Industriell IoT og edge computing med Azure Cloud standardkomponenter

Fredrik Frostad
Ole-Martin Heggen
Michael Mobæk Thoresen
Jakob Simonsen

24. mai 2020

Institutt for Informasjonsteknologi
Postadresse: Postboks 4 St. Olavs plass, 0130 Oslo
Besøksadresse: Holbergs plass, Oslo

Telefon: 22 45 32 00

PROSJEKT NR.
7

TILGJENGELIGHET
Åpen

BACHELORPROSJEKT

HOVEDPROSJEKTETS TITTEL	DATO
Industriell IoT og edge computing med Azure Cloud standardkomponenter	24.05.2020
	ANTALL SIDER / BILAG 127/3
PROSJEKTDELTAKERE Fredrik Frostad (s325853) Ole-Martin Heggen (s325905) Jakob Bogen Simonsen (s325908) Michael Mobæk Thoresen (s325903)	INTERN VEILEDER Eva Hadler Vihovde

OPPDRAGSGIVER	KONTAKTPERSON
Kongsberg Digital AS	Kåre Langedrag

SAMMENDRAG

I løpet av dette prosjektet på OsloMet – Storbyuniversitetet har vi implementert et funksjonelt proof-of-concept system for industriell IoT basert på Azure standardkomponenter. Formålet med prosjektet er å evaluere kostnad og ytelse for et slikt system, for å gi oppdragsgiver et bredere beslutningsgrunnlag ved en eventuell migrering fra egenutviklede komponenter til Azure standardkomponenter.

3 STIKKORD
Internet of Things
Skytjenester
Edge Computing

Forord

Dette dokumentet er sluttrapporten for vårt bachelorprosjekt i ingeniørfag data ved OsloMet - Storbyuniversitetet, vårsemesteret 2020. Rapporten beskriver arbeidet med vårt evalueringsprosjekt av et IoT system bygget med Azure standardkomponenter.

Som følge av at dette prosjektet er en research-oppgave, er det ikke et mål og utvikle et produksjonsklart system med et polert brukergrensesnitt. Systemet som er utviklet av prosjektgruppen er etter oppdragsgivers spesifikasjon å anse som en teknologi demonstrator og testplatform, ment for evaluering av Microsoft Azure komponentene som inngår i denne.

Rapporten følger ikke OsloMets dokumentmal, men er skrevet på et format som vi har utarbeidet i samarbeid med vår interne veileder. Rapporten er optimalisert for digital lesning og inneholder pekere til tilleggsinformasjon og utdypende seksjoner i teksten der det er relevant. Dersom det er ønskelig å lese rapporten i papirformat er dette uproblematisk.

Rapporten legger til grunn at leseren har grunnleggende forståelse for programmering, informasjonsteknologi og skytjenester, da dette er sentrale elementer i prosjektet. Til tross for at rapporten er skrevet med dette som premiss, har vi forsøkt å forklare begreper, problemstillinger og løsninger på en slik måte at det skal kunne forstås av et bredt utvalg av lesere.

I teksten vil det forekomme lenker til andre seksjoner i rapporten, disse lenkene vil ha blå tekst. Alle oppføringer i rapportens innholdsfortegnelse er klikkbare lenker som tar leseres til korrekt side i rapporten. I rapporten refererer vi også til en rekke eksterne kilder. Komplett kildehenvisning følger med rapporten som vedlegg, og all referanser til denne er markert med tall i klammeparenteser [1]. Referansene er klikkbare lenker og vil ta leseren til korrekt oppføring i referanselisten.

Det er lagt ved en ordliste som forklarer fremmedord og teknologier som omtales i rapporten. Det vil i teksten lenkes til ordlisten der det er relevant.

Innhold

Kapittel 1

Innledning

Vår bachelorgruppe ble dannet våren 2019. Vi har arbeidet godt sammen gjennom hele studiet, og våre respektive interesse og kunnskapsområder utfyller hverandre godt. Da arbeidet med å finne en oppgave ble iverksatt tidlig høst 2019, hadde vi fremdeles ikke klart for oss hva vi spesifikt ønsket å jobbe med, men vi hadde et par kriterier klare:

- Vi ville jobbe med teknologi vi ikke hadde tidligere erfaring med fra undervisning
- Vi ville ha en utfordrende oppgave

Vi tok kontakt med en rekke potensielle oppdragsgivere, og etter samtaler med de mest aktuelle kandidatene, landet vi til slutt på et prosjekt fra Kongsberg Digital AS. Bakgrunnen for at vi valgte akkurat dette prosjektet er at vi anså Kongsberg Digital som en interessant og attraktiv oppdragsgiver, og at prosjektet gav oss muligheten til å få erfaring innen flere nye fagområder, samt at vi får muligheten til å arbeide med svært relevant og fremtidsrettet teknologi.

Oppgaven er i all hovedsak et research prosjekt, der vi har satt opp, sikret, ytelsestestet og evaluert et system for industriell IoT basert på Azure standardkomponenter. Hovedformålet med prosjektet har vært å evaluere systemets ytelse med hensyn på ende-til-ende dataflyt, sikkerhet, og kostnad. Oppdragsgiver har ønsket at det rettes spesielt fokus mot å evaluere lagringsløsninger for tidsseriedata, med utgangspunkt i en sammenligning av dokument vs relasjonsdatabase.

