager vår egen certificate-ressurs istedenfor å la Ingressen generere den for oss.

5_4_cert-manager\cert-manager_default_cert.yaml

```yaml
apiVersion: cert-manager.io/v1alpha2
kind: Certificate
metadata:
  name: example-com
  namespace: sandbox
spec:
  # Secret names are always required.
  secretName: example-com-tls
  duration: 2160h # 90d
  renewBefore: 360h # 15d
  organization:
  - jetstack
  # The use of the common name field has been deprecated since 2000 and is
  # discouraged from being used.
  commonName: example.com
  isCA: false
  keySize: 2048
  keyAlgorithm: rsa
  keyEncoding: pkcs1
  usages:
    - server auth
    - client auth
  # At least one of a DNS Name, URI, or IP address is required.
  dnsNames:
  - example.com
  - www.example.com
  uriSANs:
  - spiffe://cluster.local/ns/sandbox/sa/example
  ipAddresses:
  - 192.168.0.5
  # Issuer references are always required.
  issuerRef:
    name: ca-issuer
    # We can reference ClusterIssuers by changing the kind here.
    # The default value is Issuer (i.e. a locally namespaced Issuer)
    kind: Issuer
    # This is optional since cert-manager will default to this value however
    # if you are using an external issuer, change this to that issuer group.
    group: cert-manager.io
```

Dette er en generisk certificate-ressursfil tatt fra dokumentasjonen til cert-manager. Vi ser av kommentarene at ikke alle feltene er nødvendige. De viktige feltene som må endres på her

er `secretName:` som er navnet på Secreten som skal inneholde sertifikatet, `dnsNames:`, `uriSANs:` eller `ipAddresses:` som er dns-navnet, URIen eller IP-addressen til tjenesten som skal få tilknyttet et sertifikat, `organization:` som er organisasjonsnavnet som skal vises i sertifikatet og `issuerRef:` som er instillingene for issueren som skal brukes. Ellers kan flere av feltene fjernes. Blant annet de som ikke brukes av `dnsNames:`, `ipAddresses:` og `uriSANs:` og `commonName:` som helst ikke skal brukes lenger.

Hvis vi skulle laget et certificate til webserveren fra forrige kapittel kan det da se slik ut:

5_4_cert-manager\nginx-cert.yaml

```yaml
apiVersion: cert-manager.io/v1alpha2
kind: Certificate
metadata:
  name: test-k-torset-me
  namespace: testing
spec:
  secretName: test-k-torset-me-tls
  duration: 2160h # 90d
  renewBefore: 360h # 15d
  organization:
  - bach
  isCA: false
  keySize: 2048
  keyAlgorithm: rsa
  keyEncoding: pkcs1
  usages:
    - server auth
    - client auth
  dnsNames:
  - test.k.torset.me
  issuerRef:
    name: letsencrypt-staging
    kind: ClusterIssuer
```

Vi fylte ut `name:` og `kind:` under `issuerRef:`. `group:` er ikke nødvendig å ha med om vi ikke bruker en ekstern issuer. Vi fjernet `commonName:`, `ipAddresses:` og `uriSANs:` siden vi ikke bruker de og endret `dnsNames:` til dns-navnet vi ønsker å bruke. Vi endret også navnet på sertifikatressursen og Secreten som vil bli laget og la det i samme Namespace som nginx-Deploymenten. Vi kan nå legge til dette sertifikatet med `kubectl apply -f <filnavn.yaml>`.

```
magnus@k8s-master:/opt/kubernetes/cert-manager$ kubectl get cert
NAME               READY   SECRET                AGE
test-k-torset-me   True    test-k-torset-me-tls   26m
test-tls           True    test-tls               22h
magnus@k8s-master:/opt/kubernetes/cert-manager$ kubectl get cr
NAME                          READY   AGE
test-k-torset-me-2107850554   True    26m
test-tls-3066562019           True    21h
magnus@k8s-master:/opt/kubernetes/cert-manager$ kubectl get secrets
NAME                          TYPE                                  DATA   AGE
cloudflare-api-token-secret   Opaque                                1      13d
default-token-qssbf           kubernetes.io/service-account-token   3      33d
test-k-torset-me-tls          kubernetes.io/tls                     3      23h
test-tls                      kubernetes.io/tls                     3      22h
magnus@k8s-master:/opt/kubernetes/cert-manager$
```

Vi ser det ble laget en certificate-ressurs “test-k-torset-me” og en certificate request “test-k-torset-me-2107850554” og til slutt en Secret “test-k-torset-me-tls”. Denne Secreten inneholder seritfikatet og kan gis til en Pod om det er en tjeneste på innsiden av en po som skulle trenge det eller brukes i en Ingress. For å bruke denne Secreten i en Ingress lager vi en ny Ingress-ressurs.

5_4_cert-manager\test-tls-ingress.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-tls-ingress
  annotations:
    kubernetes.io/ingress.class: ingress-nginx
spec:
  tls:
    - hosts:
      - test.k.torset.me
      secretName: test-k-torset-me-tls
  rules:
  - host: test.k.torset.me
    http:
      paths:
      - path: /
        backend:
          serviceName: nginx-deployment
          servicePort: 80
status:
  loadBalancer: {}
```

Vi ser at denne er veldig lik Ingressen fra forrige kapittel, men vi har fjernet annotationen som sier at cert-manager skal lage sertifikatet selv og vi har endret `secretName:` til sertifikatet vi nettopp lagde. Hvis vi legger til denne Ingressen og går til test.k.torset.me vil vi se at vi har fått tildelt et sertifikat fra “Fake LE Intermediate X1” siden vi brukte letsencrypt-

staging issueren. For å oppgradere dette til et gyldig sertifikat trenger vi bare endre på certificate-ressursen. Dette gjør vi med kommandoen `kubectl edit cert <certificate-navn>`. Her trenger vi bare endre `name:` under `isserRef:` til letsencrypt-prod for så å lagre og gå ut av editoren. Certificate-ressursen skal se slik ut:

```
# Please edit the object below. Lines beginning with a '#' will be
ignored,
# and an empty file will abort the edit. If an error occurs while saving
this file will be
# reopened with the relevant failures.
#
apiVersion: cert-manager.io/v1alpha3
kind: Certificate
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"cert-
manager.io/v1alpha2","kind":"Certificate","metadata":{"annotations":{},"
name":"test-k-torset-
me","namespace":"testing"},"spec":{"dnsNames":["test.k.torset.me"],"dura
tion":"2160h","isCA":false,"issuerRef":$
  creationTimestamp: "2020-04-16T09:18:08Z"
  generation: 1
  name: test-k-torset-me
  namespace: testing
  resourceVersion: "14042066"
  selfLink: /apis/cert-
manager.io/v1alpha3/namespaces/testing/certificates/test-k-torset-me
  uid: d6595c9b-f8fd-46b0-a095-0910eae40e15
spec:
  dnsNames:
  - test.k.torset.me
  duration: 2160h0m0s
  issuerRef:
    kind: ClusterIssuer
    name: letsencrypt-prod
  keyAlgorithm: rsa
  keyEncoding: pkcs1
  keySize: 2048
  renewBefore: 360h0m0s
  secretName: test-k-torset-me-tls
  subject:
    organizations:
    - bach
  usages:
  - server auth
```

```
  - client auth
status:
  conditions:
  - lastTransitionTime: "2020-04-16T09:47:36Z"
    message: Certificate is up to date and has not expired
    reason: Ready
    status: "True"
    type: Ready
  notAfter: "2020-07-15T08:47:35Z"
```

Når vi lagrer nå vil det lages en ny certificate request som oppdaterer Secreten med sertifikatet og den er allerede lagt inn i Ingressen så vi trenger ikke gjøre noe mer. Hvis vi går til test.torset.me nå vil vi ha et gyldig sertifikat.

6. Sette opp tjenester med Kubernetes

I dette kapittelet vil vi vise praktisk bruk av Kubernetes. Dette er eksempler på enkle tjenester eller vanlige use caser for Kubernetes, og vil vise hvordan noen av ressursene vi har gått igjennom kan brukes i praksis

6.1 Webserver med autoskalering

Dette eksempelet går gjennom hvordan man kan bygge et image ut fra eksisterende kode og rulle det ut i Kubernetes. Det vil også vise hvordan man kan bruke Kubernetes til å automatisk skalere opp og ned en applikasjon etter behov.

6.1.1 Bygge Image

For å demonstrere hvordan man setter opp en webserver i Kubernetes har vi laget en enkel webserver som henter en melding fra en MongoDB database. Meldingen er basert på stien i adressen:

6_1_web-server/index.js

```
const express = require('express');
const mongoose = require('mongoose');
const Message = require('./message.js');
const { user, pwd } = require('./dbconf.json');

const port = 5000;
const mongo_url = `mongodb://${user}:${pwd}@10.100.39.31:27017/web-server`;

const app = express()
    .disable('X-Powered-By')
    .set('view engine', 'pug')

    .get('/', (req, res) => {
        Message.findOne({ id: 0 }).then( (response) => {
            res.render('index', { message: response.message });
        });
    })

    .get('/hello', (req, res) => {
        Message.findOne({ id: 1 }).then( (response) => {
            res.render('index', { message: response.message });
        });
    })

    .get('/test', (req, res) => {
        Message.findOne({ id: 2 }).then( (response) => {
            res.render('index', { message: response.message });
        });
    })
```

```
    .get('/goodbye', (req, res) => {
        Message.findOne({ id: 3 }).then( (response) => {
            res.render('index', { message: response.message });
        });
    });

mongoose.connect(mongo_url, { useCreateIndex: true, autoIndex: false,
useNewUrlParser: true, useUnifiedTopology: true })
    .then( () => {
        console.log('Connected to database!');
        app.listen(port, (err) => {
            if (err) {
                console.error(err);
                console.error(err.stack);
                process.exit(1);
            }
            console.log(`Running on port: ${port}!`);
        });
    })
    .catch( (err) => {
        console.error(`Could not connect to database: ${err}`);
        process.exit(1);
    });
```

For at denne skal kunne rulles ut i en container, må vi først bygge et docker-image. Da trenger vi en dockerfile som forteller hvordan imaget skal se ut.

6_1_web-server/Dockerfile

```
FROM node:latest

WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install --production
COPY . .

CMD [ "npm", "start" ]
```

`FROM` forteller hva imaget skal baseres på. Dette er en Node.js applikasjon, og vi baserer derfor imaget på det nyeste node imaget.
`WORKDIR` forteller hvilken mappe vi vil at applikasjonen skal ligge i inne i containeren, og `COPY` kopierer package.json og package-lock.json til containeren. Disse filene brukes i Node.js applikasjoner for å fortelle hvilke pakker applikasjonen er avhengig av.
`RUN` kjører npm install, som sjekker disse filene og installerer de nødvendige pakkene. --production flagget sier at vi bare skal installere de pakkene som er nødvendig for produksjon
`COPY` kopierer resten av kildekoden over i containeren. Vi har også laget en `.dockerignore` fil, som forteller hvilke filer og mapper docker skal overse når filene kopieres. I Node.js

applikasjoner er det vanlig å legge mappen `node_modules` her. Den inneholder alle de installerte pakkene; ofte mer enn det som er nødvendig i containeren.
Til slutt forteller `CMD` hvilken kommando som skal kjøres for å starte applikasjonen, i dette tilfellet `npm start`

Nå kan imaget bygges ved å kjøre følgende kommando i rotmappen til applikasjonen. Vi må også legge imaget i et Docker-registry for at det skal kunne brukes i Kubernetes. Vi gir det en passende tag, og laster den opp:

```
docker build -t docker.torset.me/web-server:1.1 .
docker push docker.torset.me/web-server:1.1
```

For at Kubernetes skal kunne hente imaget må vi lage en Secret som inneholder autentiseringsinformasjon til registryet. Først må vi logge inn i registryet selv:

```
docker login <registry-navn>
```

Denne kommandoen tar inn brukernavn og passord og oppretter filen `$HOME/.docker/config.json`, som inneholder en autentiseringsnøkkel til registryet. Vi bruker denne filen til å opprette en Secret som vi kaller `regcred`:

```
kubectl create secret generic regcred \
    --from-file=.dockerconfigjson=$HOME/.docker/config.json \
    --type=kubernetes.io/dockerconfigjson
```

6.1.2 Utrulling i Clusteret

Når imaget er klart kan vi lage en Deployment for å rulle ut webserveren i clusteret:

6_1_web-server/k8s/deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: web-server
spec:
    replicas: 1
    selector:
        matchLabels:
            app: web
    template:
        metadata:
            labels:
                app: web
        spec:
            containers:
```

```
        - name: web-server
          image: docker.torset.me/web-server:1.1
          ports:
          - containerPort: 5000
        imagePullSecrets:
        - name: regcred
```

Her lager vi en Deployment som heter `web-server` med labelet `app: web`. Vi sørger også for å oppgi imaget vi har laget. Seksjonen `imagePullSecrets:` inneholder Secreten `regcred` som vi laget tidligere. Dette er informasjonen Kubernetes bruker for å hente imaget. Nå har vi rullet ut webserveren i Kubernetes, men den kan ikke nås enda. For det trenger vi en Service:

6_1_web-server/k8s/service.yaml

```
apiVersion: v1
kind: Service
metadata:
    name: web-server
spec:
    type: ClusterIP
    selector:
        app: web
    ports:
    - port: 5000
      targetPort: 5000
```

Dette er en Service av typen ClusterIP, som sender trafikk fra port 5000 på Servicen til port 5000 på alle Podder med label `app: web`.
Med en Service kan applikasjonen nås internt i clusteret, men den kan enda ikke nås fra utsiden. Det siste vi trenger for å få til dette er en Ingress:

6_1_web-server/k8s/ingress.yaml

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
    name: web-server
spec:
    rules:
    - host: web.example.com
      http:
          paths:
          - backend:
              serviceName: web-server
              servicePort: 5000
```

Denne vil rute all trafikk som kommer inn til Ingress-Controlleren på domenenavnet `web.example.com` til port 5000 i Servicen vår.

Nå er applikasjonen oppe å kjører i Kubernetes, og det er mulig å nå den fra utsiden på adressen web.example.com:

Tjenesten viser kun en enkel side med en melding hentet fra databasen. De ulike linkene henter en ny melding. Vi kan også se navnet på Poden vi kjører på. Hvis vi har flere Poder vil dette endres avhengig av hvilken av dem vi blir sendt til.

6.1.3 Skalering og Oppgradering

Webservere er en av de tjenestene som kan dra størst nytte av fordelene fra Kubernetes, da spesielt lastbalansering og skalering. Webservere trenger som regel ikke noe data selv, og kommuniserer i stedet med en ekstern database slik som applikasjonen vår gjør. Da kan man kjøre tjenesten i flere Poder uten å måtte tenke på synkronisering av data, noe som gjør det enkelt å ta i bruk autoskalering. Dette krever en Metrics Server, og vil skalere tjenesten basert på forhåndsdefinerte grenser for ressursbruk. For at dette skal fungere må vi først spesifisere disse grensene i Deploymenten vår:

6_1_web-server/k8s/deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: web-server
spec:
    replicas: 1
    selector:
        matchLabels:
            app: web
    template:
        metadata:
            labels:
                app: web
        spec:
            containers:
            - name: web-server
              image: docker.torset.me/web-server:1.1
              ports:
              - containerPort: 5000
              resources:
                limits:
                    cpu: 500m
                requests:
                    cpu: 200m
            imagePullSecrets:
            - name: regcred
```

Tjenesten har to forskjellige ressurs parametre; limits som er den øvre grensen tjenesten kan bruke, og requests som er minstekravet for ressurser tjenesten vil ha. CPU-bruk i Kubernetes måles i kjerner, altså er `500m`, eller 500 millikjerner, det samme som 50% av en enkelt CPU-kjerne.

Vi må også lage en HPA (Horizontal Pod Autoscaler), som er den ressursen som tar seg av selve skaleringen. Den kan se slik ut:

6_1_web-server/k8s/hpa.yaml

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
    name: web-autoscaler
spec:
    scaleTargetRef:
        apiVersion: apps/v1
        kind: Deployment
        name: web-server
    minReplicas: 1
    maxReplicas: 10
    targetCPUUtilizationPercentage: 50
```

Her velger vi hva som skal skaleres under `scaleTargetRef`, og setter at tjenesten skal ha minimum 1 opp til 10 replikaer. `targetCPUUtilizationPercentage` sier at når en Pod når 50% av grenseverdien sin skal det lages nye replikaer.
Nå har vi satt opp skalering for applikasjonen vår. Når trafikken øker nok vil det lages flere Poder, og lasten vil fordeles mellom dem.

En annen fordel ved Kubernetes er ConfigMaps. De er mye brukt for å konfigurere ferdige containere, og gjør det mulig å flytte eller endre hele eller deler av infrastrukturen uten å være nødt til å endre selve container-imaget. Man vil også ha mulighet til å flytte ConfigMapet for å bruke den samme konfigurasjonen flere steder.
For webserveren vår kan vi for eksempel legge port og url til databasen i et ConfigMap. Da kan den enkelt tilpasses andre installasjoner uten at man trenger å endre koden:

6_1_web-server/k8s/configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: web-server
data:
    PORT: "5000"
    DB_URL: "mongodb://dbadmin:QE7!tRu!@10.100.39.31:27017/web-server"
```

For å gi tjenesten tilgang til disse variablene legger vi til en `envFrom` seksjon i Deploymenten vår:

6_1_web-server/k8s/deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: web-server
spec:
    replicas: 1
    selector:
        matchLabels:
            app: web
    template:
        metadata:
            labels:
                app: web
        spec:
            containers:
            - name: web-server
              image: docker.torset.me/web-server:1.1
              ports:
              - containerPort: 5000
              resources:
                limits:
                    cpu: 500m
                requests:
                    cpu: 200m
              envFrom:
              - configMapRef:
                    name: web-server
            imagePullSecrets:
            - name: regcred
```

Disse kan hentes inn og brukes i koden til applikasjonen. Hvordan det gjøres vil avhenge av språket som brukes, men i vår applikasjon kan det gjøres slik:

6_1_web-server/index.js

```javascript
const express = require('express');
const mongoose = require('mongoose');
const Message = require('./message.js');
const port = process.env.PORT || 5000;
const mongo_url = process.env.DB_URL ||'mongodb://localhost:27017/web-server';

const app = express()
    .disable('X-Powered-By')
    .set('view engine', 'pug')
```

Vi endrer de variablene vi har satt for port og database URL. `process.env` henter ut miljøvariabler fra containeren. Vi har også satt verdier som vi kan falle tilbake på dersom miljøvariablene er tomme. Den database-URLen vi faller tilbake på her er en database på lokal maskin, som er en vanlig løsning brukt i testing og under utvikling. Vi har også fjernet brukernavn og passord til databasen, som nå ligger i ConfigMapet vårt. Dermed kan man enkelt gjøre endringer i databasen uten at man trenger å tenke på dette i koden.

For å ta i bruk disse endringene må vi oppdatere imaget vårt. Her kan vi ta i bruk enda en stor fordel ved Kubernetes; rullende oppdateringer. Når det gjøres en endring i en ressurs som krever omstart i en Pod, vil det med en gang lages en ny Pod med de nye innstillingene. Den gamle Poden tas ikke ned før den nye er klar, slik at tjenesten er tilgjengelig under hele oppdateringen. Dette gjøres for alle Podene det gjelder, en etter en, og antall aktive Poder holder seg konstant. Hvis man forsøker å nå tjenesten under oppdatering, vil noen nå den gamle versjonen, mens noen vil kunne nå den nye.
Når koden er oppdatert må vi først bygge et image for den nye versjonen og laste det opp til registryet:

```
docker build -t docker.torset.me/web-server:1.2 .
docker push docker.torset.me/web-server:1.2
```

Nå trenger vi kun å endre image tag i Deploymenten vår til den nye versjonen:

6_1_web-server/k8s/deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: web-server
spec:
    replicas: 1
    selector:
        matchLabels:
            app: web
    template:
        metadata:
            labels:
                app: web
        spec:
            containers:
            - name: web-server
              image: docker.torset.me/web-server:1.2
              ports:
              - containerPort: 5000
              resources:
                limits:
                    cpu: 500m
                requests:
                    cpu: 200m
              envFrom:
              - configMapRef:
                    name: web-server
            imagePullSecrets:
            - name: regcred
```

Kubernetes vil hente ned det nye imaget og opprette en ny Pod. Den gamle forblir aktiv helt til den nye er klar, og tjenesten vil være tilgjengelig gjennom hele oppdateringen.