1.1 Ekstraordinære omstendigheter

I likhet med de aller fleste bachelorprosjekter som har blitt avholdt dette vårsemesteret, har også vårt prosjekt blitt påvirket av de omfattende smittevernstiltakene Norge har vært igjennom. Samarbeidet internt i gruppen har heldigvis ikke blitt påvirket i negativ grad, og alle gruppens medlemmer har båret sin del av arbeidsbyrden gjennom hele prosjektet. Men vi vil også si at det har vært utfordrende å gjennomføre prosjektet uten å kunne møtes fysisk i lange perioder og med begrenset kontakt med oppdragsgiver.

1.2 Aktører

I dette delkapittelet vil vi presentere de viktigste aktørene som har hatt innvirkning på prosjektets sluttresultat.

1.2.1 Oppdragsgiver

Figur 1.1: Kongsberg Digital AS

Vår oppdragsgiver, Kongsberg Digital[1] er et selskap i Kongsberg Gruppen som leverer programvare og tekniske løsninger til gruppens kunder. Kongsberg Digital ble etablert 17. mars 2016 og har i dag over 480 ansatte. Kongsberg Gruppen har i sin helhet over 11000 ansatte fordelt i gruppens verdensomspennende virksomheter. Kongsberg digital ble etablert for å fungere som et samlingspunkt for all digital kompetanse i Kongsberg Gruppen, og for å hjelpe de industrielle aktørene Kongsberg Gruppen leverer tjenester til med digital omstilling. Kongsberg Digital leverer i all hovedsak teknologiske løsninger til kunder innen følgende områder:

- Handelsflåten
- Olje og gass
- Fornybar energi og kraftforsyning

Selskapets kjerneområder er IoT, kunstig intelligens, stor-data innsikt, maritim simulering, automatisering og autonome operasjoner. En av selskapets mest kjente løsninger er den digitale plattformen Kognifai[2]. Kognifai gir kunder tilgang til bla. avansert datadrevet ressursstyring og Dynamisk Digital Tvilling teknologi.

1.2.2 Prosjektgruppen

Vår gruppe består av fire dataingeniørstudenter. Alle gruppens medlemmer har relevant arbeidserfaring ved siden av studiene. To av gruppens medlemmer jobber som utviklere, en er systemadministrator og en jobber som sikkerhetsanalytiker.

Fredrik Frostad Fredrik er dataingeniørstudent ved OsloMet og har tidligere arbeidet som prosjektleder og lydtekniker hos Bright Norway AS. Fra denne stillingen har Fredrik bred erfaring med prosjektering og gjennomføring av større prosjekter innen konserter, corporate-events og TV-produksjon. Fredrik har hatt sommerjobb som backend java utvikler hos Oslo Market Solutions, og jobber i dag deltid i samme bedrift. Fredrik starter som konsulent hos Solidsquare AS høsten 2020.

Ole-Martin Heggen Ole-Martin er dataingeniørstudent ved OsloMet og har tidligere utdanning som befal og dataelektroniker lærling i cyberforsvaret. Ole-Martin jobber i dag deltid hos Computas på avdeling for IT og analyse hvor han også har hatt sommerjobb. Stillingen hans innebærer systemadministrasjon, bruker supportering, Active Directory og Google Cloud Platform administrator. Han vil starte som konsulent i Sopra Steria høsten 2020. Ole-Martin har ellers en interesse innenfor CTF og DIY hjemme IoT.

Michael Mobæk Thoresen Michael er dataingeniørstudent ved OsloMet og jobber deltid som sikkerhetsanalytiker hos BDO. Michael har erfaring med Blue Team sikkerhet igjennom jobben hos BDO og noe erfaring med Red Team sikkerhet igjennom diverse CTF-konkurranser og aktiviteter som OverTheWire og HackTheBox. Michael planlegger etter fullført bachelorgrad å gå videre på en mastergrad innen informasjonssikkerhet på UiO.

Jakob Bogen Simonsen Jakob jobber som studentutvikler hos IBM og er studentassistent i faget programmering. Han har hovedsaklig jobbet med webapplikasjoner, distribuerte systemer og sky-løsninger. Siden 3. semester på dataingeniør studiet, har han jobbet 70 prosent deltid og gjennomført prosjekter på alt fra maskinlæringsmodeller til verdikjede-løsninger med blockchain. Sist sommer hadde Jakob summer internship hos IBM. Til høsten har Jakob søkt en mastergrad innen datateknologi ved NTNU.

1.2.3 Kontaktpersoner

Eva Hadler Vihovde Eva har vært vår interne veileder gjennom prosjektperioden, og har vært til stor hjelp for oss. Vi oppsøkte henne som veileder etter å ha fått gode anbefalinger fra tidligere bachelorgrupper hun har veiledet. Eva har hjulpet til med planlegging og strukturering av både prosjektet og prosjektrapporten, og har gjennom hele perioden gitt oss råd om fremdriften videre. Eva foreleser i Diskret Matematikk, Programmering og Testing ved OsloMet og UiO.

Kåre Langedrag - Cybersecurity Architect, Lead on technical cybersecurity architecture on platform and edge. Kåre har vært vår primære kontaktperson hos Kongsberg Digital. Han har vært ansvarlig for å følge oss gjennom prosjektperioden, og har kommet med jevnlige innspill under arbeidet med oppgaven. Kåre har også satt oss i kontakt med personer internt i Konsgberg Digital som har hatt bidratt med domenekunnskap nødvendig for utførelsen av prosjektet. Kåre har sørget for at vi har hatt tilgang til nødvendige ressurser gjennom hele prosjektperioden.