6.2 Storage Provisioner og Statefulset

Containere som trenger lagringsplass, slik som databaser, er mer kompliserte enn en enkel webserver. For det første må denne lagringsplassen tildeles, og Poden må vite hvor den finner sine filer etter omstart. For det andre må dataen synkroniseres mellom flere instanser dersom man vil dra nytte av skalering og replikaer. I dette eksempelet vil vi vise hvordan man kan sette opp en Storage Provisioner for automatisk tildeling av lagring, og vi vil bruke en MySQL database for å vise praktisk bruk av et Statefulset. Når man har en database i Kubernetes brukes en master-slave ordning, hvor kun master kan skrive data. Slavene kan kun lese, og synkroniserer seg etter masteren. Dette er mulig fordi det er betydelig mer lese- enn skrive-trafikk. Store databaser blir ofte veldig komplekse med mye trafikk mellom Podene, slik at det ikke er anbefalt å kjøre disse i Kubernetes

6.2.1 Storage Provisioner

Når man kjører Kubernetes på en skyplattform vil tildeling av lagringsplass automatisk håndteres av skyleverandøren. I en bare-metal installasjon slik som vår må man løse dette selv. Kubernetes community har laget en tjeneste som tildeler plass på en NFS server etter hvert som Poder ber om det. [11] Vi bruker en NFS server som vi har satt opp på forhånd, og vi vil ikke gå gjennom oppsett av denne her. For å installere Storage Provisioneren i clusteret vårt må vi hente et sett med ressursfiler fra Github:

```
git clone https://github.com/kubernetes-incubator/external-storage.git
cd external-storage/nfs-client/deploy
```

Det første vi gjør er å kjøre en kommando som endrer Namespacet i alle ressursfilene. Vi skiller tjenesten ut i et eget Namespace fordi dette er en system-tjeneste, som vi vil holde atskilt fra vanlige applikasjoner. Vi aktiverer også `rbac.yaml`, som setter opp autorisasjon for tjenesten:

```
sed -i'' "s/namespace:.*/namespace: nfs-provisioner/g" \
  ./rbac.yaml ./deployment.yaml
kubectl apply -f rbac.yaml
```

Vi må også konfigurere selve provisioneren til å bruke NFS-serveren vår. Vi endrer filen `deployment.yaml` før vi ruller den ut i clusteret:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfs-client-provisioner
  labels:
    app: nfs-client-provisioner
  # replace with namespace where provisioner is deployed
  namespace: nfs-client-provisioner
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nfs-client-provisioner
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          image: quay.io/external_storage/nfs-client-provisioner:latest
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: nfs-server
            - name: NFS_SERVER
              value: 10.100.39.28
            - name: NFS_PATH
              value: /mnt/export
      volumes:
        - name: nfs-client-root
          nfs:
            server: 10.100.39.28
            path: /mnt/export
```

Her har vi endret env-seksjonen til å matche vår NFS-server, og vi har gitt provisioneren et enklere navn. Vi har også endret nfs-seksjonen under volumes, slik at provisioneren får tildelt riktig lagringsområde. Når disse endringene er gjort, lagrer vi filen og ruller ut provisioneren med `kubectl apply -f deployment.yaml`

Det siste vi trenger for Storage Provisioneren er en StorageClass, som definerer en lagringstype som kan deles ut i clusteret vårt. Denne finner vi i `class.yaml`, og her må vi også gjøre noen endringer:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-storage
provisioner: nfs-server
parameters:
  archiveOnDelete: "false"
```

Her har vi endret navnet på selve StorageClassen, og vi har endret `provisioner:` til å matche det navnet vi satt på provisioneren tidligere. For å aktivere StorageClassen bruker vi:

```
kubectl apply -f class.yaml
```

Når vi spesifiserer denne StorageClassen i et PersistentVolumeClaim vil Storage Provisioneren automatisk opprette et PersistentVolume og tildele det til Poden det gjelder. Vi kan også opprette flere StorageClasser med ulike parametre og egenskaper etter hvert som det er nødvendig. Kun de StorageClassene som refererer til denne Provisioneren vil kunne opprettes automatisk.

6.2.2 Oppsett av MySQL med StatefulSet

Som nevnt over vil vi sette opp en MySQL database med en master-slave ordning. Hver Pod vil inneholde to containere, én MySQL container som kjører selve databasen og en sidevogn som tar seg av backup og synkronisering. [12]

6.2.2.1 ConfigMap og Service

Det første vi vil gjøre er å sette opp konfigurasjonen for Podene. I et ConfigMap setter vi opp konfigurasjon for master og slave.

6_2_mysql/configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
  master.cnf: |
    # Apply this config only on the master.
    [mysqld]
    log-bin
  slave.cnf: |
    # Apply this config only on slaves.
    [mysqld]
    super-read-only
```

master.cnf vil gjelde for master-poden, og vil la den gi ut replikasjons-logger til slavene, mens slavene settes til read-only. ConfigMapet i seg selv gjør ikke at Podene kjører som master eller slave. I StatefulSetet senere vil vi sette opp hvordan Podene skal velge hvilken rolle de har.
Vi vil også opprette to Servicer for databasen. Den ene er uten IP, og skal kun gi stabile oppføringer i den interne DNSen i clusteret. Den andre kobler til hvilken som helst av Podene, og er for bruk ved lese-forespørsler. For å skrive må man koble direkte til master-poden:

6_2_mysql/service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
```

```
    port: 3306
  clusterIP: None
  selector:
    app: mysql
---
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  selector:
    app: mysql
```

Vi aktiverer ConfigMapet og de to Servicene med `kubectl apply -f configmap.yaml -f service.yaml`

6.2.2.2 StatefulSet

Nå kan vi lage et StatefulSet for å opprette selve database-podene. StatefulSetet bruker init-containere, og har i tillegg to containere i hver Pod, slik at filen blir veldig lang. Derfor har vi bare med deler av filen her. Hele filen ligger vedlagt under *`6_2_mysql/statefulset.yaml`*

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: mysql
  replicas: 3
```

Den første delen ligner på en Deployment. Forskjellen her er at vi også definerer en Service som er knyttet til StatefulSetet. Poder i et StatefulSet blir navngitt med navnet på StatefulSetet kombinert med indexen til Poden. Siden vi lager tre Poder her, vil de få navn `mysql-0`, `mysql-1` og `mysql-2`.

De to containerne som kjøres i hver Pod er en MySQL container, som er selve databasen, og en Percona Xtrabackup container, som er en populær programvare for backup av MySQL databaser. Det første vi skal se på er init-containerne. De kjører ved oppstart for å sette den nødvendige konfigurasjonen og synkronisere data. De kjører til de har gjort ferdig oppgaven

sin, og blir byttet ut med de faktiske containerne etterpå. Poden kan ikke gå videre før alle init-containerne er ferdig.

Det som er viktig å merke seg ved disse er command-seksjonen. For `init-mysql` ser den slik ut:

```
set -ex
[[ `hostname` =~ -([0-9]+)$ ]] || exit 1
ordinal=${BASH_REMATCH[1]}
echo [mysqld] > /mnt/conf.d/server-id.cnf
echo server-id=$((100 + $ordinal)) >> /mnt/conf.d/server-id.cnf
if [[ $ordinal -eq 0 ]]; then
  cp /mnt/config-map/master.cnf /mnt/conf.d/
else
  cp /mnt/config-map/slave.cnf /mnt/conf.d/
fi
```

Her sjekker vi indexen til Poden for å hente riktig konfigurasjon fra ConfigMapet vi laget tidligere. Hvis den er 0 blir Poden master, hvis ikke blir den en slave.
Den andre init-containeren heter `clone-mysql` og har følgende command-seksjon

```
set -ex
[[ -d /var/lib/mysql/mysql ]] && exit 0
[[ `hostname` =~ -([0-9]+)$ ]] || exit 1
ordinal=${BASH_REMATCH[1]}
[[ $ordinal -eq 0 ]] && exit 0
ncat --recv-only mysql-$(($ordinal-1)).mysql 3307 | xbstream -x -C
/var/lib/mysql
xtrabackup --prepare --target-dir=/var/lib/mysql
```

Denne vil først sjekke om det allerede finnes data i Podens PersistentVolume. Hvis det ikke gjør det, og Poden ikke har index 0, kopieres data fra Poden som er før den i rekken. Altså vil `mysql-1` kopiere fra `mysql-0`, `mysql-2` vil kopiere fra `mysql-1` osv.

Når begge init-containerne er ferdig, vil de faktiske containerne lages. Det som er verdt å merke seg ved mysql-containeren her er at den har en `livenessProbe` og en `readinessProbe`, som brukes for å sjekke når den har startet, og når den er klar:

```
livenessProbe:
    exec:
    command: ["mysqladmin", "ping"]
    initialDelaySeconds: 30
    periodSeconds: 10
    timeoutSeconds: 5
readinessProbe:
    exec:
```

```
    command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]
    initialDelaySeconds: 5
    periodSeconds: 2
    timeoutSeconds: 1
```

Backup-containeren har igjen en command-seksjon:

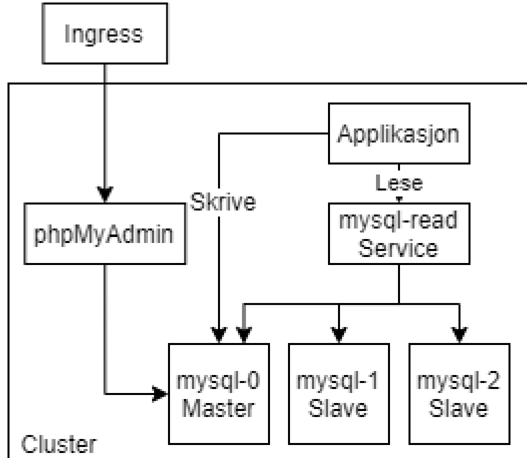
```
set -ex
cd /var/lib/mysql

if [[ -f xtrabackup_slave_info && "x$(<xtrabackup_slave_info)" != "x" ]];
then
  cat xtrabackup_slave_info | sed -E 's/;$//g' > change_master_to.sql.in
  rm -f xtrabackup_slave_info xtrabackup_binlog_info
elif [[ -f xtrabackup_binlog_info ]]; then
  [[ `cat xtrabackup_binlog_info` =~ ^(.*?)[[:space:]]+(.*?)$ ]] || exit 1
  rm -f xtrabackup_binlog_info xtrabackup_slave_info
  echo "CHANGE MASTER TO MASTER_LOG_FILE='${BASH_REMATCH[1]}',\
        MASTER_LOG_POS=${BASH_REMATCH[2]}" > change_master_to.sql.in
fi

if [[ -f change_master_to.sql.in ]]; then
  echo "Waiting for mysqld to be ready (accepting connections)"
  until mysql -h 127.0.0.1 -e "SELECT 1"; do sleep 1; done

  echo "Initializing replication from clone position"
  mysql -h 127.0.0.1 \
        -e "$(<change_master_to.sql.in), \
                MASTER_HOST='mysql-0.mysql', \
                MASTER_USER='root', \
                MASTER_PASSWORD='', \
                MASTER_CONNECT_RETRY=10; \
            START SLAVE;" || exit 1
  mv change_master_to.sql.in change_master_to.sql.orig
fi

exec ncat --listen --keep-open --send-only --max-conns=1 3307 -c \
"xtrabackup --backup --slave-info --stream=xbstream --host=127.0.0.1 --
user=root"
```

Her sjekker vi først om vi kloner fra en slave eller direkte fra master. Hvis vi er på en slave settes den opp til å synkronisere endringer fra Poden før seg i rekken. Det startes også en server som vil sende backup-info til andre Poder som spør etter det.

Til slutt i filen har vi `volumeClaimTemplates`. Dette er en mal for de PersistentVolumeClaim som vil opprettes for hver av Podene. Her bruker vi samme StorageClass som vi opprettet tidligere.

```
volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      storageClassName: nfs-storage
      resources:
        requests:
          storage: 10Gi
```

Som nevnt over bruker vi Servicen `mysql-read` for lese-forespørsler til databasen. For å skrive bruker vi i stedet masteren direkte; `mysql-0.mysql`. Her er `mysql-0` DNS oppføringen til Poden. Siden Poden inneholder to containere bruker vi også `.mysql` til å velge riktig container innad i Poden. Disse er interne DNS oppføringer, og er kun tilgjengelig på innsiden av clusteret. Det finnes ingen effektiv måte å gjøre databasen tilgjengelig fra utsiden, slik at den er ment for å brukes av applikasjoner som kjører i samme cluster. Dersom man har bruk for ekstern administrasjon av databasen gjøres dette best gjennom en PhpMyAdmin Pod, som kan gjøres tilgjengelig gjennom en Ingress. Diagrammet viser hvordan oppsettet kan se ut:



Figur 6.1: Lese-forespørsler går gjennom en Service, mens skrive-forespørsler går direkte til master

6.3 Oppsett av utviklingsmiljø

I dette eksempelet vil vi vise hvordan man kan sette opp et utviklingsmiljø i Kubernetes. Dette inkluderer GitLab med container-registry for versjonskontroll og lokal lagring av Docker images og Jenkins for bygging og testing av ny kode. Sammen vil dette gi et komplett utviklingsmiljø hvor ny kode automatisk blir testet og lagt ut i produksjon

6.3.1 GitLab med Registry

GitLab er en tjeneste à la GitHub, men med mulighet for å drifte hele systemet selv på ditt eget hardware. Det er basert på Git som er et versjonskontrollsystem for å holde styr på prosjekter og endringer. Det er spesielt mye brukt til programmering, men kan også brukes til andre ting som dokumentasjon og feilrapportering. Siden du drifter det selv gir det deg gode muligheter for integrasjon med lokale tjenester du ikke ønsker å gjøre tilgjengelig over internett. Det gir deg også full kontroll over brukere og grupper og kan integreres med blant annet LDAP. Vi tenkte dette var en typisk tjeneste å drifte selv og ønsker å vise hvordan dette kan gjøres med Kubernetes.

Installasjonen vi bruker inneholder også et Docker Registry. Et Docker Registry gir deg mulighet til å lagre Docker imager lokalt til bruk i blant annet Kubernetes. Disse imagene integereres med repositoryene i GitLab og det er også en SSO-tjeneste inkludert som gjør at du bruker samme brukernavn og passord på både Docker Registry og GitLab.

6.3.1.2 Installasjon

For å installere GitLab bruker vi Helm. [17] Helm chartet inneholder alt vi trenger pluss litt til for å få satt opp GitLab. Chartet er laget av GitLab selv og ligger i deres eget repo på Helm. Vi må derfor legge til repoet med `helm repo add gitlab https://charts.gitlab.io/`. Det er noen verdier vi må endre på før vi kan installere så vi laster ned values-filen til chartet med `helm show values gitlab/gitlab > gitlab.yaml`. Vi lager vår egen fil, values.yaml for å overskrive standardinnstillingene med.

Registry er nødt til å ha TLS for å fungere så vi er nødt til å bruke et domenenavn vi eier og kan skaffe sertifikater til, selv om vi ikke skal eksponere tjenesten til internett. Vi bruker k.torset.me. De forskjellige tjenestene vil få et subdomene til dette domenet. Altså vil gitlab bli på gitlab.k.torset.me og registry på registry.k.torset.me. Vi legger dette til i values-filen med:

6_3_1_gitlab_med_registry\values.yaml

```yaml
global:
  hosts:
    domain: k.torset.me
```

Vi har allerede installert cert-manager så vi sier at cert-manager ikke skal installeres eller konfigureres og at installasjonen skal bruke issueren vi allerede har satt opp med:

```yaml
# 6_3_1_gitlab_med_registry\values.yaml
global:
  ingress:
    configureCertmanager: false
    annotations:
      kubernetes.io/tls-acme: true
      cert-manager.io/cluster-issuer: letsencrypt-prod
certmanager:
  install: false
```

Vi ønsker heller ikke å installere prometheus så vi legger til

```yaml
# 6_3_1_gitlab_med_registry\values.yaml
prometheus:
  install: false
```

Hvis vi slår alle disse sammen ender vi opp med en values.yaml-fil som ser slik ut:

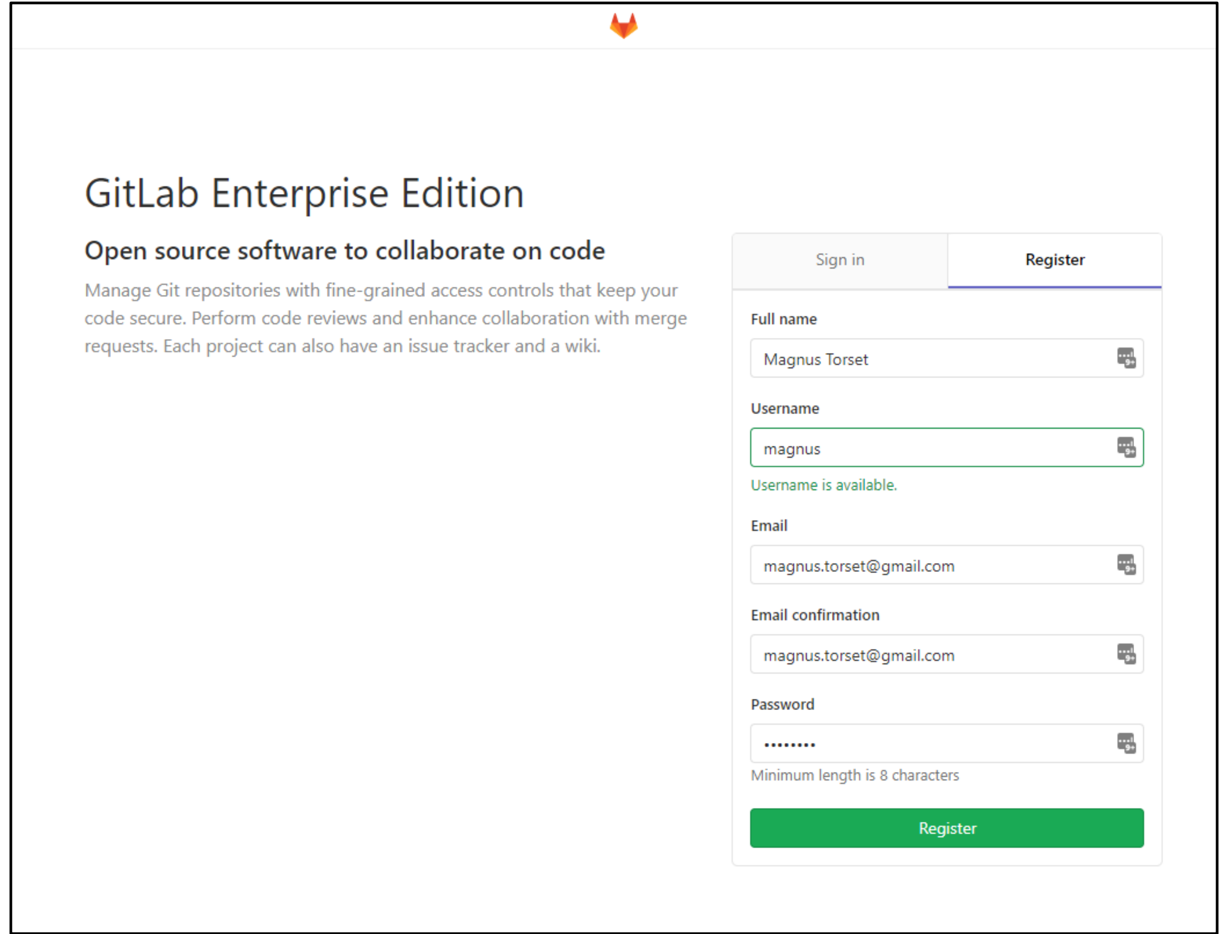
```yaml
# 6_3_1_gitlab_med_registry\values.yaml
global:
  ingress:
    configureCertmanager: false
    annotations:
      kubernetes.io/tls-acme: true
      cert-manager.io/cluster-issuer: letsencrypt-prod
  hosts:
    domain: k.torset.me

prometheus:
  install: false
certmanager:
  install: false
```

Vi er nå klare til å installere med `helm install <deploymentnavn> gitlab/gitlab -f values.yaml`. Det er mye som skal installeres så installasjonen kan ta litt tid. Vi må også vente på å få tildelt sertifikat som også kan ta litt tid.

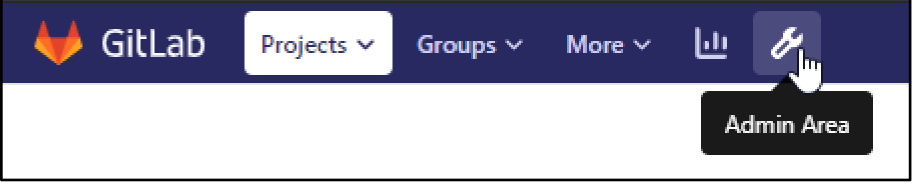
6.3.1.3 Konfigurasjon av GitLab

Lage administratorbruker

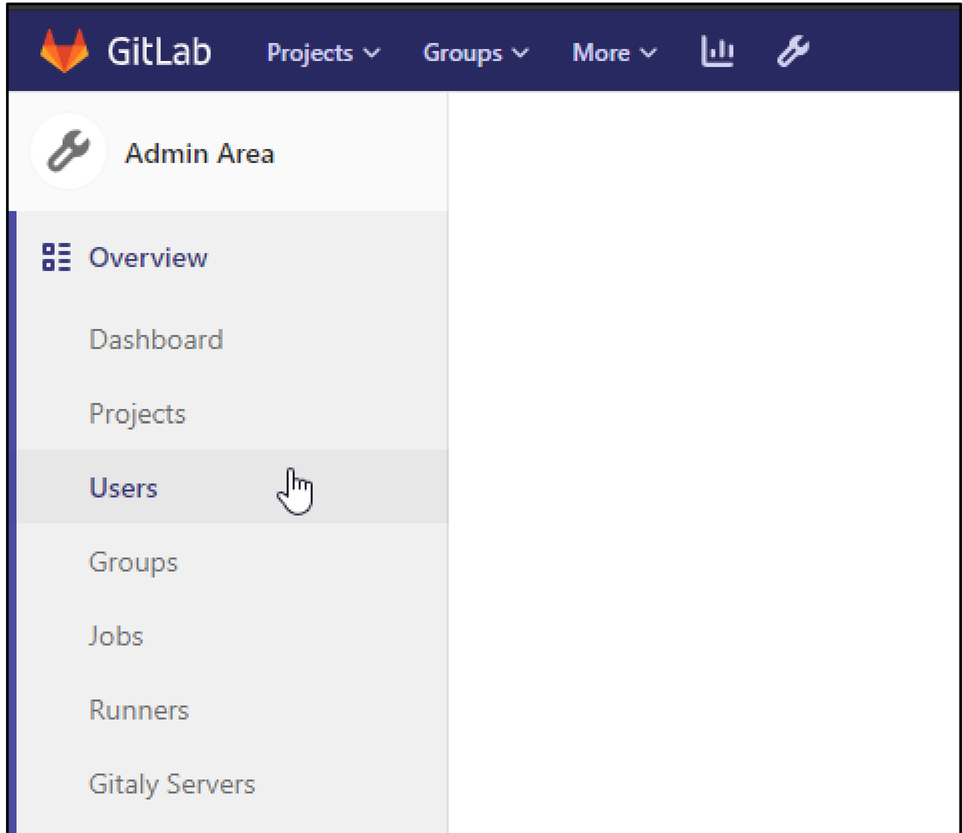
Etter installasjonen er ferdig kan vi gå til gitlab.k.torset.me og lage en bruker ved å klikke “Register” og fylle inn feltene.



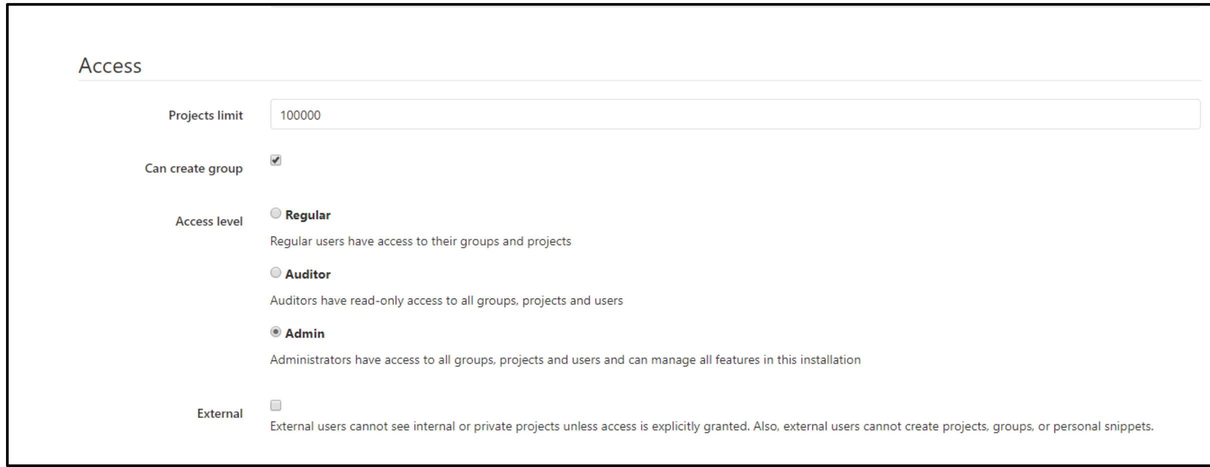
Etter vi har laget en bruker kan vi logge ut igjen og logge inn som administrator med brukernavn root og hente passordet fra Kubernetes med kommandoen `kubectl get secret <helm-installasjonsnavn>-gitlab-initial-root-password -ojsonpath='{.data.password}' | base64 --decode ; echo`. For å komme til administratorområdet klikker vi på skiftenøkkelen øverst i venstre hjørne.



Herfra går vi til “Users” i sidepanelet.



Her klikker vi edit på brukeren vi laget og gir den adminrettigheter.

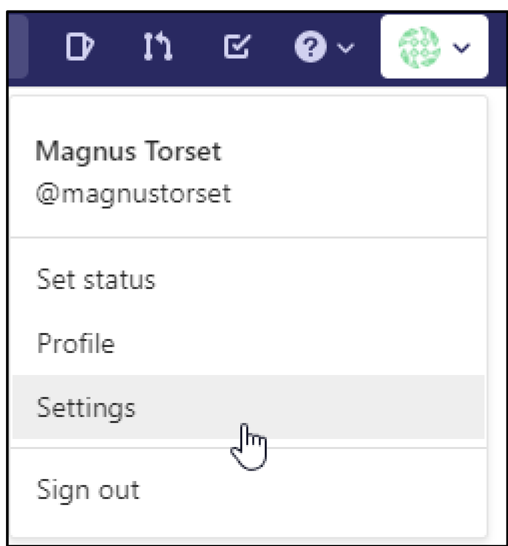


Nå kan vi lagre og logge inn igjen som vår egen bruker som nå skal være administrator.

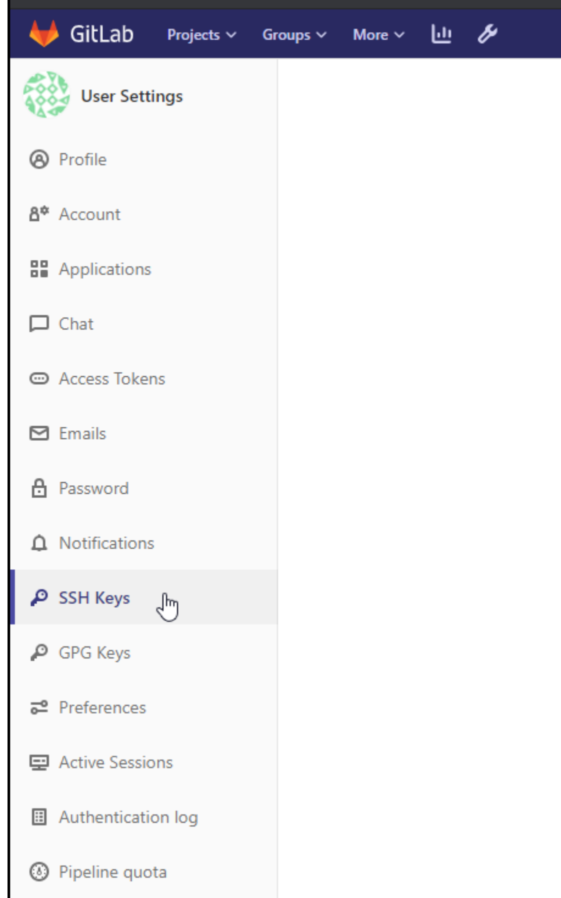
Sette opp SSH-nøkkel for en bruker

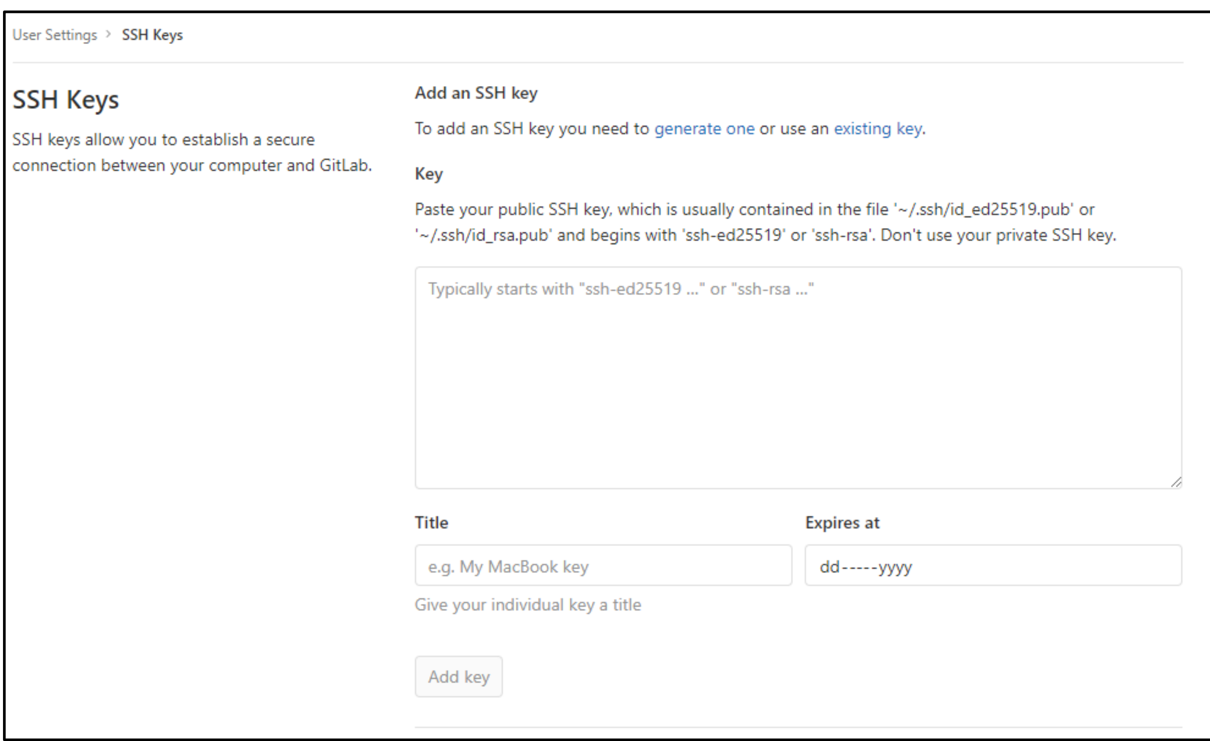
For å gjøre det enkelt å bruke Git er det anbefalt å sette opp en SSH-nøkkel så vi slipper å skrive inn brukernavn og passord hver gang vi skal gjøre noe med et repository. Dette må gjøres på alle maskiner vi ønsker å bruke med GitLab. For å lage en SSH-nøkkel kjører vi `cd ~/.ssh` for å komme oss til SSH-mappen for brukeren vår. Deretter skriver vi kommandoen `ssh-keygen -t ed25519`. Vi vil bli spurt om hvor vi vil lagre nøkkelen. Med mindre du

allerede har laget en ssh-nøkkel av denne typen er default pathen grei. Vi blir spurt om vi ønsker å gi den et passord. Vi velger å holde feltet tomt for å ikke gi den et passord. Hvis du valgte å beholde default pathen vil det nå dukke opp to nye filer i SSH-mappen. Kopier public keyen, altså id_ed25519.pub ved å printe den ut med `cat id_ed25519.pub`. Vi må nå legge den inn i GitLab. I GitLab klikker du på brukerikonet ditt øverst i høyre hjørne, så "Settings".



Deretter velger du SSH Keys i menyen til venstre.





Her limer du inn nøkkelen vi laget, gir den et navn og om du ønsker kan du også gi den en utløpsdato. Hvis utløpsdatoen er tom vil nøkkelen vare evig. Til slutt klikker vi “Add key” og vi kan nå bruke SSH for å snakke med GitLab.

6.3.1.4 Oppsett av Docker Registry

For å kunne bruke det nye Docker Registryet må vi legge det til som en Secret i Kubernetes. Dette gjør vi ved å første logge inn med Docker med kommandoen `docker login registry.k.torset.me` Vi blir så bedt om å fylle inn brukernavn og passord. Dette er det samme brukernavnet og passordet som vi lagde når vi registrerte oss på GitLab. Vi får beskjed om at det ble laget en fil i ~/.docker/config.json. Vi lager en Secret basert på denne filen med kommandoen:

```
kubectl create secret generic regcred \
    --from-file=.dockerconfigjson=”~/.docker/config.json” \
    --type=kubernetes.io/dockerconfigjson
```

Denne secreten må legges til i alle Namespace der du har tenkt å bruke det lokale registryet. Nå kan vi lage en Pod der vi henter et image fra det lokale registryet med:

6_3_1_gitlab_med_registry\registry_eksempel.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: private-reg
spec:
  containers:
```

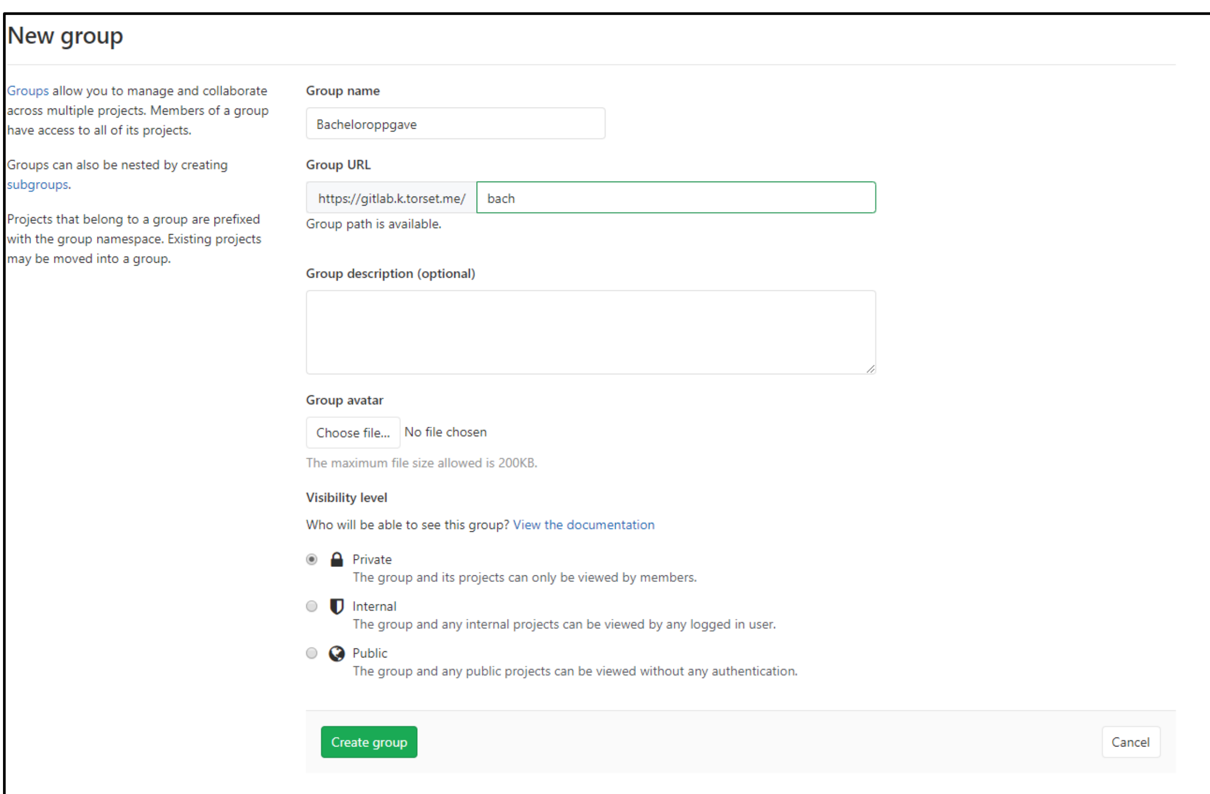
```
  - name: private-reg-container
    image: registry.k.torset.me/en-container
  imagePullSecrets:
  - name: regcred
```

Vi må spesifisere `imagePullSecrets:` hver gang vi skal bruke en container fra dette registryet.

6.3.1.5 Bruk av GitLab og Docker Registry

Lage grupper

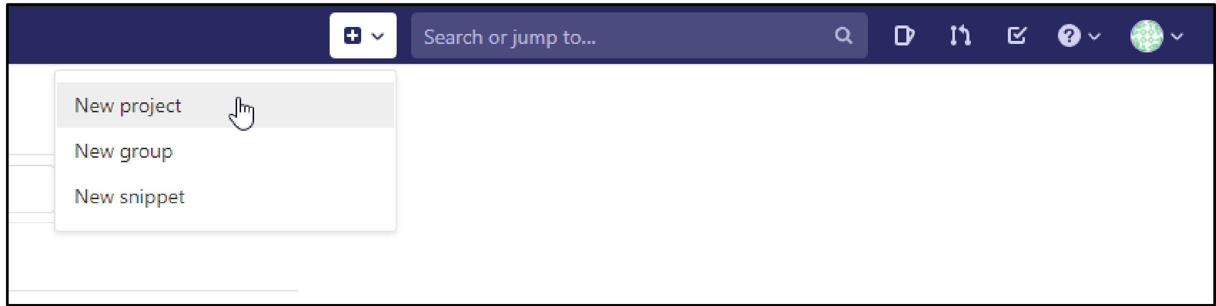
I GitLab er det mulig å lage grupper for å enkelt samarbeide og ha et felles sted å legge prosjekter uten å måtte spesifisere medlemmer for hvert nye prosjekt. Man kan lage en gruppe ved å gå til plusstegnet øverst i høyre hjørne ved siden av søkeboksen og klikke “New group”

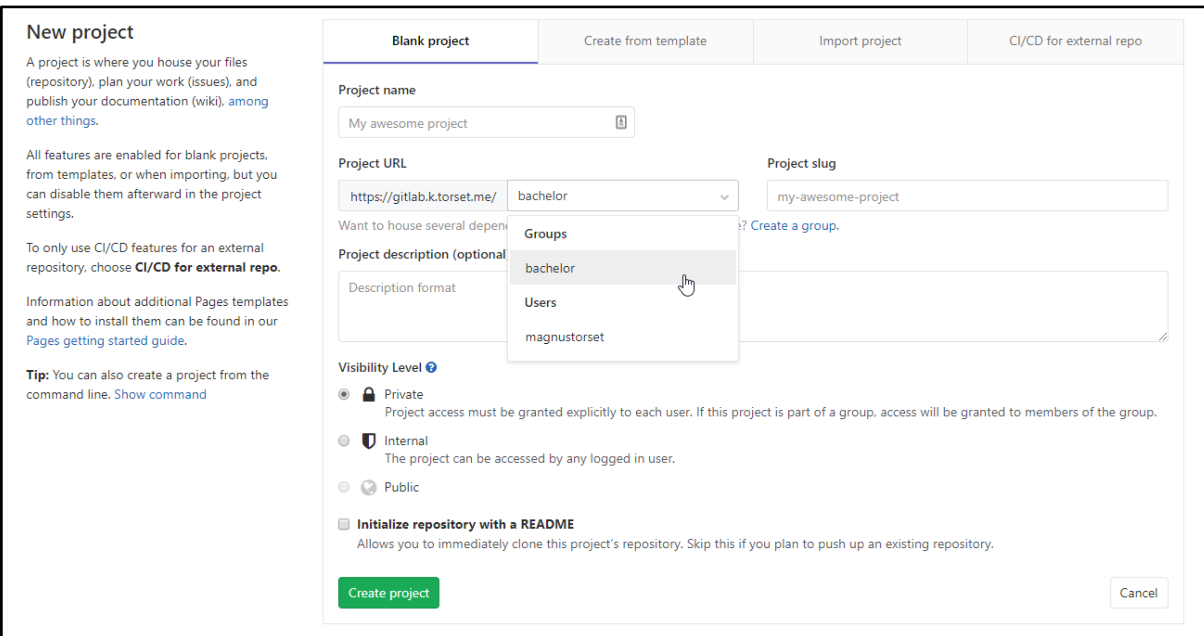


Her gir man gruppen et navn og en URL. URLen blir automatisk fylt inn med gruppenavnet, men de trenger ikke nødvendigvis være like. Man kan gi gruppen et bilde og en beskrivelse og man kan sette hvem gruppen og dens prosjekter skal være synlig for. Alternativene er public, som vil si at det er synlig for alle uten at de trenger å logge inn, internal, som vil si at man må ha registrert en bruker og være logget inn for å se innholdet og private, hvor innholdet kun er synlig for medlemmer av gruppen.