Marco Constantino - Cybersecurity Specialist, Lead on cybersecurity training and awareness. Marco har sammen med Kåre vært tett involvert i prosjektet og har ved siden av Kåre vært den hos Konsgberg Digital vi har hatt tettest kontakt med. Marco var svært sentral under onboarding prosessen i starten av prosjektet, og har gitt oss svært viktig opplæring i de forskjellige systemene vi har brukt under arbeidet med oppgaven. Marco har også vært en viktig ressurs for å løse utfordringer med tilganger og privilegier i Azure Cloud i prosjektperioden.

1.3 Bakgrunn for oppgaven

Kongsberg Digital er med sin platform KognifAI en internasjonalt etablert aktør innen IoT i industrien, og tilbyr gjennom KognifAI en rekke løsninger for data-drevet innsikt til sine kunder. Kongsberg Digital bruker Microsoft Azure som leverandør av skytjenester og ønsker i den sammenheng å utrede hvilke muligheter som ligger å benytte standardkomponenter fra Azure i sin IoT-platform. Kongsberg Digital har i dag en stor grad av skreddersydde komponenter i sin teknologiplattform. De ønsker et beslutningsgrunnlag for å vurdere om det vil være fordelaktig å ta i bruk en PAAS strategi for sine løsninger. Tanken er at dette vil frigjøre ressurser som i dag benyttes til å vedlikeholde, oppdatere og sikre eksisterende kodebase, til å fokusere sterkere på utvikling av Kongsberg Digitals forretningslogikk.

1.4 Dagens situasjon

Da Kongsberg Digital ble stiftet i 2016 hadde Microsoft akkurat lansert deler av sine IoT-løsninger, men viktige komponenter slik som selve IoT Edge ble først lansert i 2018. Den første utgaven av IoT Edge tilfredsstilte ikke de kravene som Kongsberg Digital hadde til sin plattform, og det fantes ingen skytjenester som kunne lagre og prosessere tidsseriedata på en kostnadseffektiv måte. Det har skjedd en stor utvikling i Microsoft sine IoT-løsninger de siste to årene, og for at Kongsberg Digital skal få et bedre beslutningsgrunnlag for å vurdere hvilke av Microsoft sine IoT-løsninger som potensielt kan erstatte egenutviklet programvare, ønsker de at det etableres et full ende-til-ende IoT-system basert kun på Microsoft Azure skytjenester.

En demonstrasjon av et slik ende-til-ende-system anses som dekkende for å demonstrere om de funksjonelle kravene til systemet er innfridd. Videre ønsker KDI at det gjøres målinger for å kalkulere kostnad og ytelse for å kunne få et bedre grunnlag for å vurdere hvilke komponenter som eventuelt burde erstattes.

I løpet av prosjektperioden har vi i prosjektgruppen implementert et funksjonelt ende-til-ende system som svarer til de krav fremstilt av oppdragsgiver. Vi har brukt implementasjonen av dette systemet som grunnlag for arbeidet vi har gjort for å evaluere både systemet som helhet, og dets enkeltkomponenter. Denne rapporten beskriver i detalj det implementerte systemet, prosessen med å evaluere nevnte system, og resultatene vi har kommet frem til.

1.5 Rapportens oppbygning

For å gjøre det enklere for leseren å navigere i rapporten, vil vi i dette delkapittelet gjennomgå rapportens oppbygging, samt gi en kort forklaring av de ulike kapitlenes innhold.

Prosjektrapporten består av følgende deler:

1. Innledning

Innledningen inneholder en presentasjon av prosjektet. Her presenteres aktørene; oppdragsgiver, veileder og prosjektgruppen. Innledningen beskriver bakgrunnen for prosjektet og det anbefales at dette kapittelet sammen med kapittel 2, problemstilling og mål, leses i sin helhet.

2. Problemstilling og mål

Dette kapittelet definerer rammene for oppgaven, hvilke krav og forventninger oppdragsgiver har, og hvilke mål prosjektgruppen har satt seg for gjennomføringen av oppdraget. Her finner man rammebetingelser og krav til systemet.

3. Gjennomføringsprossess

Dette er rapportens prosessdokumentasjon. Kapittelet omhandler hvordan vi har arbeidet med oppgaven, verktøy vi har brukt for å strukturere arbeidet, samt beskrivelse av metode brukt for evaluering av systemets ytelse og kostnad.

4. Systembeskrivelse

Dette kapittelet er en høynivå beskrivelse av komponentene som inngår i systemet og sammenhengen mellom disse. Dette kapittelet er har som formål og gi leseren en forståelse av systemets sammensetning før lesning av kapittel 5 og 6.

5. Resultater

I dette kapittelet presenterer vi resultatene fra de syntetiske belastningstestene vi har gjort for å evalurere de enkelte komponenters, og systemets som en helhet, ytelse sett opp i mot kostnad. Vi gjennomgår hver testfase i detalj, og presenterer et konklusjons-avsnitt for hver gjennomførte testfase.

6. Drøfting og analyse

Dette kapittelet er en sammenstilling av resultatene presentert i kapittel 5. Her vil vi presentere våre synspunkter og slutninger basert på datagrunnlaget vi har samlet inn gjennom testkjøringer. Vi vil også drøfte arbeidsprosessen og muligheter for videreutvikling av funksjonalitet i systemet.