Lage prosjekter

For å lage et prosjekt går man til det samme plusstegnet ved siden av søkeboksen som når vi laget en gruppe, men nå klikker vi “New project”.



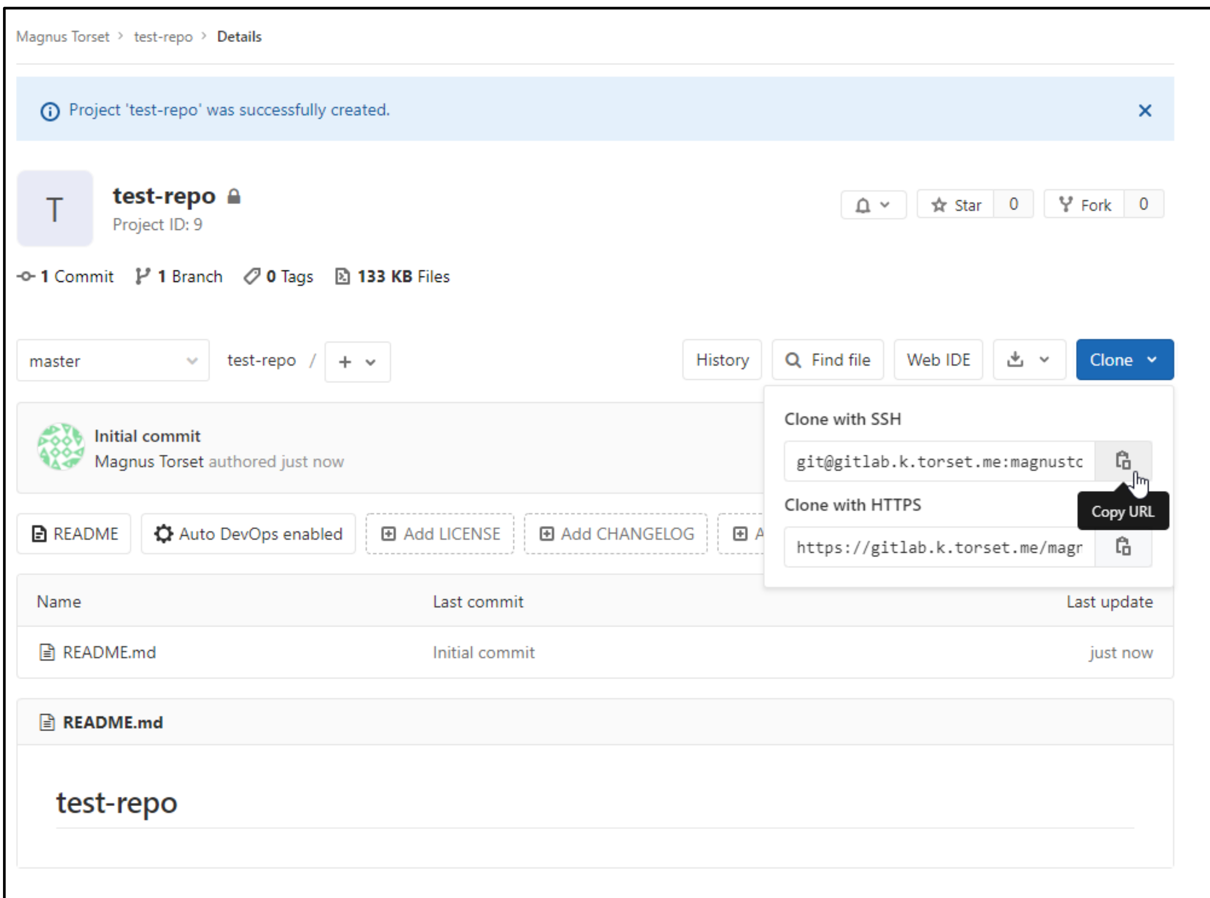


Her kan vi sette et navn på prosjektet vårt. Også her vil URLen autoutfylles med prosjektnavnet, men de trenger ikke være like. Vi kan også velge om prosjektet skal tilhøre brukeren vår eller en gruppe vi er medlem i. Som ved laging av en gruppe har vi her også alternativer for hvor synlig prosjektet skal være med samme regler som for en gruppe.

Bruk av Git

Når du har laget et nytt prosjekt og skal legge til ting i det har du to alternativer. Du kan enten initialisere repositoriet i webgrensesnittet når du lager det, eller gjøre det i kommandolinjen manuelt senere.

Om du ønsker å starte et nytt tomt prosjekt er den enkleste måten å bruke webgrensesnittet. Da huker du av for “Initialize repository with a README”. Nå har du et repo med kun en README. For å begynne å legge til ting laster du ned repoet med `git clone <SSH-URL>`. Denne URLen finner du i prosjektet ditt på gitlab under clone.



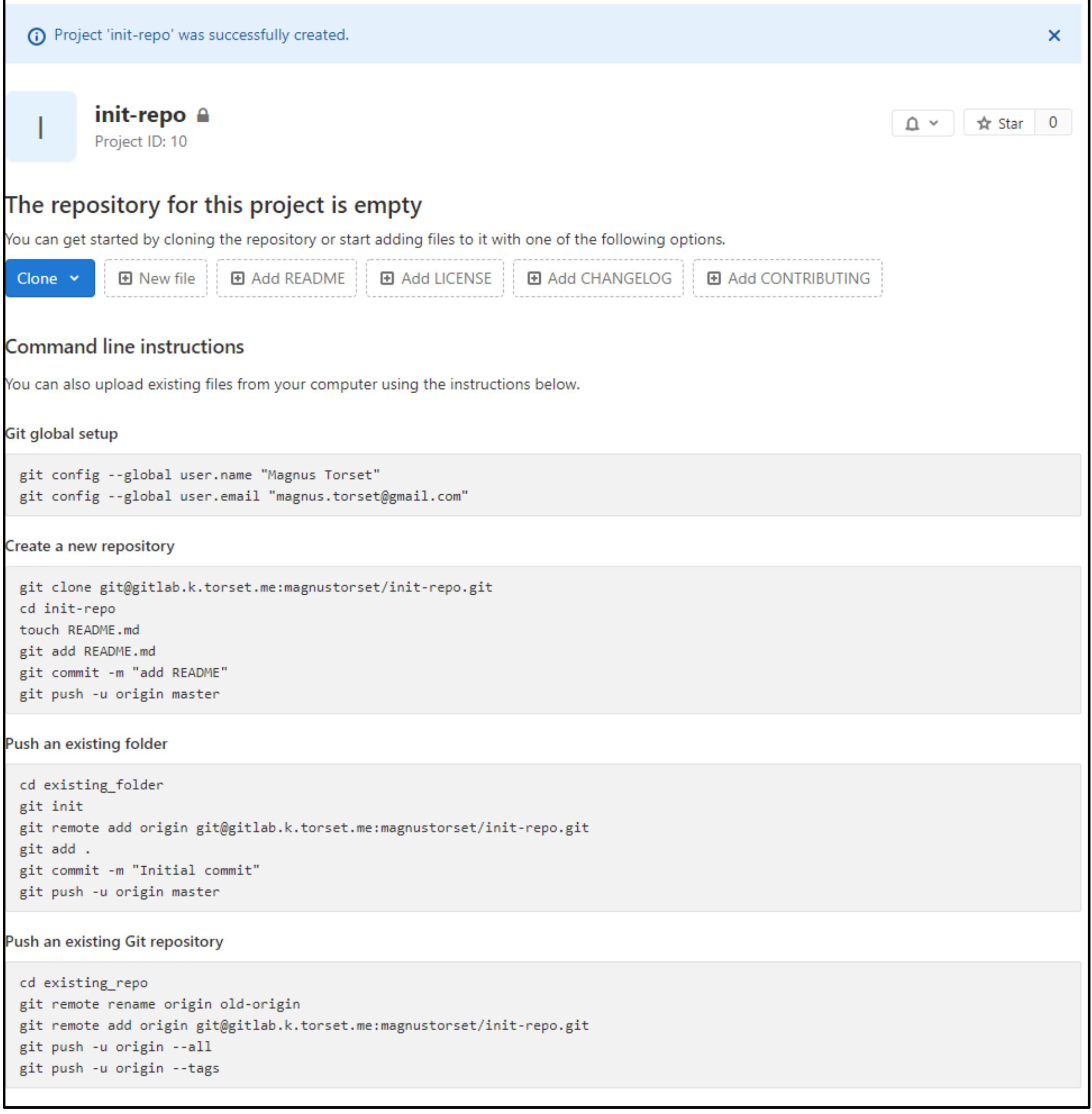
For dette repoet vil kommandoen da være `git clone git@gitlab.k.torset.me:magnustorset/test-repo.git`. Vi satt opp en SSH-nøkkel tidligere, men man kan også bruke HTTPS. Dette krever imidlertid at du skriver inn brukernavn og passord hver gang du skal gjøre noe der du trenger tilgang til repoet som clone, pull eller push. Du vil nå få en mappe med samme navn som repoet og du kan begynne å legge til filer der. Når du har lagt til litt filer og vil laste opp endringene dine er du først nødt til å si til Git at de nye filene du har lagt til skal være en del av repoet. Dette gjøres med `git add`. Du kan legge til én og én fil med `git add <filnavn>` eller legge til alle filene i mappen med `git add .`. Vi må nå lage en commit. En commit er en kommando du kan kjøre på enkelte filer eller en hel mappe som sier at filene inneholder endringer du har gjort og at du ønsker å laste dem opp. I tillegg må det også legges til en kort melding som beskriver hva det er du har gjort. Dette gjøres med `git commit -am "<melding>"`.

```
magnus@k8s-master:~/git/test-repo$ git add .
magnus@k8s-master:~/git/test-repo$ git commit -am "lagt til filene enFil.txt, test og test2"
[master 410160b] lagt til filene enFil.txt, test og test2
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 enFil.txt
 create mode 100644 test
 create mode 100644 test2
magnus@k8s-master:~/git/test-repo$
```

Før vi er klare til å laste opp er vi først nødt til å sjekke om det er blitt utført nye endringer siden sist vi oppdaterte vår lokale versjon av repoet. Dette gjøres med kommandoen `git pull`. Nå kan vi endelig kjøre `git push` for å laste opp endringene våre.

Om du har flere filer du allerede har laget kan det være enklere å laste opp mappen med disse filene i som et repo istedenfor å clone et repo og kopiere filene inn der. Da må vi passe på å ikke velge “Initialize repository with a README” når vi lager prosjektet.

Når vi lager et tomt prosjekt får vi en liste over nyttige kommandoer.



Vi vil pushe en eksisterende mappe så vi navigerer til mappen vår og skriver `git init`. Denne kommandoen lager en skjult mappe, .git, med konfigurasjonsfiler for repoet vårt. Vi må også spesifisere hvilket repo mappen skal være en del av med `git remote add origin <SSH-URL>` og legge til de eksisterende filene med `git add .`. Vi kan nå lage en commit med `git commit -am "<Melding">`. For å pushe endringer må vi legge til hvilken branch vi ønsker å comitte til med `git push --set-upstream origin master`. Denne kommandoen sier at alt som blir pushet herfra, origin, skal pushes til master. Fra nå av trenger vi bare skrive `git push`.

Legge til Docker Images

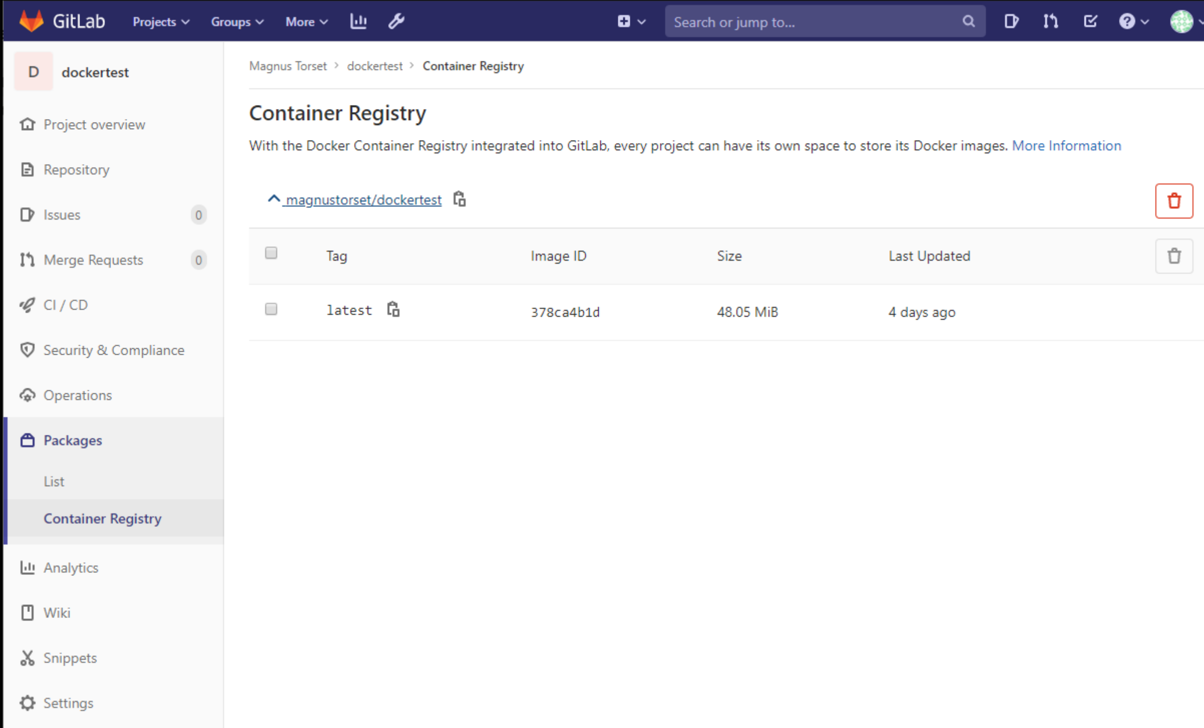
For å legge til et Docker Image i registryet er det nødt til å knyttes til et repository. Vi går ikke over hvordan man lager et Docker Image her. Kun hvordan det legges til i registryet. Vi ønsker å legge til en containerversjon av Debian i registryet vårt. For å få tak i imaget kjører vi `docker pull debian`. Hvis vi kjører `docker images` nå ser vi at vi har et repository debian med en Image ID.

```
magnus@k8s-master:~/git/dockertest$ docker images
REPOSITORY                                TAG              IMAGE ID         CREATED          SIZE
registry.k.torset.me/bachelor/jenkins     slave-1.0        12b7a05b2ad0     2 days ago       1.07GB
registry.k.torset.me/bachelor/jenkins     1.0              0df6c6bbf597     4 days ago       1.33GB
debian                                    latest           378ca4b1d2fe     4 days ago       114MB
jenkins/jnlp-slave                        latest           ffd2afd580a6     5 days ago       523MB
jenkins/jenkins                           lts              7e250da768ed     3 weeks ago      619MB
k8s.gcr.io/kube-proxy                     v1.17.3          ae853e93800d     2 months ago     116MB
k8s.gcr.io/kube-controller-manager        v1.17.3          b0f1517c1f4b     2 months ago     161MB
k8s.gcr.io/kube-apiserver                 v1.17.3          90d27391b780     2 months ago     171MB
k8s.gcr.io/kube-scheduler                 v1.17.3          d109c0821a2b     2 months ago     94.4MB
calico/node                               v3.12.0          fc05bc4225f3     2 months ago     258MB
calico/pod2daemon-flexvol                 v3.12.0          98793d0a88c8     2 months ago     111MB
calico/cni                                v3.12.0          cb6799752c46     2 months ago     207MB
calico/kube-controllers                   v3.12.0          53aa421faf0a     2 months ago     56.6MB
kubernetesui/dashboard                    v2.0.0-beta8     eb51a3597525     4 months ago     90.8MB
k8s.gcr.io/coredns                        1.6.5            70f311871ae1     5 months ago     41.6MB
metallb/speaker                           v0.8.2           4fa93685c115     5 months ago     43.5MB
k8s.gcr.io/etcd                           3.4.3-0          303ce5db0e90     5 months ago     288MB
kubernetesui/metrics-scraper              v1.0.1           709901356c11     9 months ago     40.1MB
quay.io/prometheus/node-exporter          v0.18.1          e5a616e4b9cf     10 months ago    22.9MB
k8s.gcr.io/pause                          3.1              da86e6ba6ca1     2 years ago      742kB
```

For at vi skal kunne laste opp dette imaget til vårt eget regsitry er vi nødt til å endre på navnet slik at docker forstår hvor det skal. I tillegg, siden vi bruker registryet levert av GitLab er vi nødt til å ha et eksisterende repo på GitLab som vi kan knytte imaget til. Vi har derfor laget repoet magnustorset/dockertest. Imaget er nødt til å ha navn av formen <registry-URL>/<repository>. Vi gir Debian et nytt navn med kommandoen `docker tag debian registry.k.torset.me/magnustorset/dockertest`. Hvis vi igjen kjører `docker images` ser vi at vi har fått et nytt registry med samme Image ID som debian og debian ligger der enda.

```
magnus@k8s-master:~/git/dockertest$ docker images
REPOSITORY                                   TAG            IMAGE ID       CREATED         SIZE
registry.k.torset.me/bachelor/jenkins        slave-1.0      12b7a05b2ad0   2 days ago      1.07GB
registry.k.torset.me/bachelor/jenkins        1.0            0df6c6bbf597   4 days ago      1.33GB
debian                                       latest         378ca4b1d2fe   4 days ago      114MB
registry.k.torset.me/magnustorset/dockertest latest         378ca4b1d2fe   4 days ago      114MB
jenkins/jnlp-slave                           latest         ffd2afd580a6   5 days ago      523MB
jenkins/jenkins                              lts            7e250da768ed   3 weeks ago     619MB
k8s.gcr.io/kube-proxy                        v1.17.3        ae853e93800d   2 months ago    116MB
k8s.gcr.io/kube-apiserver                    v1.17.3        90d27391b780   2 months ago    171MB
k8s.gcr.io/kube-controller-manager           v1.17.3        b0f1517c1f4b   2 months ago    161MB
k8s.gcr.io/kube-scheduler                    v1.17.3        d109c0821a2b   2 months ago    94.4MB
calico/node                                  v3.12.0        fc05bc4225f3   2 months ago    258MB
calico/pod2daemon-flexvol                    v3.12.0        98793d0a88c8   2 months ago    111MB
calico/cni                                   v3.12.0        cb6799752c46   2 months ago    207MB
calico/kube-controllers                      v3.12.0        53aa421faf0a   2 months ago    56.6MB
kubernetesui/dashboard                       v2.0.0-beta8   eb51a3597525   4 months ago    90.8MB
k8s.gcr.io/coredns                           1.6.5          70f311871ae1   5 months ago    41.6MB
metallb/speaker                              v0.8.2         4fa93685c115   5 months ago    43.5MB
k8s.gcr.io/etcd                              3.4.3-0        303ce5db0e90   5 months ago    288MB
kubernetesui/metrics-scraper                 v1.0.1         709901356c11   9 months ago    40.1MB
quay.io/prometheus/node-exporter             v0.18.1        e5a616e4b9cf   10 months ago   22.9MB
k8s.gcr.io/pause                             3.1            da86e6ba6ca1   2 years ago     742kB
magnus@k8s-master:~/git/dockertest$
```

Vi kan nå laste opp dette nye imaget til registryet vårt med `docker push registry.k.torset.me/magnustorset/dockertest`. Hvis vi går til repositoryet dockertest i GitLab nå og går til Packages og så Container Registry kan vi se at imaget vårt har blitt lagt til.