7. Konklusjon

Dette kapittelet oppsummerer arbeidet utført i prosjektperioden. Vi vil med dette kapittelet gi leseren en oversikt og oppsummering over det viktigste innholdet i oppgaven.

8. Testing

Her presenterer vi en testrapport for alle komponenter som inngår i systemet utviklet av prosjektgruppen. Vi går igjennom alle tester som er utført, og resultatene av disse.

9. Systemet - Teknisk beskrivelse, arkitektur og implementasjonsdetaljer

Detaljert teknisk beskrivelse av systemet vi har implementert som grunnlag for å kunne evaluere systemet, og dets komponenters, ytelse og kostnad.

10. Teknologier og verktøy

Oversikt over teknologier og verktøy prosjektgruppen har benyttet seg av igjennom prosjektperioden.

Vedlegg

Ordliste og kodeeksempler. Relevant egenutviklet kode er også vedlagt oppgaven som et zip-arkiv. Merk at all kode utviklet under prosjektet er Kongsberg Digitals eiendom, og kan kun brukes i vurderingsprosessen av oppgaven.

Kapittel 2

Problemstilling og mål

I dette kapittelet vil vi presentere vår problemstilling for oppgaven. Her vil også mål og rammebetingelser for prosjektet presiseres og utdypes.

2.1 Problemstilling

Industrien er i dag inne i en omfattende digital transformasjon. Dette omtales som den 4. industrielle revolusjon, og er basert på intelligente fabrikker og prosesser som er koblet sammen i skyen via tingenes internett (IoT).[3, p. 3] Den teknologiske utviklingen på feltet går i et hurtig tempo, og der man for få år siden var avhengig av egenutviklede løsninger, tilbys det i dag platform-as-a-service produkter som tillater bruker å flytte fokus fra vedlikehold av infrastruktur til videreutvikling av sin forretningslogikk.[4, p. 1]

Kongsberg Digital bruker i dag en stor del egenutviklede produkter for å tilby datadrevet innsikt til sine kunder. Hva om man kunne migrere noen av disse produktene over til Azures skybaserte infrastruktur for IoT? Dette kan potensielt føre til kostnadsbesparelser, økt oppetid, samt frigjøre ressurser som i dag brukes på drift, til å utvikle funksjonalitet som skaper økt verdi for Kongsberg Digitals kunder.

Vi ønsket da vi startet dette prosjektet å undersøke hvilke muligheter og utfordringer som er knyttet til å implementere et rudimentært ende-til-ende IoT-system, basert på standardkomponenter som tilbys i Azure-cloud, herunder har vi rettet spesielt fokus mot følgende:

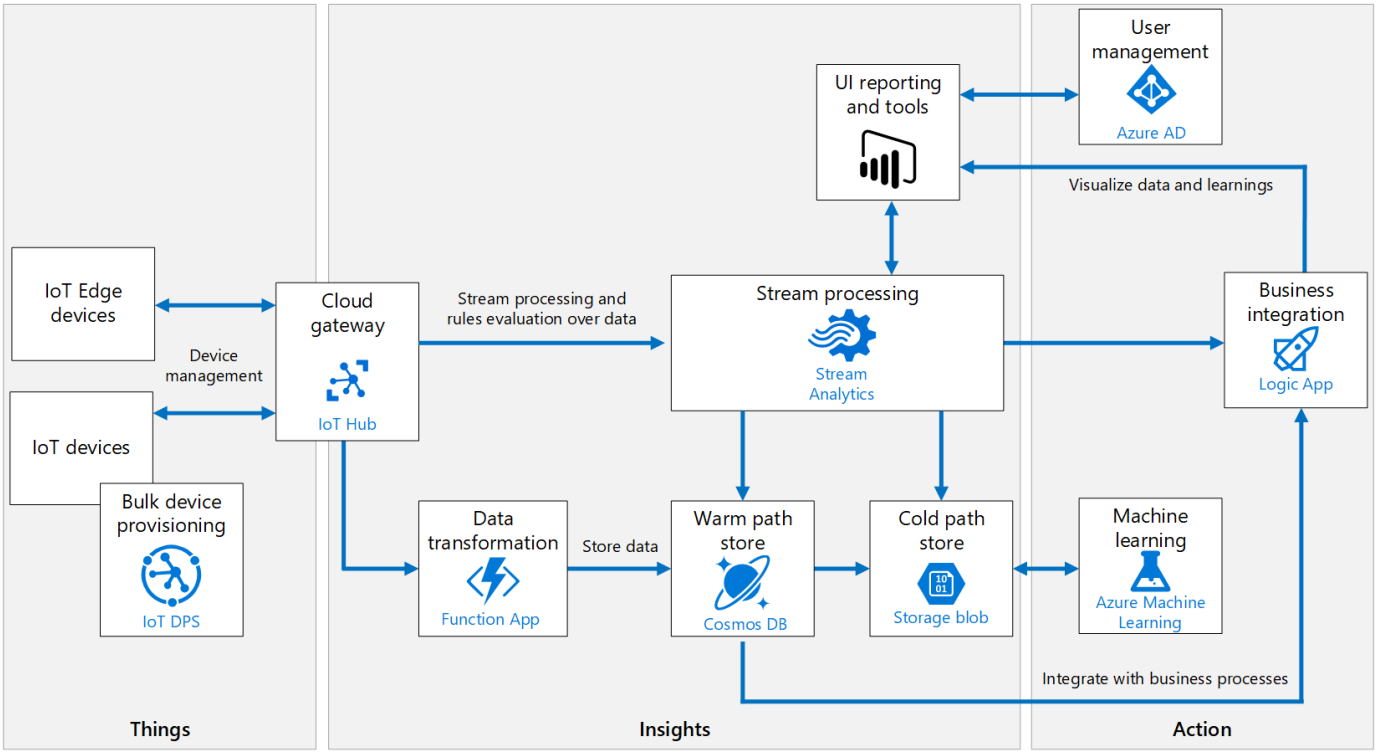
- Skybaserte lagringsløsninger for tidsseriedata og evaluering av kostnad for disse
- Utvikling av systemet basert på DevOps prinsipper.
- Bruk av LXC containere for utvikling av funksjonalitet rettet mot Azure IoT Edge, da det i dag er unødvendig høye kostnader knyttet til bruk av virtuelle maskiner i utviklingsøyemed.