6.3.2 Jenkins

Jenkins er et populært verktøy for automatisering av bygging, testing og utrulling av prosjekter. Vi vil sette opp Jenkins til å følge med på et prosjekt i GitLab, for så å bygge, teste og rulle ut endringer til Kubernetes.

6.3.2.1 Installasjon

Det første vi gjør er å opprette en ServiceAccount, en Role og en RoleBinding som skal gi Poden de nødvendige rettighetene:

6_3_2_1_jenkins/rbac.yaml

```yaml
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  name: jenkins
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/log"]
  verbs: ["get","list","watch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: jenkins
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: jenkins
subjects:
- kind: ServiceAccount
  name: jenkins
```

Her er det Role som setter rettighetene. Poden får tilgang til å opprette, endre og slette Poder, se Pod-logger og hente Secrets i sitt Namespace.
Vi vil beholde innstillinger og tillegg vi installerer gjennom omstart, så vi setter også opp et PersistentVolumeClaim, som skal gi Poden lagringsplass:

6_3_2_1_jenkins/persistentvolumeclaim.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: jenkins-volume
spec:
  accesModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Dette er et standard PersistentVolumeClaim som gir 1 GB lagring til Poden. Den har tilgangen ReadWriteOnce, som betyr at volumet kan mountes av én Pod av gangen.
For å la Jenkins bygge alle typer applikasjoner vil vi lage et eget image med Docker installert. Da kan vi enkelt bygge applikasjonen i et image med riktig språk, i stedet for å måtte installere alle språkene vi vil bruke i Jenkins installasjonen. Dette gjør vi gjennom en Dockerfile:

6_3_2_1_jenkins/Dockerfile

```dockerfile
FROM jenkins/jenkins:lts

# Install plugins for better GUI
RUN /usr/local/bin/install-plugins.sh greenballs
RUN /usr/local/bin/install-plugins.sh ansicolor
# Install kubernetes plugin
RUN /usr/local/bin/install-plugins.sh kubernetes

USER root
# Install docker and everything needed to do so
RUN apt update && apt install -y apt-transport-https ca-certificates curl
software-properties-common
RUN curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add -
RUN apt-key fingerprint 0EBFCD88
RUN add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/debian $(lsb_release -cs) stable"
RUN apt update && apt install -y docker-ce=17.12.1~ce-0~debian
RUN usermod -aG docker jenkins

USER jenkins
```

Her installerer vi først noen plugins for bedre GUI og Kubernetes pluginen. Det er flere plugins vi vil ha, men det er ikke alle som er tilgjengelig her. De må vi installere senere. Vi installerer også Docker, men bare kommando-verktøyet. Selve Docker-miljøet skal vi låne fra noen Poden kjører på. Før vi kan bruke det må imaget bygges og legges i Docker-registryet vårt

```
docker build -t registry.k.torset.me/bachelor/jenkins:1.0 .
docker push registry.k.torset.me/bachelor/jenkins:1.0
```

Vi trenger også et ConfigMap med Kubernetes-konfigurasjonen vår. Den kan vi lage med en enkel kommando:

```
kubectl create configmap kube-conf --from-file $HOME/.kube/config
```

Nå har vi det vi trenger i bunnen, og kan lage en Deployment for Poden:

6_3_2_1_jenkins/deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
name: jenkins
spec:
  selector:
    matchLabels:
      app: jenkins
  replicas: 1
  template:
  metadata:
  labels:
    app: jenkins
  spec:
    securityContext:
        fsGroup: 998
    containers:
    - name: jenkins
      image: registry.k.torset.me/bachelor/jenkins:1.0
      env:
      - name: JAVA_OPTS
        value: -Djenkins.install.runSetupWizard=false
      ports:
      - name: http-port
        containerPort: 8080
      - name: jnlp-port
        containerPort: 50000
      volumeMounts:
      - name: jenkins-volume
        mountPath: /var/jenkins_home
```

```yaml
      - name: dockersock
        mountPath: /var/run/docker.sock
      - name: kubeconf
        mountPath: /root/.kube/
    volumes:
    - name: jenkins-volume
      persistentVolumeClaim:
        claimName: jenkins-volume
    - name: dockersock
      hostPath:
        path: /var/run/docker.sock
    - name: kubeconf
      configMap:
        name: kube-conf
    serviceAccountName: jenkins
    imagePullSecrets:
    - name: jenkins-regcred
```

Det er flere ting å legge merke til her. Det første er `securityContext`. Vi legger på en ekstra gruppe på brukeren som kjører Poden. I dette tilfellet er det gruppe-IDen til Docker gruppen på noden vår, som vi har med for å få tilgang til docker senere. Vi har også en miljøvariabel som forhindrer Jenkins i å kjøre installasjonsveiledningen ved oppstart. Det gjør vi fordi vi allerede har installert det viktigste vi trenger, og derfor ikke har behov for veiledningen. Vi har også tre volum montert i Poden. `jenkins-volume` er PersistentVolumet vi har laget. Det er her konfigurasjonen vil lagres. `dockersock` er her vi monterer nodens Docker installasjon. Dette lar Poden kjøre docker-containere uten å ha docker installert selv. Det siste volumet er `kubeconf`, som gir tilgang til å bruke kubectl. Dette trenger Poden senere for å deploye en app til Kubernetes. Til sist gir vi Poden den ServiceAccounten vi laget tidligere og den nødvendige informasjonen for å hente ned imaget vårt.
Poden skal også ha to Servicer. Den ene heter jenkins-master og er for tilgang til GUIet, mens den andre heter jenkins-jnlp, og er for kommunikasjon mellom master og slave. Disse peker til hver sin port på den samme Poden.

6_3_2_1_jenkins/services.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: jenkins-master
spec:
  ports:
  - port: 8080
    targetPort: 8080
  selector:
    app: jenkins
---
apiVersion: v1
```

```
kind: Service
metadata:
  name: jenkins-jnlp
spec:
  ports:
  - port: 50000
    targetPort: 50000
  selector:
    app: jenkins
```

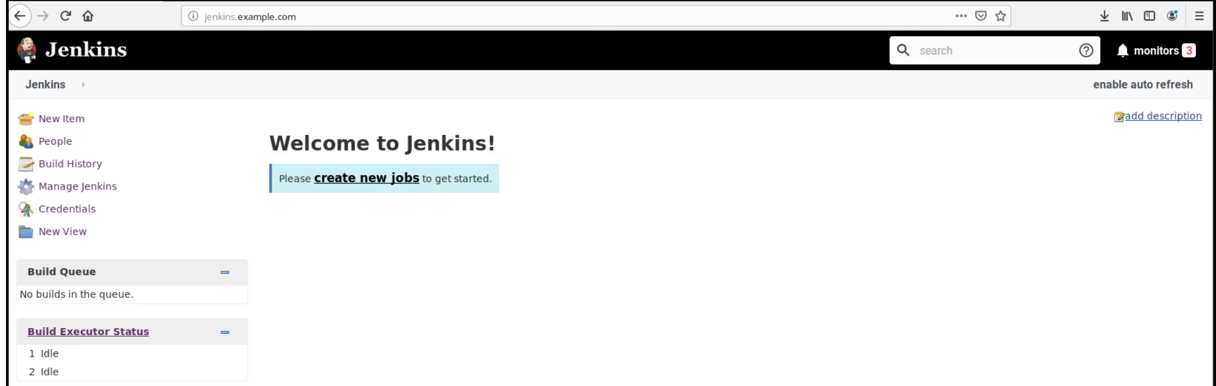
Det siste vi må gjøre er å lage en Ingress for å gjøre GUIet tilgjengelig utenfra. Vi setter adressen jenkins.example.com til å peke til tjenesten:

```
                                                    6_3_2_1_jenkins/ingress.yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: jenkins
spec:
  rules:
  - host: jenkins.example.com
    http:
      paths:
      - backend:
          serviceName: jenkins-master
          servicePort: 8080
```

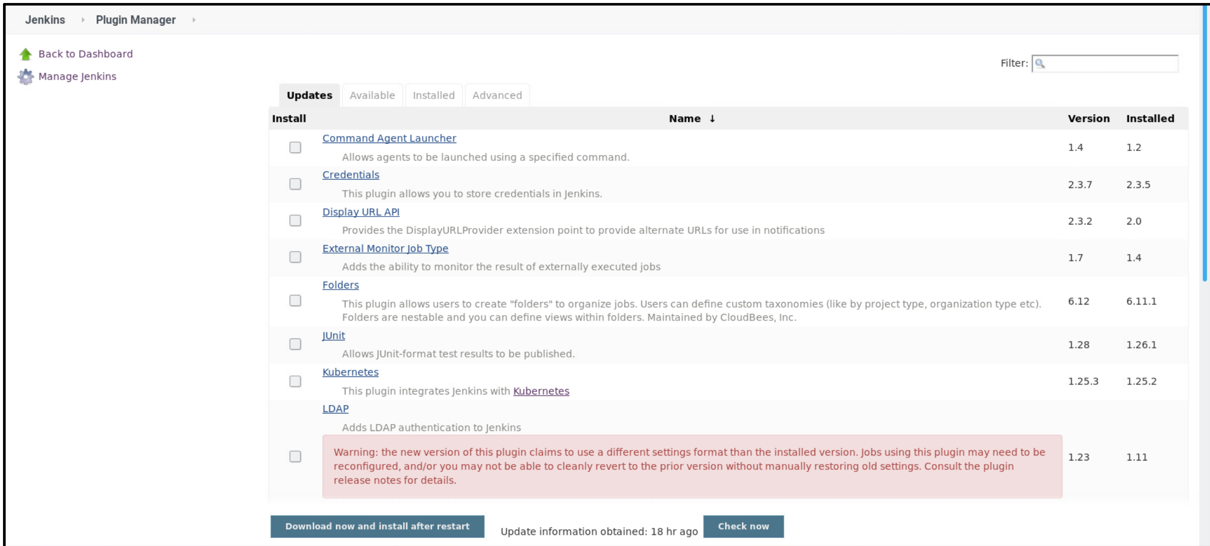
Selve oppsettet av Jenkins skjer gjennom GUIet, men først trenger vi IP-adressen til Kubernetes masteren:

```
simen@k8s-master:/opt/kubernetes/jenkins$ kubectl cluster-info | grep master
Kubernetes master is running at https://10.100.39.23:6443
simen@k8s-master:/opt/kubernetes/jenkins$
```

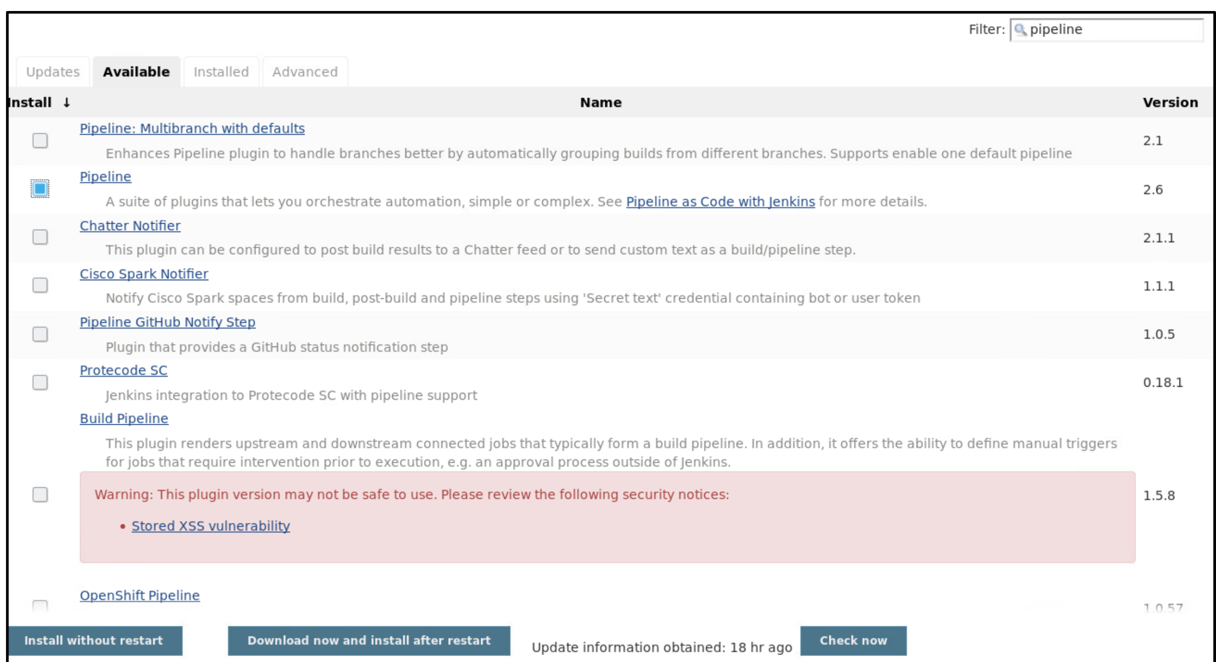
Nå kan vi finne Jenkins-GUIet i nettleseren ved å gå til jenkins.example.com. Vi kommer til hjemmesiden, hvor vi vil finne alle jobbene våre senere.



Før vi kan begynne å bruke Jenkins er vi nødt til å gjøre noen endringer. Dette gjør vi under Manage Jenkins og Manage Plugins, hvor vi først kommer til siden for oppdateringer.

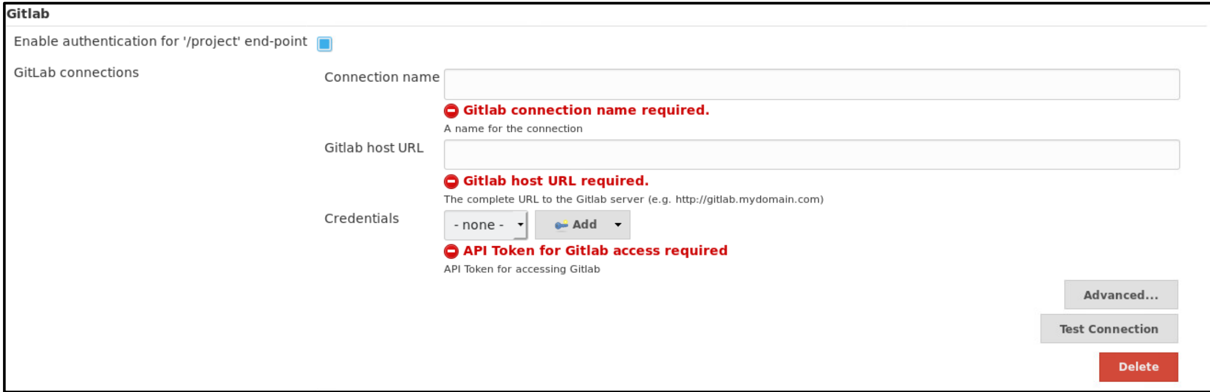


Mange av de installerte pluginene trenger oppdateringer, og er i tillegg nødvendige for andre plugins vi vil installere. Vi markerer alle oppdateringene og velger download now and install after restart. Da vil oppdateringene lastes ned og jenkins startes på nytt. Når oppdateringen er ferdig går vi tilbake til Plugins-siden igjen, men går i stedet til Available-fanen. Her søker vi oss frem til og velger GitLab og Pipeline som de Pluginene vi vil installere. Igjen velger vi Download now and install after restart.

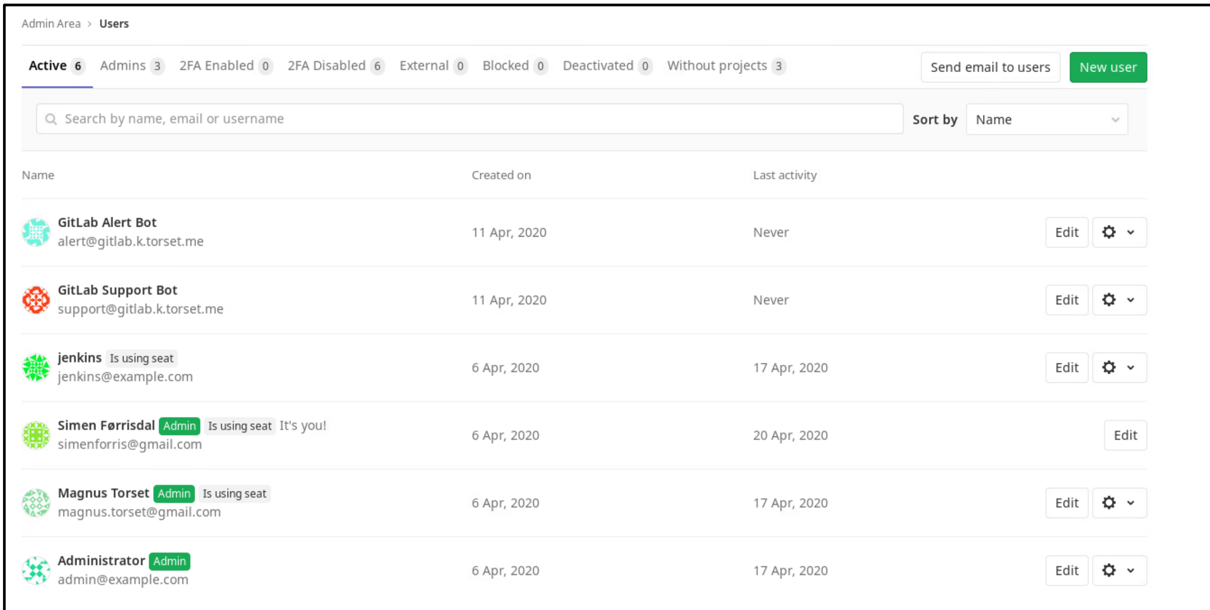


6.3.2.2 Konfigurasjon

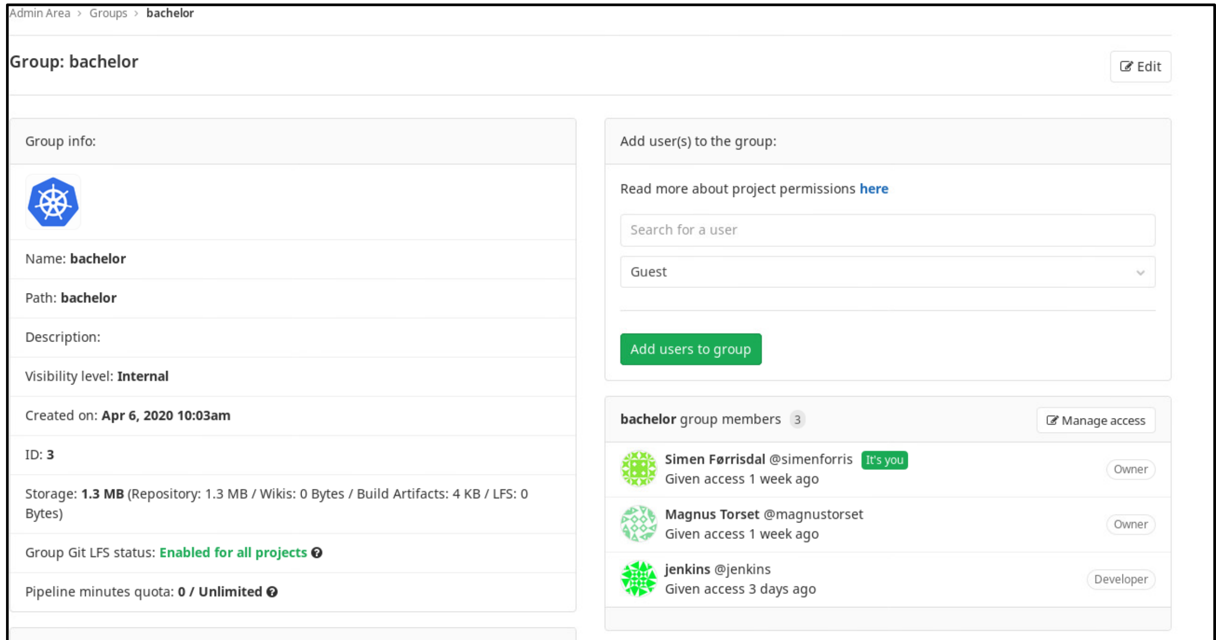
Når denne installasjonen er ferdig kan vi begynne å konfigurere de pluginene vi har installert. Vi går til Manage Jenkins og Configure System og blar ned til GitLab seksjonen.



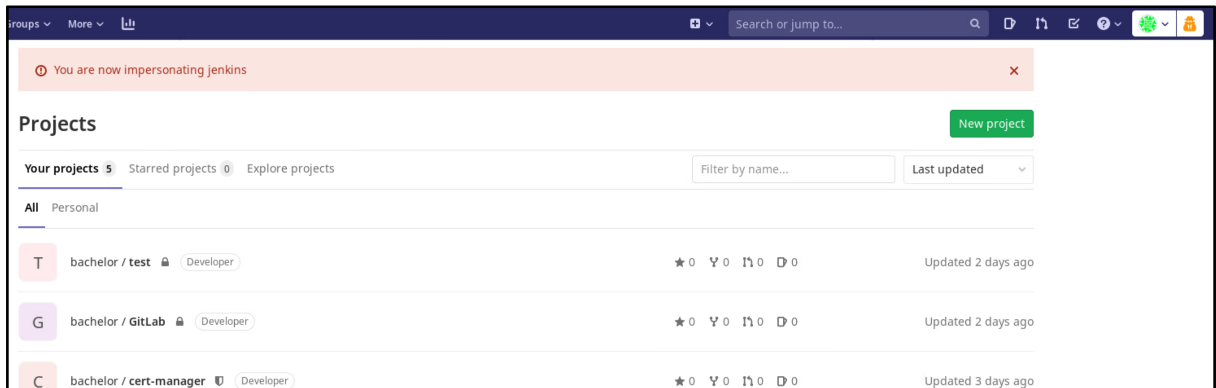
Her trenger vi et API-Token, som er en nøkkel knyttet opp mot en bruker i Gitlab for å gi Jenkins tilgang. Vi åpner et nytt vindu og går til gitlab.k.torset.me., hvor vi logger vi inn som en administrator-bruker og går til Admin Area, Users og New User for å opprette en bruker vi kaller jenkins



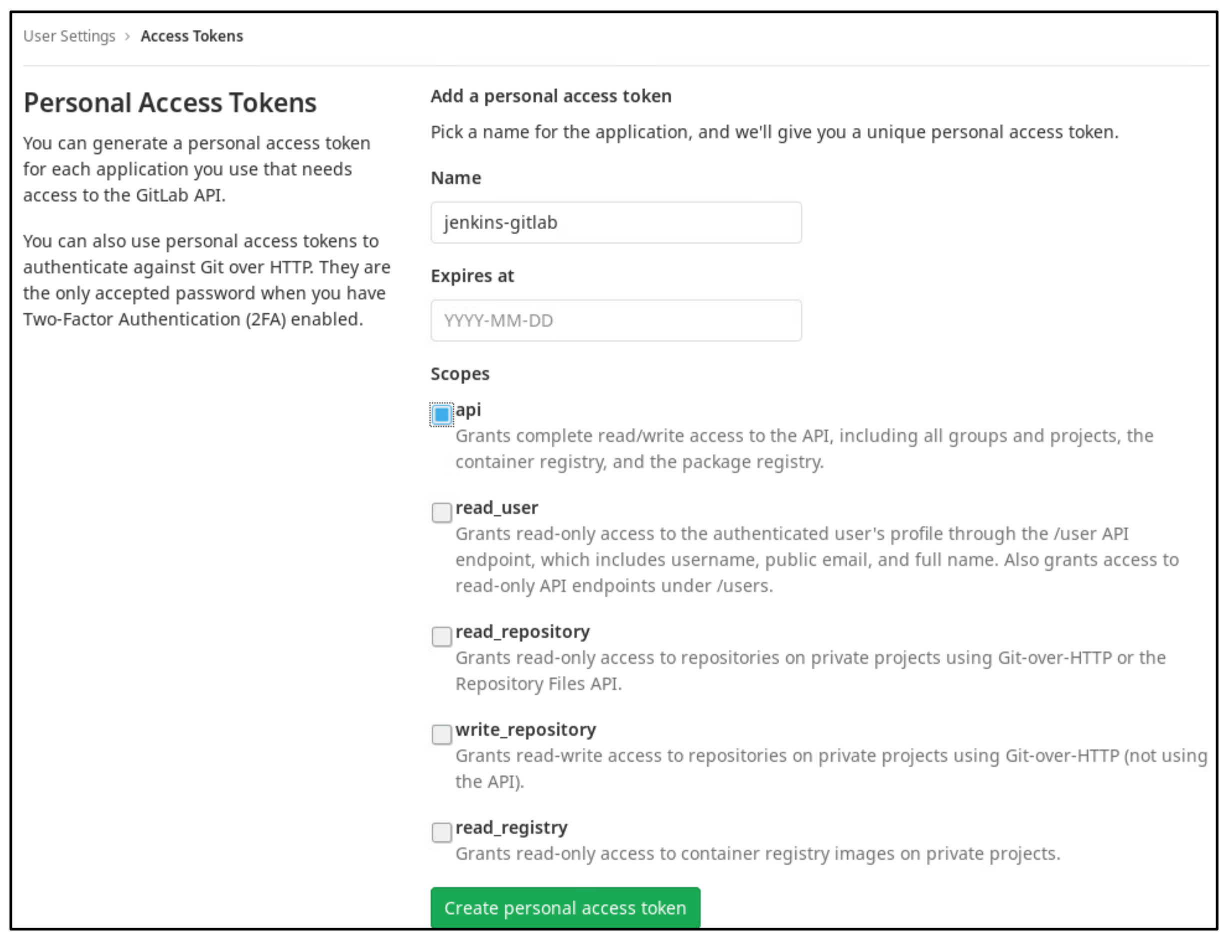
For at brukeren skal ha den nødvendige tilgangen går vi til Groups, velger gruppen Bachelor og legger til jenkins-brukeren som en Developer.



Nå kan vi gå tilbake til listen over brukere, trykke på brukeren jenkins og velge Impersonate. Da kan vi utgi oss for å være denne brukeren og hente et API token på dens vegne

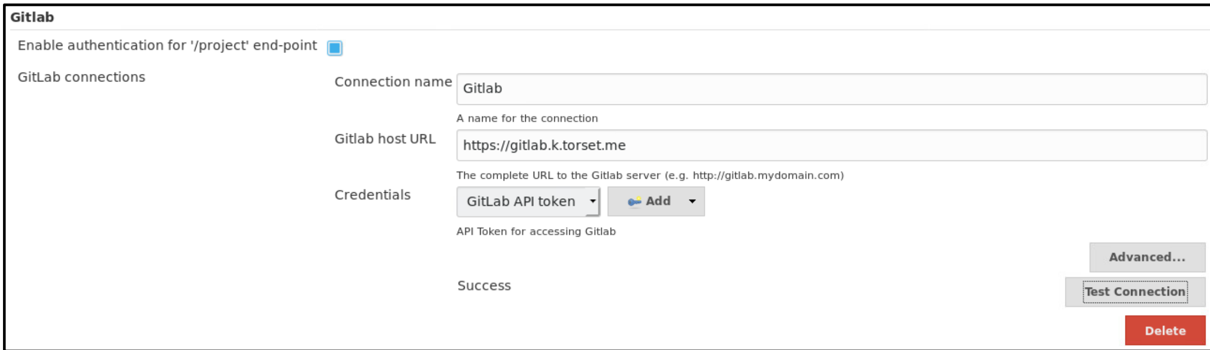


Oppe i høyre hjørne kan vi trykke på profilbildet og velge Settings. Her går vi til Access Tokens og lager en nøkkel av typen API som vi kaller jenkins-gitlab



Når vi lager Tokenet får vi muligheten til å kopiere det. Det er viktig å gjøre det før man forlater siden, fordi det ikke kan nås senere. Nå kan vi gå tilbake til Jenkins-GUIet og legge til tokenet. Vi trykker Add og fyller legger inn tokenet med en ID slik at vi kan finne det igjen

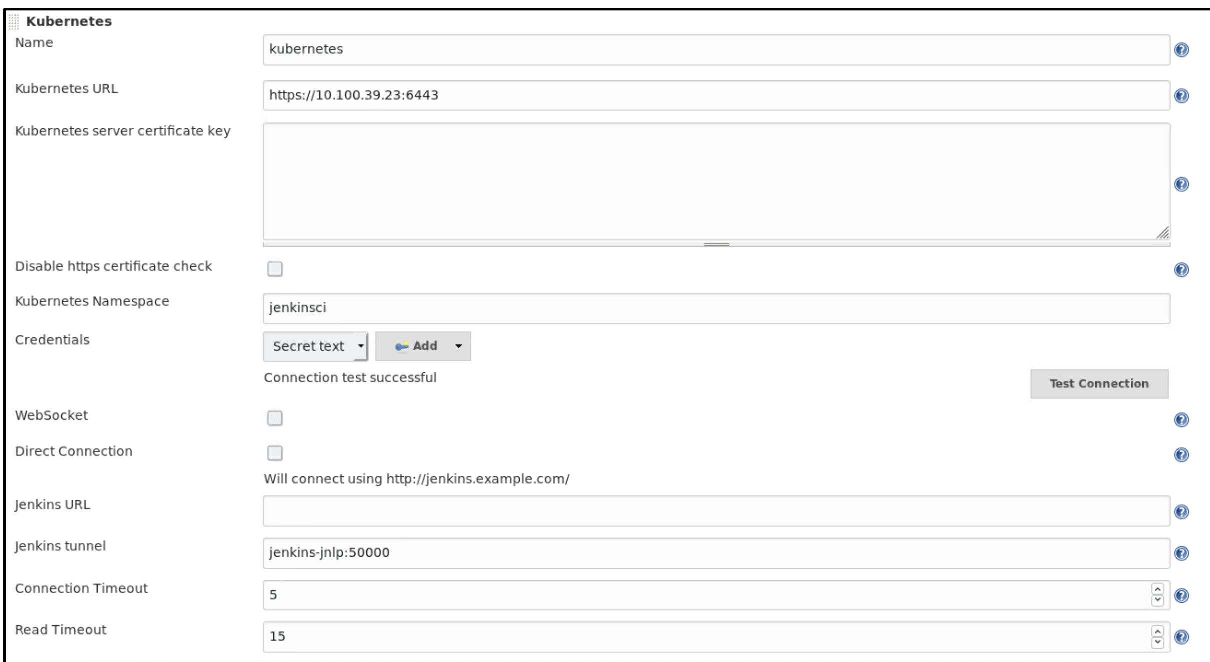
Nå kan vi finne tokenet i listen, og vi kan fylle inn resten av informasjonen. Når vi trykker på Test Connection skal vi se at tilkoblingen er vellykket.



Når vi ser at tilkoblingen har lyktes kan vi lagre innstillingene og gå tilbake til hovedskjermen.

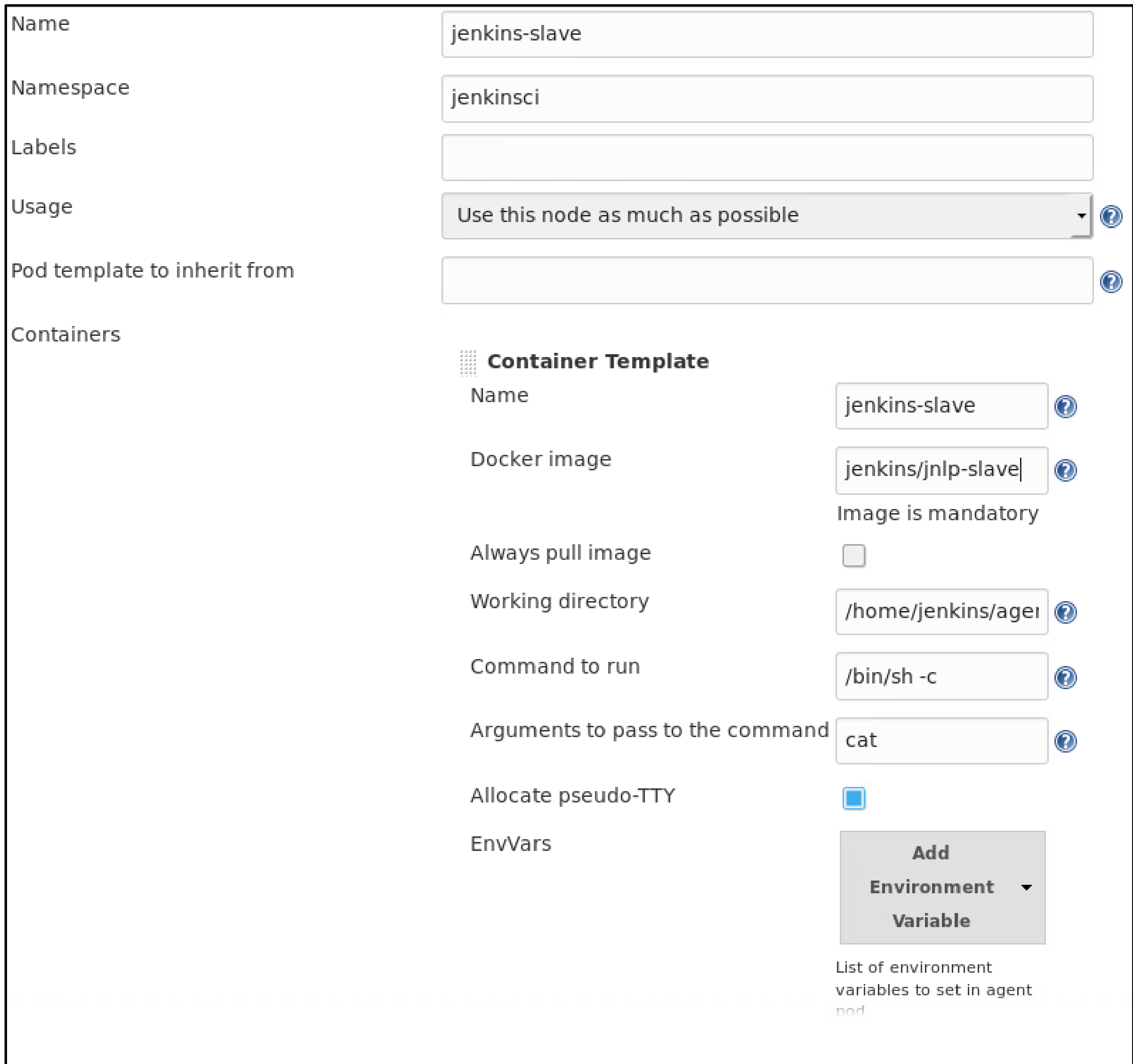
For å la jenkins bruke slaver må vi sette opp Kubernetes som en Cloud. Vi går til Manage Jenkins, Manage Nodes and Clouds og Configure Clouds. Her velger vi Add a new Cloud og velger Kubernetes.

Her må vi sette opp detaljer for Kubernetes og en mal Podene skal bruke. Vi starter med selve Kubernetes konfigurasjonen ved å trykke på Kubernetes Cloud Details.



Her har vi fyllt inn URLen til Kubernetes master-noden som vi hentet tidligere, oppgitt Namespacet vi bruker og lagt til ServiceAccounten vår som credentials. Når vi legger til en ServiceAccount trenger vi ikke fylle ut noe. Vi velger bare typen Kubernetes ServiceAccount, og den ServiceAccounten vi ga til Poden vil velges automatisk. Når vi trykker Test Connection skal vi se at testen er vellykket. Vi oppgir også slave-Servicen vi laget tidligere som Jenkins Tunnel. Den brukes av slavene for å kommunisere med master.

Under Pod templates fyller vi inn informasjon om slave-poden. Vi gir den et navn og sier hvilket Namespace den skal ligge i og velger at denne noden skal brukes så mye som mulig, slik at masteren er mest mulig ledig. Under container-template setter vi imaget til jenkins/jnlp-slave, som er det offisielle slave-imaget fra jenkins.

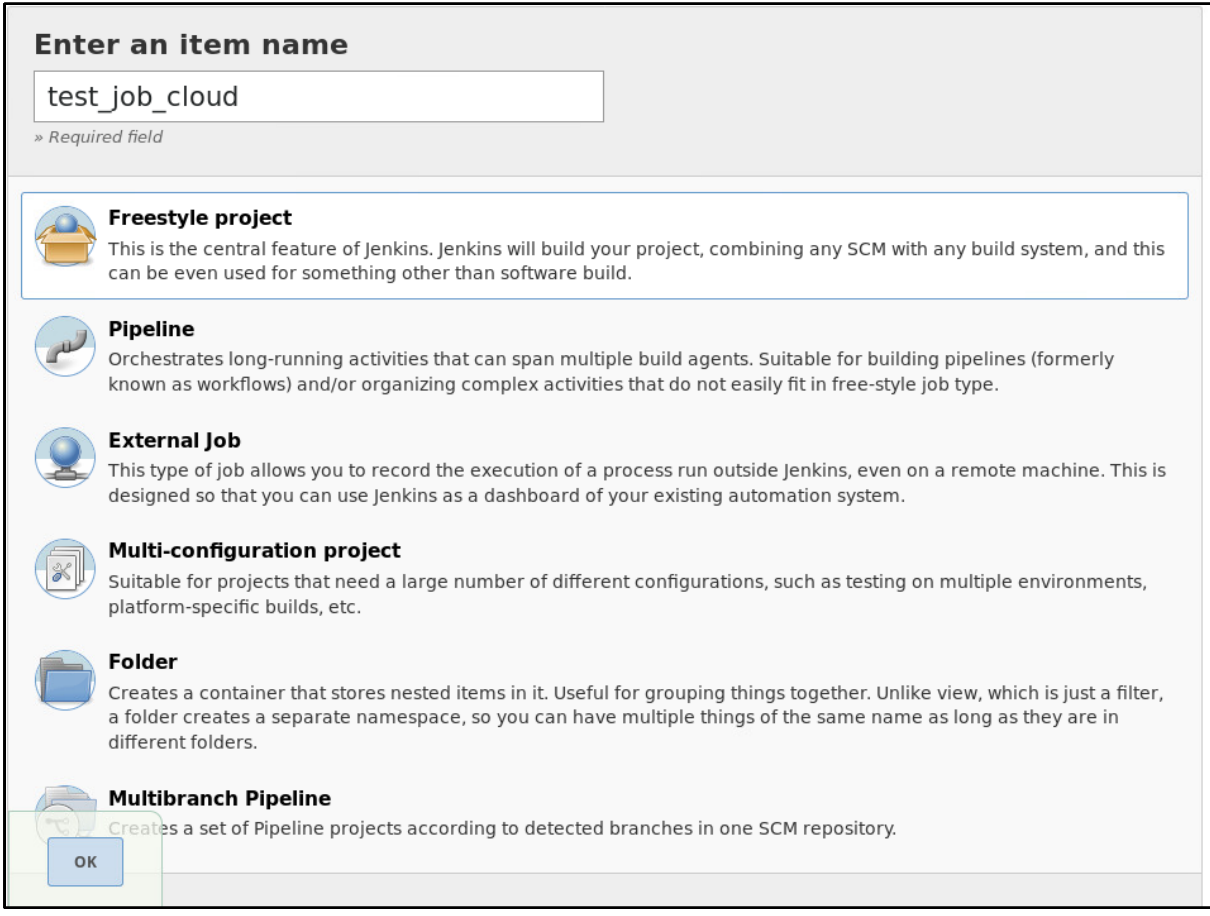


Det er kun master som kan bruke docker. For å fortelle hvilke jobber som bruker docker og skal kjøre på master vil vi bruke label. Vi går til Manage Jenkins, Manage Nodes and Clouds og trykker på tannhjulet ved master-noden. Her velger vi Only jobs with label expression matching this node, og setter label til docker. Dermed kan vi sette label docker på de jobbene som skal kjøre på master.

Nå er jenkins ferdig konfigurert for Kubernetes. Vi kan lagre innstillingene og gå videre til å lage noen test-jobber.