2.2 Prosjektets mål

Det overordnede målet for prosjektet har vært å evaluere ytelse og kostnad i et system for industriell IoT, som i stor grad er basert på Azure standardkomponenter. For å kunne gjøre realistiske

målinger har vi implementert et ende-til-ende system som håndterer innsamling, lagring, prosessering og visualisering av telemetridata.

Vi har i dette prosjektet valgt å basere vårt demonstrasjonssystem på Microsofts referansearkitektur for industriell IoT i Azure(fig 2.1).



Figur 2.1: Referansearkitektur

Vårt system tar inn simulerte sensordata, via IoT Edge instanser, som kjører i LXC containere på en VM i Azure. Simulasjonsdata genereres av et python-program som simulerer en PLC, som igjen kommuniserer med IoT Edge instansen via ModbusTCP kommunikasjonsprotokoll. I IoT Edge mappes dataene til en JSON serialiserbar trestruktur og sendes videre til IoT Hub for inntak i skyen.

Etter inntak splittes datastrømmen opp i en varm og en kald lagringssti. Disse stiene inneholder samme data, men behandlingen av dataene er forskjellig. I den kalde stien skrives dataene direkte til Azure blob-storage for langtidslagring, mens data i den varme stien gjennomgår et transformasjonssteg, for deretter og persisteres i en database som tillater effektive spørringer mot dataene.

Data som ligger i varmlager databasen kan aksesseres for presentasjon via henholdsvis en klient og en backend-applikasjon som snakker direkte med databasen. Data som ligger i kaldlager databasen kan aksesseres for historikk formål og brukes til trening av maskinlæringsmodeller

2.3 Rammebetingelser

Hovedoppgaven i prosjektet har vært å utforske hvilke muligheter og utfordringer som ligger i å utvikle en IoT-platform kun ved bruk av standardkomponenter som er tilgjengelige i Azure

cloud. Dermed har vi hatt en svært stor grad av frihet til å utforske forskjellige teknologier og fremgangsmåter for å oppnå ønsket resultat. Oppdragsgiver har imidlertid gitt oss noen føringer for ønsket resultat, og vi har rettet arbeidet med oppgaven etter disse.

2.3.1 Ytelse og kostnad

Det ønskes at følgende metrikker evalueres:

- Kostnad pr IoT Edge på en Azure virtuell maskin
- Kostnad ved inntak pr telemetrimelding og hvordan dette skalerer når antall meldinger øker
- Kostnad pr databaseløsning ved lagring av telemetridata
- Ende-til-ende ytelse målt i antall messages håndtert pr tidsenhet

2.3.2 Proof-of-concept system

Gruppen skal implementere et proof-of-concept system i Microsoft Azure. Systemet skal bestå av følgende komponenter:

IoT Edge: en containerbasert komponent som skal kjøre på en Linux virtuell maskin(VM). IoT Edge vil kjøre på Ubuntu 18.04 LTS. Denne komponenten skal kommunisere med tilkoblede simulerte IoT-enheter via Modbus/TCP. IoT-Edge komponenten skal virtualiseres i Azure cloud for å underlette igangsetting av et høyt antall enheter, dette er for å kunne stressteste komponenter nedstrøms i systemet.

IoT-enhetssimulator: dette skal være en containerbasert enhet som enkelt kan simulere et stort antall enheter som kommuniserer med IoT-Edge over Modbus/TCP.

IoT Hub: Azure komponent som eksponerer skyen mot IoT Edge enheter. Denne enheten håndterer all kommunikasjon mellom Azure skyen og tilkoblede enheter. Denne enheten skal konfigureres slik at den oppfyller ytelseskravene fra oppdragsgiver (se avsnitt 2.4.4).

Cosmos DB: distribuert dokumentdatabase for varmlagring av bla. tidsseriedata. Data som lagres i databasen skal struktureres slik at det er mulig å ha flere forskjellige eiere av data i databasen uten at disse kan få tilgang til hverandres data.

PostgreSQL / TimescaleDB: SQL database med plugin for varmlagring av tidsseriedata. Data som lagres i databasen skal struktureres slik at det er mulig å ha flere forskjellige eiere av data i databasen uten at disse kan få tilgang til hverandres data.

Azure blob storage: filbasert langtidslagring av kalde data.

Azure Stream Analytics: pipeline for strømprossessering. Denne komponenten ser på datastrømmen som produseres av IoT Hub og utfører operasjoner mot denne.

Azure Time Series Insights: analyse, lagrings og presentasjons-service for spørring mot og utforskning av lagrede data fra IoT-enheter. Denne enheten tilbyr REST API-er for integrasjon mot web-applikasjoner(Webapp).

Webapp frontend: enkel webapp for visualisering av telemetridata. Skal kjøre på Azure app-service.

Webapp backend: denne komponenten skal skrives i .NET Core og tilbyr data fra database i varmlager database til frontenden. Skal kjøre på azure app-service.

2.4 Krav til systemet

Systemet som skal bygges er komplekst, og har mange komponenter som avhenger av hverandre. For å lettere holde oversikt over oppdragsgivers spesifikasjoner, velger vi å dele kravene inn i tre kategorier avledet fra referansearkitekturen (fig 2.1):

2.4.1 Datasimulering og Iot Edge

Denne kategorien av krav omhandler de delene av systemet som ligger oppstrøms for skyen.

- Oppdragsgiver ønsker mulighet for å teste tilkobling av 1 til 50 IoT Edge enheter mot skyen
- Provisioning av Edge enheter skal skje automatisk via Azure Device Provisioning Service(Azure DPS)
- Hver edge instans skal bestå av en LXC container avbildning, som inneholder en ferdig konfigurert edge modul og enhetssimulator-modul:
  - Hver enhetssimluator skal simulere 50 til 200 sensorer som sampler med frekvens = 1Hz
  - Edge enheten skal levere data til skyen mappet i en trestruktur.
  - Data skal sendes til skyen som json
- Edge enheten skal benytte følgende kommunikasjonsprotokoller mot IoT Hub:
  - MQTT over WebSockets
  - AMQP over WebSockets
  - HTTPS
  - Alle kommunikasjonskanaler skal benytte TLS
- Edge enheten skal takle bortfall av tilkobling:
  - Ved bortfall av tilkobling skal sensordata buffres i en gitt tidsperiode
  - Ved tilbakekomst av tilkobling skal varm data prioriteres fremfor buffrede data

2.4.2 Sky

Denne kategorien av krav omhandler de deler av systemet som ligger nedstrøms for edge enhetene. Oppdragsgiver har hatt følgende overordnede krav til skyløsningen og dens komponenter:

- Prosjektgruppen skal generere metrikker for:
  - Kostnad pr edge enhet
  - Kostnad pr data-enhet mottatt av IoT Hub
  - Kostnad / ytelse ved bruk av Azure Time Series Insights
  - Kostnad / ytelse ved bruk av Cosmos DB for varmlagring av data
  - Kostnad / ytelse ved bruk av PostgreSQL/Timescale for varmlagring av data
- Cosmos DB:

  Data skal struktureres slik at et multi tenant system er mulig
- Postgres Timescale DB:

  Data skal struktureres slik at et multi tenant system er mulig
- Datastier

  Varmlagring har prioritet

  Dersom dette er fornuftig skal kaldlagring håndteres av IoT Hub
- Persistering av varm tidsseriedata

  Det skal gjøres en evaluering av Cosmos DB vs PostGreSQL med Timescale plugin for persistering og uthenting av tidsseriedata.

2.4.3 Visualisering

Vi har valgt å plassere web applikasjonen under denne kategorien til tross for at den kjører i skyen, da all kommunikasjon med Azure komponenter skjer gjennom REST API-er. Oppdragsgiver har hatt få absolutte krav for denne delen av systemet, men har gitt oss følgende retningslinjer:

- GUI:
  - Data skal kunne fremstilles grafisk
  - Brukere skal ikke kunne få tilgang til annen data enn sin egen
  - Bruker skal kunne definere tidsvindu for visning av data
- Sky-til-Enhet kommunikasjon
  - Det er ikke fremsatt noen krav om sky-til-enhet kommunikasjon fra oppdragsgiver, men det er et ønske om at dette implementers dersom gruppen har kapasitet til det.

2.4.4 Ytelse og kostnad

Siden målet med dette oppdraget har vært en evaluering av systemets ytelse og kostnad er det ikke satt noen konkrete, faste ytelsesmål. Derimot har oppdragsgiver ønsket at prosjektgruppen skal evaluere ytelse og kostnad under følgende konfigurasjonsintervaller:

Fase 1: opp til 10 edge enheter som hver håndterer 200 sensorer. Igangsetting av enhentene skal automatiseres ved scripting.

Fase 2: opp til 20 edge enheter som hver håndterer 200 sensorer. Igangsetting av enhetene skal automatiseres ved scripting.

Fase 3: opp til 50 edge enheter som hver håndterer 200 sensorer. Igangsetting av enhetene skal automatiseres ved scripting.

2.4.5 Sikkerhet

Kongsberg Digital har ønsket at vi, hvis tilgjengelig tid tillater det, skal utføre en evaluering av sikkerhetsløsningene i Microsoft Azure sky plattformen med utgangspunkt i deres egne cybersikkerhetsretningslinjer. De retningslinjene som har påvirkt oss er:

- Håndheving HTTPS og TLS på eksponerte endepunkter i Azure
- Virtuelle maskiner skal ikke være eksponert på det offentlige nettet.
- Bruke Azure Active Directory der det er mulig
- Bruke sterke genererte passord

Kapittel 3

Gjennomføringsprosess - Gjenomføring av prosjektet

Det kan være utfordrende for en gruppe å arbeide effektivt i et større prosjekt med mange "bevegelige" deler og avhengigheter. Det er derfor viktig å bli enige om en felles arbeidsmetodikk i prosjektets planleggingsfase. Slik kan man på mest mulig effektiv måte, utnytte den tilgjengelige tiden i prosjektet. Underveis i prosjektet har vi ført en daglig prosjektdagbok som har hjulpet oss å holde rede på hvem som har gjort hva og når dette har blitt gjort. Vi vil i dette kapittelet gjøre rede for hvilke arbeidsmetodikker vi har fulgt, og hvilke prosess-styringsverktøy vi har benyttet oss av i utførelsen av prosjektet.