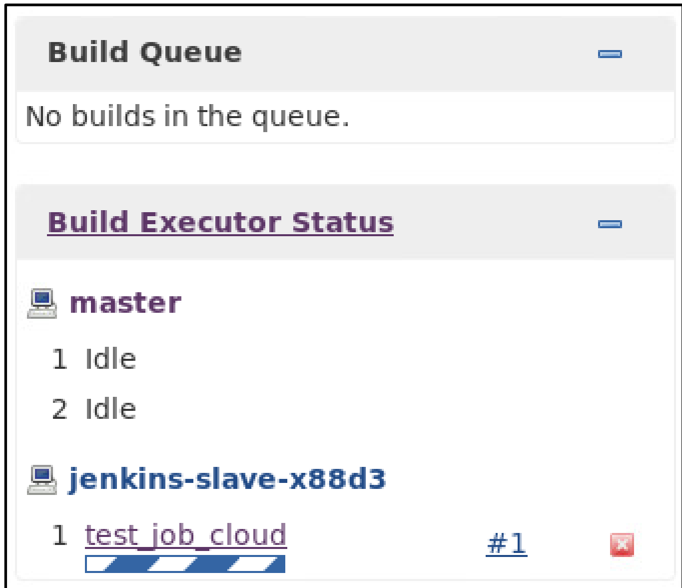
6.3.2.3 Testing

Tilbake på hovedskjermen trykker vi Create new job og lager en jobb av typen Freestyle Project som vi kaller `test_job_cloud`.



Under konfigurasjonen går vi bare til Build og legger til execute shell, hvor vi skriver `sleep 30`. Denne kommandoen gjør ingen ting, men lager en jobb som tar en liten stund, slik at vi kan se at slaven blir opprettet som den skal.

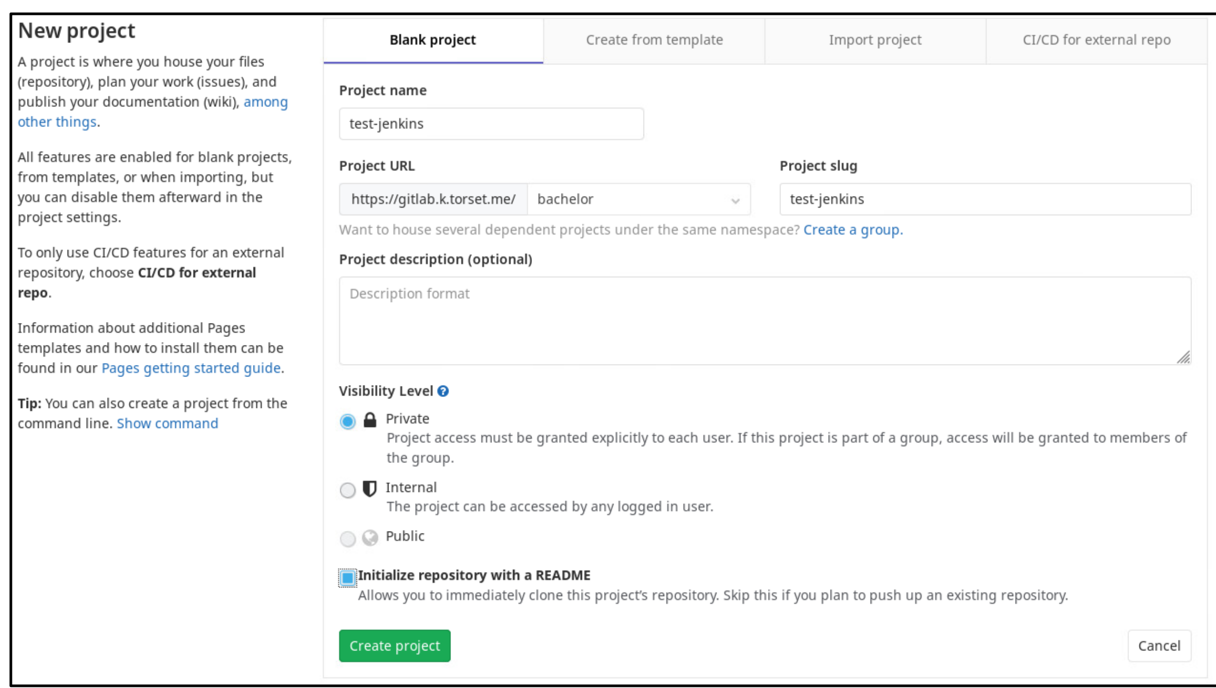
Vi lagrer jobben, og går til hovedskjermen for å kjøre den. Etter en stund vil vi se i listen til venstre at det lages en slave som gjennomfører jobben



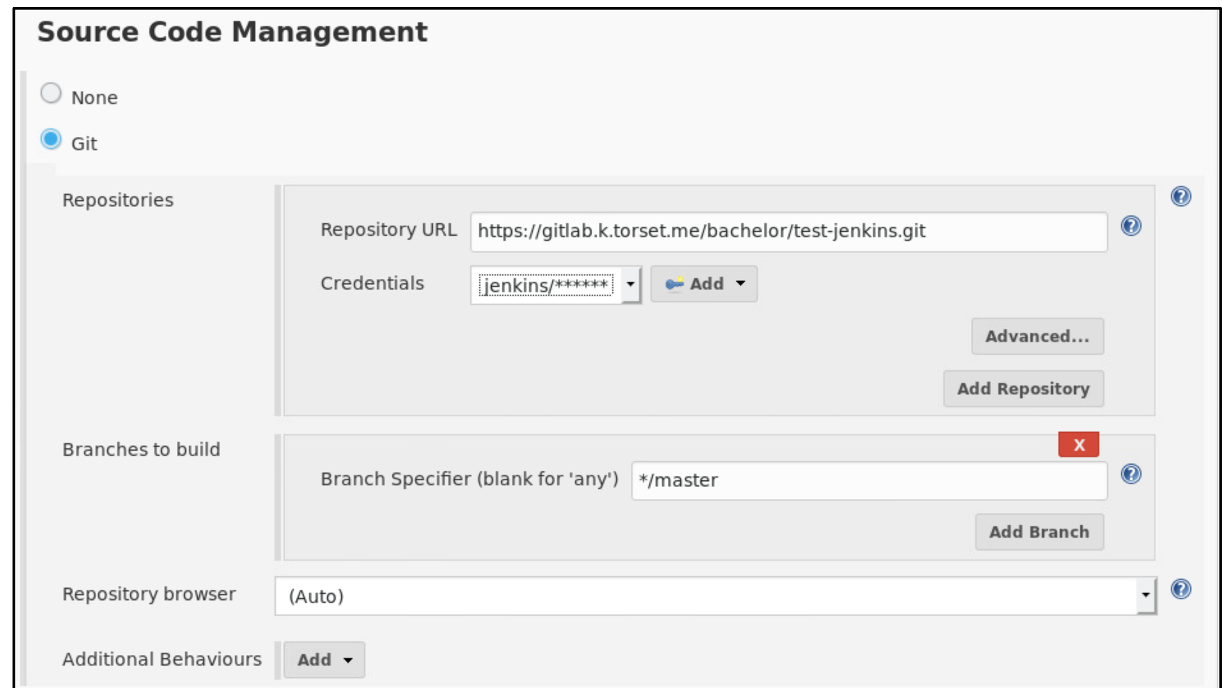
Hvis vi lister ut Poder i Kubernetes kan vi også se slaven der. Poden vil lages når jobben starter, og tas ned igjen når den er ferdig.

```
NAME                        READY   STATUS        RESTARTS   AGE
jenkins-5dc9bcd974-g59kp    1/1     Running       0          2d19h
jenkins-slave-x88d3         2/2     Terminating   0          43s
```

For å teste gitlab koblingen lager vi først et prosjekt i Gitlab. Vi går tilbake til gitlab og lager et nytt prosjekt som vi kaller `test-jenkins`. Vi velger også å starte prosjektet med en README-fil, slik at vi kan se om jenkins finner filene.



Tilbake i Jenkins lager vi en ny jobb av Typen Freestyle Project som vi kaller `test_job_gitlab`. I konfigurasjonen av jobben går vi til Source Code Management og velger Git. Her fyller vi inn url til prosjektet og legger til credentials med brukernavn og passord for brukeren vi lagde tidligere. Hvis vi ikke får noen feilmeldinger er tilkoblingen vellykket.



Videre under Build legger vi en kommando for å liste filene i tillegg til `sleep 30` kommandoen som vi brukte tidligere.

Nå kan vi lagre jobben og gå tilbake til hovedskjermen for å kjøre jobben. Hvis vi trykker på jobben mens den kjører kan vi gå til console output og følge med på hva som skjer underveis.

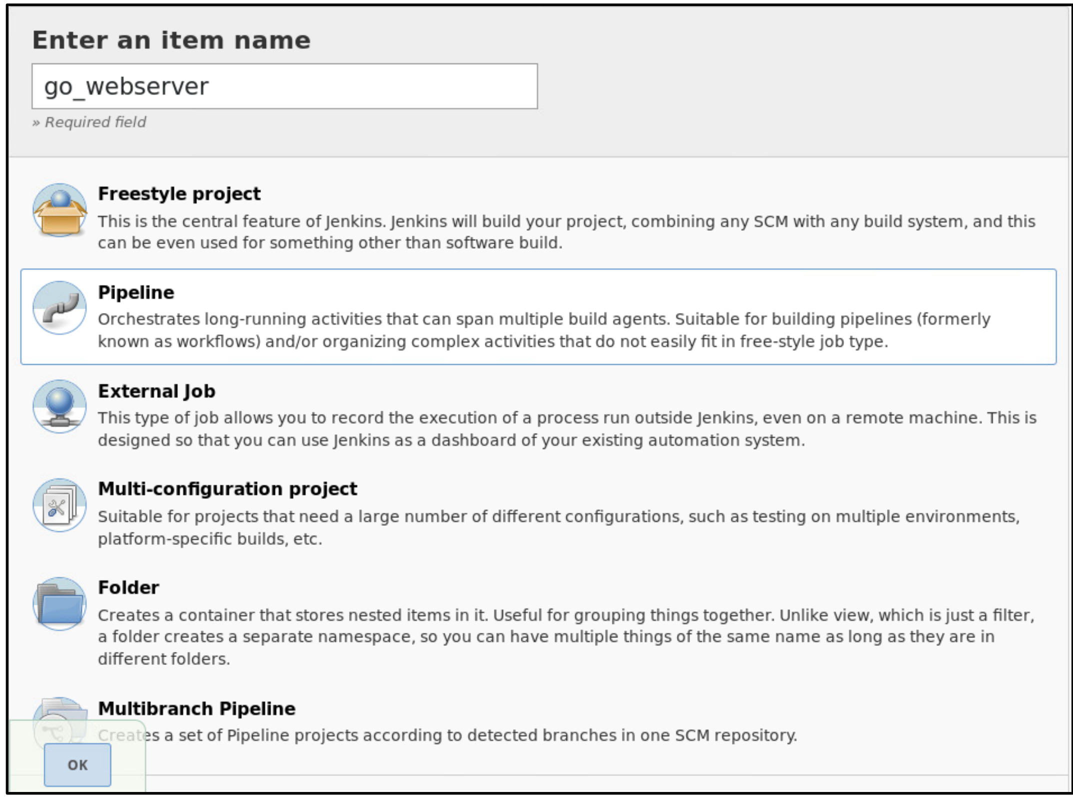
```
Fetching upstream changes from https://gitlab.k.torset.me/bachelor/test-jenkins.git
using GIT_ASKPASS to set credentials
 > git fetch --tags --progress -- https://gitlab.k.torset.me/bachelor/test-jenkins.git +refs/
 > git rev-parse refs/remotes/origin/master^{commit} # timeout=10
 > git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
Checking out Revision cc991257b3b20fb7e7a66f4e250e370dde2195da (refs/remotes/origin/master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f cc991257b3b20fb7e7a66f4e250e370dde2195da # timeout=10
Commit message: "Initial commit"
First time build. Skipping changelog.
[test_job_gitlab] $ /bin/sh -xe /tmp/jenkins3272267909814233497.sh
+ ls -a
.
..
.git
README.md
+ sleep 30
```

Vi ser at jenkins hentet prosjektet uten problemer, og vi finner README filen med `ls -a` som vi satte i jobben.

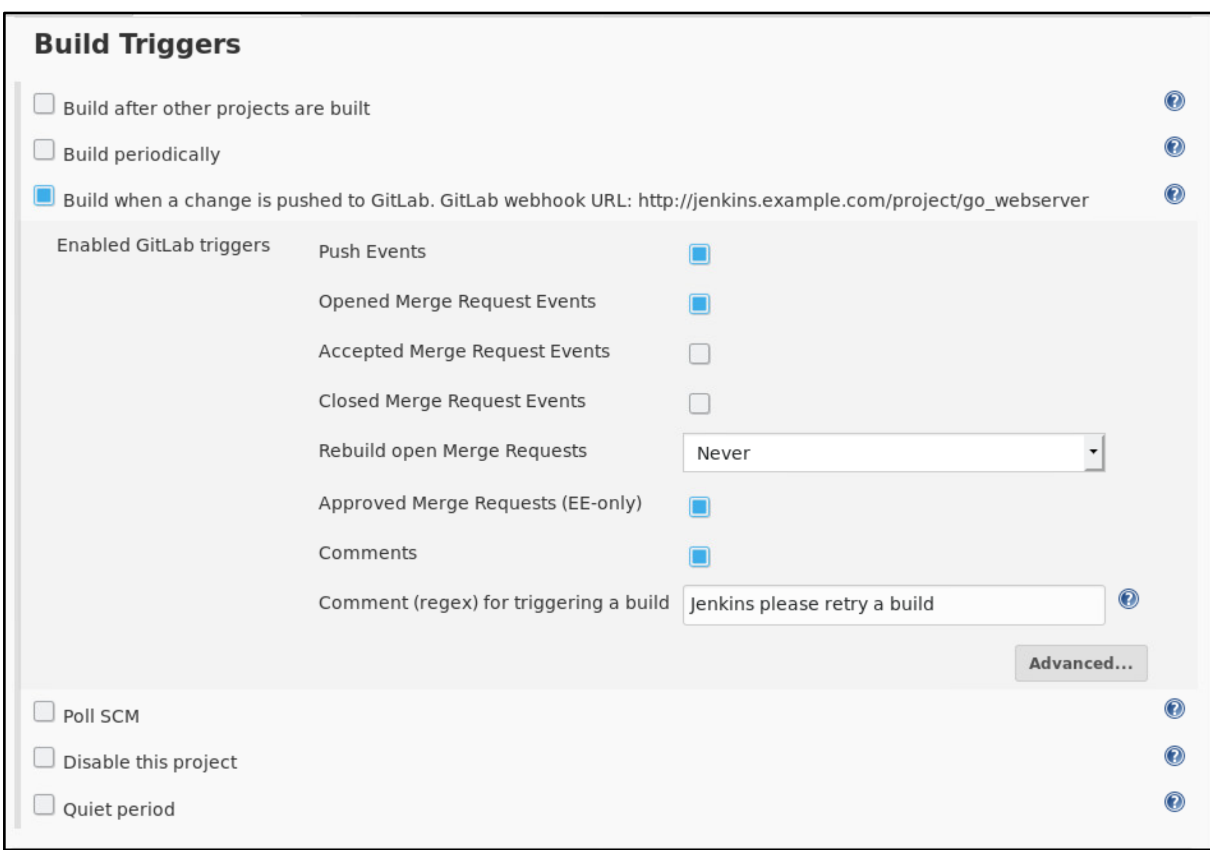
6.3.2.4 Testprosjekt

Nå vil vi sette opp et prosjekt for en webserver skrevet i Go. Når noen pusher en endring til GitLab skal jenkins bygge og teste koden, for så å bygge et Docker image og rulle ut applikasjonen i Kubernetes.

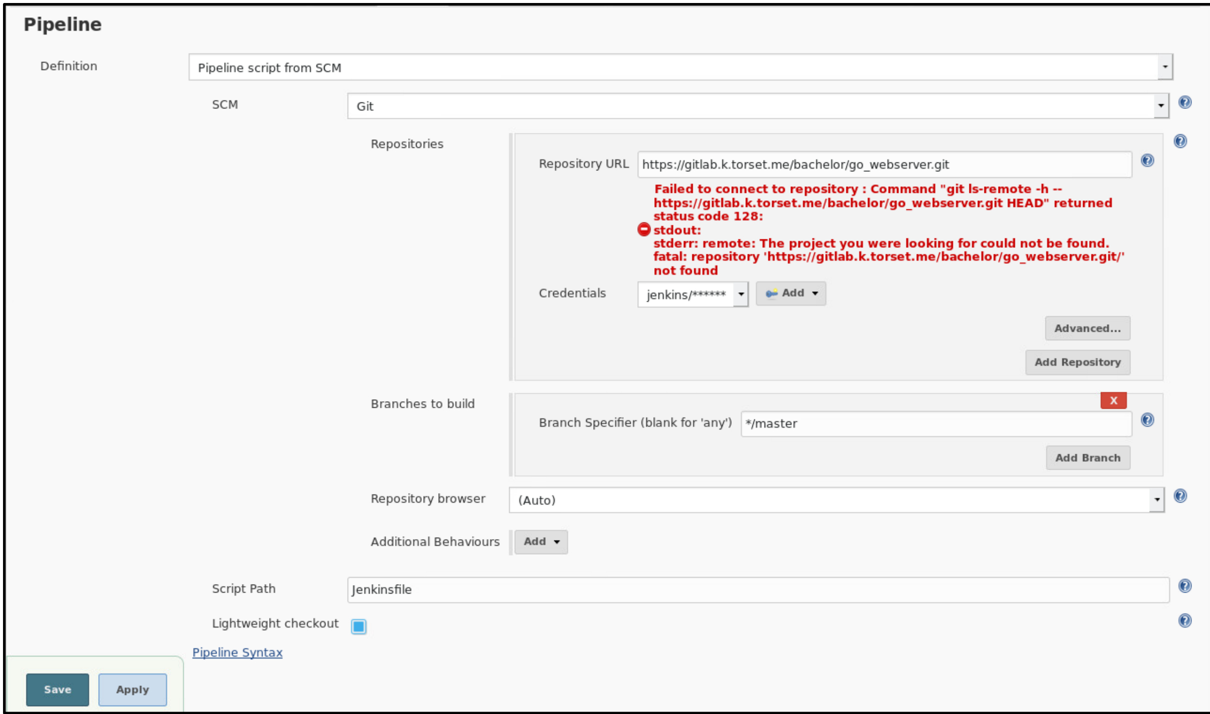
Vi starter med å sette opp Jenkins jobben. Vi går til skjermen og velger Create new job. Denne gangen velger vi typen Pipeline, og kaller den `go_webserver`



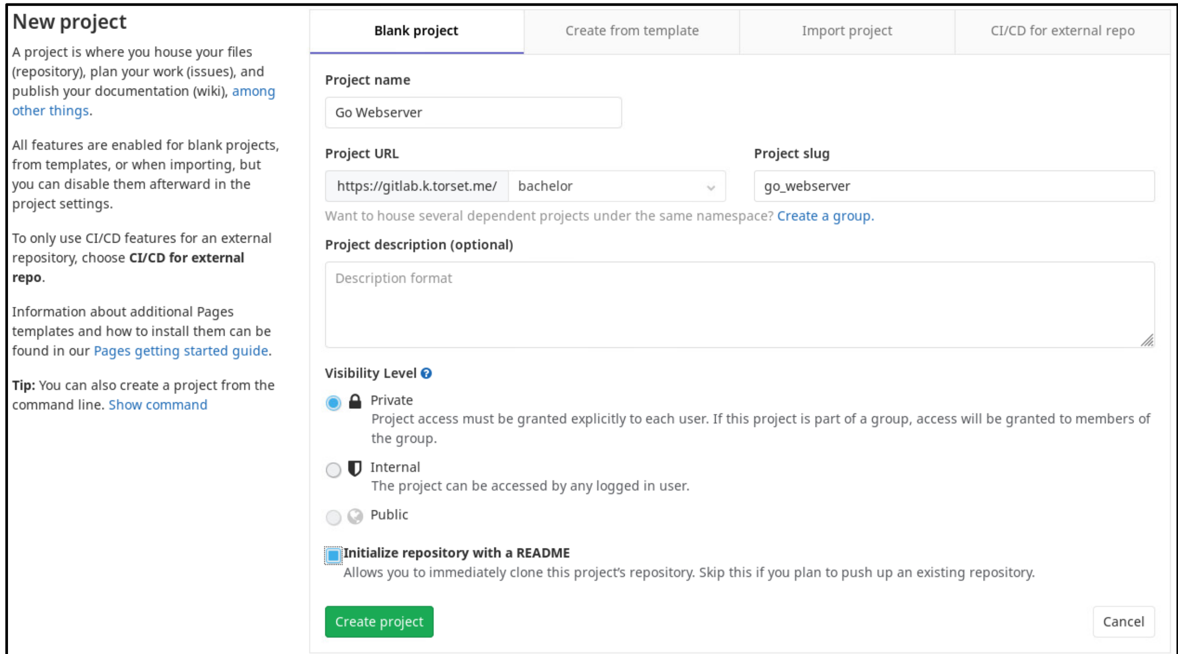
I konfigurasjonen av jobben går vi først til Build Triggers og huker av for å bygge når en endring pushes til GitLab. Dette lar Jenkins lytte etter startsignal på en URL. Vi kopierer denne URLen for å sette opp GitLab prosjektet senere.