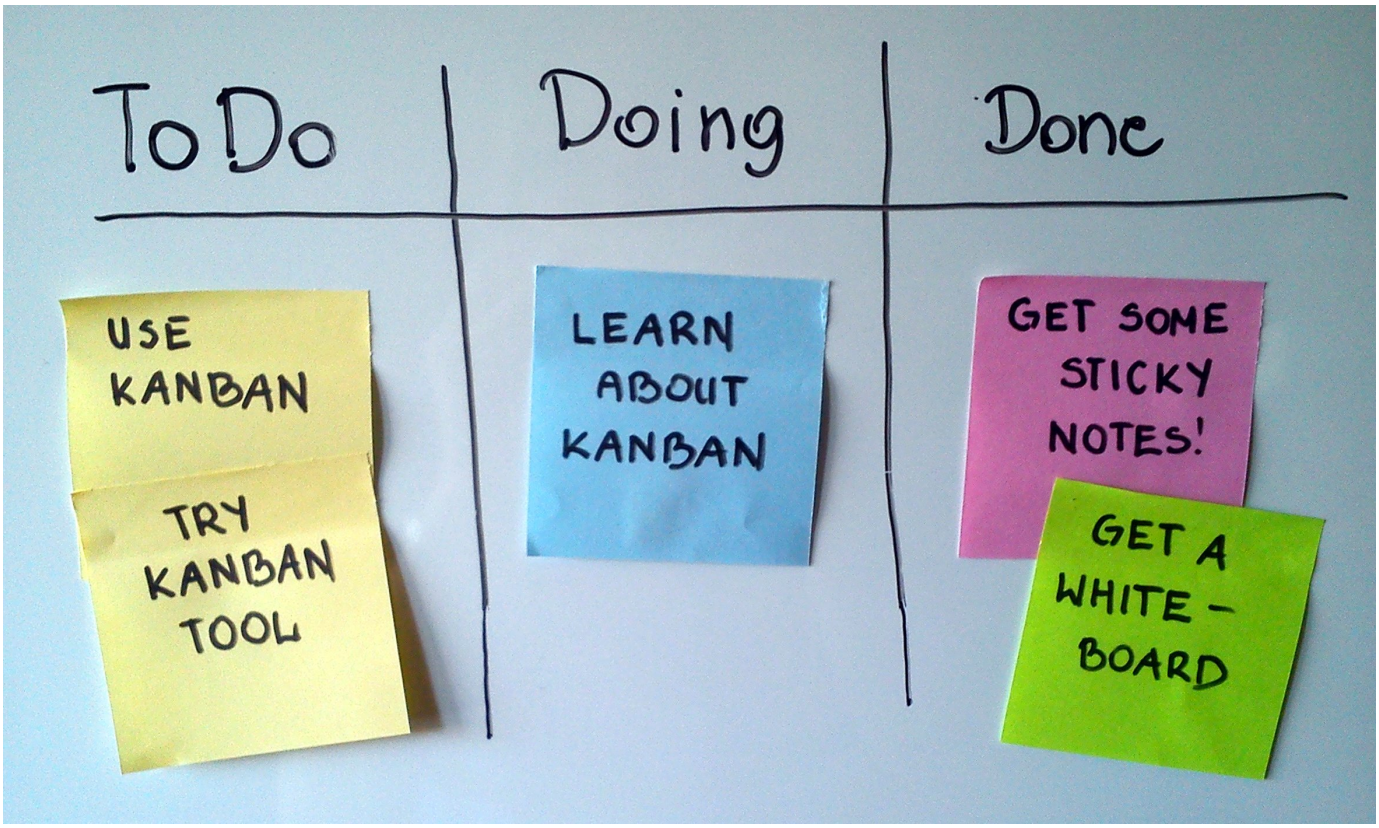
3.1 Arbeidsmetodikk ved implementasjon av systemet

Vi har arbeidet sammen som gruppe på flere større prosjekter som har inngått i studieprogrammet vi følger på Oslo-Met tidligere, og har fra disse prosjektene tatt med oss verdifull erfaring om arbeidsmetodikker som fungerer for oss. Vi forsto tidlig i arbeidet med oppgaven at det ville bli vanskelig å detaljplanlegge de ulike stegene, og vi kom derfor til enighet om å basere utviklingsprosessen på smidig metodikk[5]. Vi ville ikke låse oss til en bestemt variant av smidig utvikling tidlig i prosjektet, men basert på tidligere erfaringer kom vi frem til at en kombinasjon av elementer fra Kanban[6] og SCRUM[6] ville fungere godt for dette prosjektet.