Under Pipeline skal vi fortelle Jenkins hva som skal gjøres. Vi vil sende denne informasjonen sammen med koden, og velger derfor Pipeline script from SCM. Her fyller vi inn informasjon til prosjektet `go_webserver`. Vi får en feilmelding fordi vi ikke har laget prosjektet enda, men den kan vi bare ignorere. Script Path forteller Jenkins at filen den skal bruke heter Jenkinsfile, og i dette tilfellet ligger i rotmappen til prosjektet.



Nå er jobben klar, og vi går videre til GitLab for å lage prosjektet under gruppen bachelor. Siden jenkins-brukeren er medlem av denne vet vi at vi får riktig tilgang.



I det nye prosjektet går vi til Settings og Integrations og blar oss ned til Jenkins CI. Her fyller vi inn URLen til Jenkins og navnet på jobben. Vi ser at Push er huket av, slik at jobben vil gjennomføres når noen pusher en endring til prosjektet.

Jenkins CI

You must have installed the Git Plugin and GitLab Plugin in Jenkins. More information

Active

Trigger

Push
Event will be triggered by a push to the repository

Merge request
Event will be triggered when a merge request is created/updated/merged

Tag push
Event will be triggered when a new tag is pushed to the repository

Jenkins url: http://jenkins.example.com

Project name: go_webserver

Go_Webserver

Username

Password

Test settings and save changes

Cancel

Når vi lager innstillingene vil GitLab forsøke å kjøre jobben. I Jenkins kan vi se at jobben feiler. Dette er fordi det ikke finnes noen Jenkinsfile i prosjektet.

All +

S	W	Name ↓	Last Success	Last Failure	Last Duration
		go_webserver	N/A	3 min 46 sec - #1	3.9 sec
		test_job_cloud	2 hr 3 min - #1	N/A	34 sec
		test_job_gitlab	56 min - #1	N/A	42 sec

Nå kan vi åpne en terminal og klone prosjektet for å legge inn koden

```
git clone git@gitlab.k.torset.me:bachelor/go_webserver.git
cd go_webserver
```

Koden vi bruker er en enkel webserver skrevet i Go som er hentet fra et eksempel. Vi har main.go, som er selve applikasjonen, og en Dockerfile som forteller hvordan imaget til applikasjonen skal bygges. I tillegg har vi en mappe som heter k8s som inneholder alle konfigurasjonsfilene for applikasjonen i Kubernetes. Den filen vi vil se på er Jenkinsfile, som inneholder alle instruksjonene for Jenkins. Den ligger vedlagt under `6_3_2_4_go-web/Jenkinsfile`. Første del ser slik ut

```
pipeline {
    agent { label 'docker' }
    environment {
        registry = "registry.k.torset.me/bachelor/go_webserver"
        GOCACHE = "/tmp"
    }
    stages {
```

Filen starter med et pipeline objekt, og definerer først hvilken node som skal brukes for å kjøre jobben. Her velger vi node etter labelet docker, som vi satt opp på jenkins-master tidligere. Vi må også legge til dette labelet ved hvert steg, slik at det blir kjørt på riktig node. Vi definerer også to miljøvariabler. Den ene inneholder registryet vårt, og den andre er en cache-mappe for Go. Videre kommer et Stages objekt som inneholder de forskjellige stadiene i jobben vår. Vi har fire stadier; build, test, publish og deploy, og vi vil gå gjennom dem en etter en. Build stagen ser slik ut

```
stage('Build') {
    agent {
        docker {
            image 'golang'
            label 'docker'
        }
    }
    steps {
        sh 'cd ${GOPATH}/src'
        sh 'mkdir -p ${GOPATH}/src/hello-world'
        sh 'cp -r ${WORKSPACE}/* ${GOPATH}/src/hello-world'
        sh 'go build'
    }
}
```

Her bruker vi Docker som agent med imaget golang. Det betyr at det vil kjøres en Go-container for å kjøre kode skrevet i Go. Under steps har vi de forskjellige kommandoene som kjøres for å bygge applikasjonen. Dersom det er feil i koden vil byggingen, og dermed Videre kommer Test-stagen

```
stage('Test') {
    agent {
        docker {
            image 'golang'
            label 'docker'
        }
    }
    steps {
        sh 'cd ${GOPATH}/src'
        sh 'mkdir -p ${GOPATH}/src/hello-world'
        sh 'cp -r ${WORKSPACE}/* ${GOPATH}/src/hello-world'
        sh 'go clean -cache'
        sh 'go test ./... -v -short'
    }
}
```

Også her bruker vi en Go-container for å kjøre koden. Go har en innebygget test-funksjon som brukes her for å teste applikasjonen. Hvis den feiler vil pipelinen stoppes. Når testen er fullført skal det bygges og publiseres et docker-image i Publish-stagen

```
stage('Publish') {
    agent {
        label 'docker'
    }
    environment {
        registryCredential = 'gitlab-cred'
    }
    steps {
        script {
            def appimage = docker.build('${registry}:$BUILD_NUMBER')
            docker.withRegistry( 'https://${registry}', registryCredential )
{
                appimage.push()
                appimage.push('latest')
            }
        }
    }
}
```

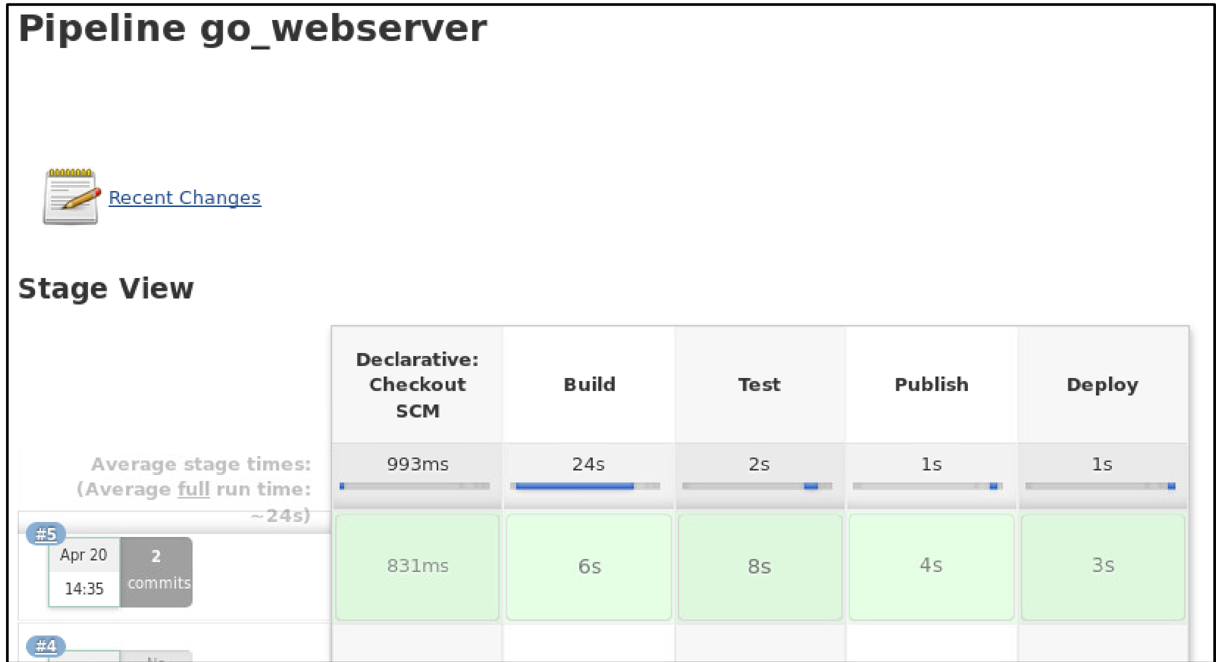
I denne stagen definerer vi en miljøvariabel. Det vi definerer her er navnet på credentials som vi laget for brukernavn og passord til gitlab. De brukes når vi pusher imaget for å få tilgang til registryet. Først bygger vi et image som får samme versjonsnummer som byggenummeret i jenkins, så pusher vi både dette og en versjon med versjonsnummer latest. Dermed vil det være mulig å hente en spesifikk versjon av applikasjonen eller nyeste versjon med latest taget. I Deploy stagen skal applikasjonen rulles ut til Kubernetes.

```
stage('Deploy') {
    agent {
        docker {
            image 'roffe/kubectl'
            args '-u 0:0'
            label 'docker'
        }
    }
    steps {
        sh 'kubectl apply -f ${WORKSPACE}/k8s/namespace.yaml'
        sh 'sed -i "s/test:latest/test:$BUILD_NUMBER/g"
${WORKSPACE}/k8s/goweb.yaml'
        sh 'kubectl apply -f ${WORKSPACE}/k8s/goweb.yaml'
    }
}
```

Her bruker vi igjen Docker med et image som kjører Kubectl. Containeren får en kopi av filene til Jenkins-poden, og vil bruke den konfigurasjonen vi mountet i Poden når vi laget den. Vi spesifiserer også hvilken bruker vi skal kjøre containeren som. I dette tilfellet er det bruker 0, som er root. Dette steget vil bruke de konfigurasjonsfilene som finnes for å rulle ut

applikasjonen i kubernetes. Dersom filene ikke er endret vil de ikke gjøre noen endringer. Først oppretter den Namespacet for applikasjonen. Vi gjør dette først fordi det ofte kan bli problemer når man prøver å lage flere ressurser uten at Namespacet de skal lages i eksisterer. Det neste som gjøres er å bytte versjonsnummeret til det nyeste, så rulles alt ut til clusteret. I de fleste tilfeller er det bare versjonen av imaget som vil endres, men alle andre endringer som er gjort vil også rulles ut her.

Når filene er klare kan vi pushe dem til GitLab, som setter i gang jobben i Jenkins. Hvis vi ser på jobben vil vi etter en stund se at alle stadiene ble gjennomført.



Vi kan også gå til adressen goweb.example.com og se at vi får et svar

Til sist kan vi liste ut alle ressursene i Namespacet go-webserver og se at de er blitt rullet ut som de skal.

```
simen@k8s-master:~$ kubectl get all -n go-webserver
NAME                                   READY   STATUS    RESTARTS   AGE
pod/hello-deployment-5c9d77b9dd-c6qj5  1/1     Running   0          8m20s
pod/hello-deployment-5c9d77b9dd-wrtx4  1/1     Running   0          8m20s

NAME                TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/hello-svc   ClusterIP   10.97.53.114   <none>        80/TCP    8m20s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/hello-deployment   2/2     2            2           8m20s

NAME                                         DESIRED   CURRENT   READY   AGE
replicaset.apps/hello-deployment-5c9d77b9dd  2         2         2       8m20s
```

Alle filene som er brukt i eksempelet ligger vedlagt under `6_3_2_4_go-web`

7. Kilder og videre lesning

[1] Kubernetes. (2020, 21. april). Nodes. Hentet fra
https://kubernetes.io/docs/concepts/architecture/nodes/

[2] Kubernetes. (2020, 15. mars). Pods. Hentet fra
https://kubernetes.io/docs/concepts/workloads/pods/pod/

[3] Kubernetes. (2020, 21. april). Service. Hentet fra
https://kubernetes.io/docs/concepts/services-networking/service/

[4] Kubernetes. (2020, 7. mai). Ingress. Hentet fra
https://kubernetes.io/docs/concepts/services-networking/ingress/

[5] Kubernetes. (2020, 12. mai). Persistent Volumes. Hentet fra
https://kubernetes.io/docs/concepts/storage/persistent-volumes/

[6] Kubernetes. (2020, 30. april). Secrets. Hentet fra
https://kubernetes.io/docs/concepts/configuration/secret/

[7] Kubernetes. (2020, 18. mai). Deployments. Hentet fra
https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[8] Kubernetes. (2020, 14. mai). StatefulSets. Hentet fra
https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

[9] Kubernetes. (2020, 21. april). DaemonSet. Hentet fra
https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

[10] Kubernetes. (2020, 9. april). Using RBAC Authorization. Hentet fra
https://kubernetes.io/docs/reference/access-authn-authz/rbac/

[11] Kubernetes. (2020, desember). Kubernetes NFS-Client Provisioner. Hentet fra
https://github.com/kubernetes-incubator/external-storage/tree/master/nfs-client

[12] Kubernetes. (2020, 15. februar). Run a Replicated Stateful Application. Hentet fra
https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/

[13] Kubernetes. (2020, 12. mai). Installing kubeadm. Hentet fra
https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

[14] Kubernetes. (2020, 12. mai). Container runtimes. Hentet fra
https://kubernetes.io/docs/setup/production-environment/container-runtimes/

[15] Kubernetes. (2020, 11. mai). Creating a single control-plane cluster with kubeadm.
Hentet fra https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/

[16] Helm. (2020, 15. mai). Prometheus Operator. Hentet fra
https://github.com/helm/charts/tree/master/stable/prometheus-operator

[17] GitLab. (u.å.). GitLab Operator. Hentet fra
https://docs.gitlab.com/charts/installation/operator.html

[18] Casey, K. (2019, 11. februar). How to explain Kubernetes Operators in plain English.
Hentet fra https://enterprisersproject.com/article/2019/2/kubernetes-operators-plain-english

[19] Ahmed, M. (2019, 9. desember). Creating Custom Kubernetes Operators. Hentet fra
https://www.magalix.com/blog/creating-custom-kubernetes-operators

Magnus Torset
Simen Sagholen Førrisdal

Kubernetes i praksis

Sluttrapport

1. Sammendrag

I løpet av dette prosjektet har vi satt opp et komplett Kubernetes cluster for utrulling og drift av applikasjoner. Alle maskinene som er brukt er virtuelle maskiner i vSphere, men behandles som bare-metal i forhold til installasjon og konfigurasjon. Selve clusteret består av fire Debian maskiner som kjører Docker og Kubernetes, i tillegg til at vi har satt opp flere tjenester ved siden av slik som NFS-server og DNS-server. Innad i Kubernetes har vi satt opp flere interne tjenester for å utføre flere cluster-oppgaver som normalt sett håndteres av en skyleverandør, slik at clusteret blir fullstendig. Gjennom en rekke eksempler forsøker vi også å vise hvordan clusteret kan tas i bruk til å sette opp og drifte ulike applikasjoner og tjenester. Driftsdokumentet skal fungere som en dokumentasjon og guide til Kubernetes og hvordan det kan brukes i praksis.

2. Endringer underveis

Vi ble nødt til å endre på problemstillingen vår midt i oppgaven. Den 11.03 hadde vi møte med veileder for å diskutere hva vi skulle gjøre videre. Den gamle problemstillingen vår, som handlet om å sammenligne hvordan det var å kjøre en enkel tjeneste, slik som en webserver eller database, på tradisjonelle VMer og i Kubernetes, ville ikke fungere. Vi fikk satt opp tjenestene i Kubernetes ganske enkelt, men vi fikk problemer med å generere nok trafikk til webserveren og med å måle resultatene i Kubernetes på grunn av begrensninger i måten de viser og eksponerer ressursbruk til API-et på. Vi så ingen umiddelbar måte å løse dette på så vi bestemte oss for å endre på problemstillingen. Vi så også at problemstillingen, sånn den var skrevet, var for liten. Dette gjorde at vi var nødt til å endre problemstillingen.

Problemstillingen vi endte opp med var en som bygget videre på ting vi allerede hadde gjort og satte mye mer fokus på Kubernetes som et verktøy og hvordan dette kan brukes på forskjellige måter istedenfor å kun fokusere på spesifikke deler. Vi bestemte oss for å prøve å lære oss så mye som mulig om Kubernetes og bruke dette til å lage et dokument som kan hjelpe andre å starte opp med Kubernetes. Siden vi ikke lenger hadde et spesifikt miljø vi skulle sette opp for å løse en spesifikk oppgave, følte vi og veileder heller ikke at det var nødvendig med et designdokument. De punktene som var relevante i designdokumentet ble lagt inn, enten i forstudierapporten eller driftsdokumentet, avhengig av hvor det passet best.

3. Arbeidsprosessen

Generelt sett har arbeidet med prosjektet gått bra. Vi har jobbet jevnt gjennom hele prosjektet og hatt stabil fremdrift hele tiden. Innad i prosjektgruppen har vi møttes digitalt tre til fire ganger i uken for å jobbe med prosjektet, og vi har hatt møter med veileder rundt hver andre uke med noe variasjon.

Vi startet på forstudiet tidlig, og ble også ferdig med det etter relativt kort tid. Vi hadde enda ikke fått tilgang til hypervisor fra oppdragsgiver, men begynte i stedet oppsett på egne maskiner for å lære Kubernetes. Det gikk en stund før vi fikk tilgang på de nødvendige ressursene. Dette førte til at arbeidet til tider stod litt stille og vi kom i gang med selve installasjonen litt senere enn vi hadde ønsket. Etter hvert som vi fikk tilgang til vSphere startet vi oppsett av det endelige systemet og startet arbeidet på designdokumentet. Underveis i prosessen kom vi frem til at den daværende problemstillingen var vanskelig å

gjennomføre med gode tester, og etter møte med veileder kom vi frem til en ny problemstilling som enklere kunne gjennomføres. Her kom vi også frem til at designdokumentet skulle settes til side og til sist tas ut av prosjektet. I stedet startet vi arbeidet på driftsdokumentet, og mye av det vi startet å skrive om i designrapporten ble lagt inn her som eksempler. Den nye problemstillingen var mer gjennomførbar, men også veldig bred. Vi støtte etter hvert på problemer hvor det var uklart hvor vi skulle ta prosjektet videre og hva oppgaven dekket, da flere av eksemplene går inn på ting som ikke direkte har med Kubernetes å gjøre. Det var også vanskelig å finne et godt sluttpunkt for oppgaven, men med hjelp fra veileder kom vi til slutt frem til et endepunkt for prosjektet.

4. Hva ville vi gjort annerledes

Hadde vi skulle ha begynt på nytt i morgen ville vi nok valgt oss en problemstilling med mer konkrete rammer og mål. Sånn den er skrevet nå er den veldig åpen, som er bra siden vi ikke blir begrenset av den hvis det er noe vi ønsker å gjøre med Kubernetes, men det er vanskelig å både vite hva neste steg er og å finne ut når vi er ferdig.

5. Videre arbeid

Systemet vi har satt opp i løpet av dette prosjektet er et fungerende Kubernetes cluster som er klart til bruk. Den grunnleggende installasjonen sammen med enkelte tjenester fra de ulike eksemplene setter opp et komplett cluster som har de fleste funksjonene man har bruk for. Hva man ønsker å gjøre videre vil være veldig individuelt avhengig av hva man ønsker å gjøre og hvilke behov man har. I utgangspunktet kan clusteret brukes til hva som helst, og driftsdokumentet vil forhåpentligvis gi den nødvendige kunnskapen til å sette opp de ressursene man trenger.

En ting vi ikke går spesielt inn på i driftsdokumentet er lastbalansering, som kan gjøres på flere ulike måter. Dersom vi skulle jobbet videre med dette prosjektet ville vi gått mer i dybden på hvordan de ulike typene lastbalansering fungerer og sammenlignet dem for ulike use-caser

NTNU
Kunnskap for en bedre verden