Under arbeidet med oppgaven har vi delt uken i to. Mandag til onsdag har vi arbeidet sammen i Kongsberg Digitals lokaler, og torsdag til fredag har vi arbeidet hver for oss eller i mindre grupper basert på de gjeldende arbeidsoppgaver. Vi har operert med en kjernetid for arbeidet mellom kl 09:00 og 15:00. Dette har både tillatt oss å effektivt feilsøke problemer og diskutere utfordringer i fellesskap, samt gitt mulighet for individuelle gruppemedlemmer til å fordype seg i arbeidsoppgaver uten forstyrrende input fra øvrige gruppemedlemmer. Det har også vært svært verdifullt å ha mulighet til å arbeide på samme lokasjon som oppdragsgiver. Som et resultat av oppgavens åpne natur var det mange problemstillinger og avveininger som måtte avgjøres underveis, og det er enklere og langt mere effektivt å føre en muntlig diskusjon enn en skriftlig over nett.

Vi skjønte tidlig i prosessen at et viktig suksesskriterie ville være å bryte ned større oppgaver til mindre deler. For å oppnå dette, samt organisere og prioritere alle deloppgavene, valgte vi å

benytte oss av en Kanban-tavle. En kanban tavle er en viktig del av prosessmetodikken Kanban og fungerer på følgende måte;



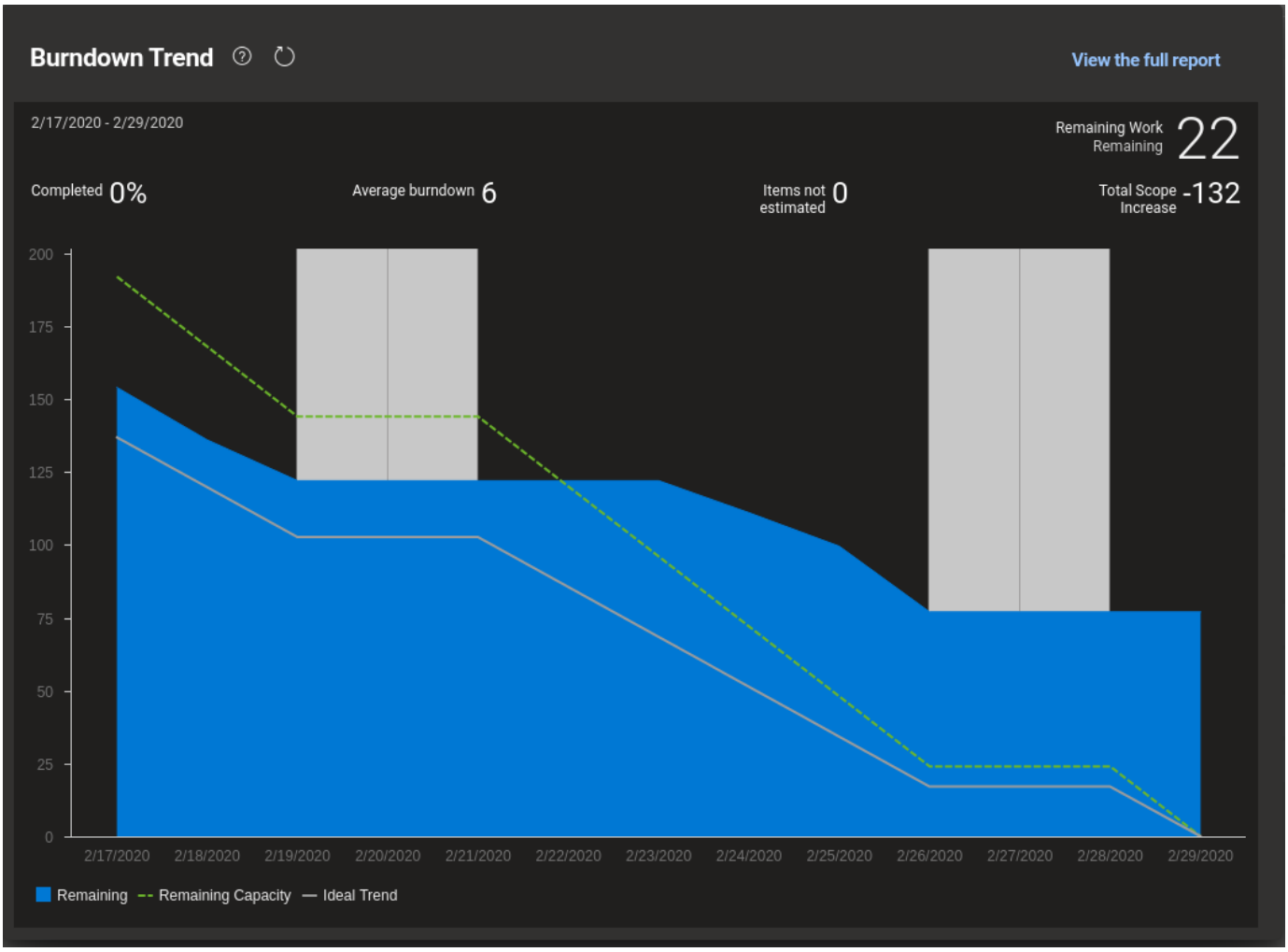
Figur 3.1: Eksempel på enkel kanban tavle[7]

Tavlen deles opp i seksjoner som beskriver en oppgaves status, eksempelvis *under arbeid*, *ikke påbegynt*, *ferdig*, og så videre. Nye oppgaver legges inn i en prioritert backlog, og herfra trekker man oppgaver som skal utføres og legger dem i *skal utføres* feltet. Oppgaven som utføres flyttes til *under arbeid* feltet mens man arbeider på den, og når den er fullført og godkjent flyttes oppgaven til feltet *fullført*. Deretter velger man en ny oppgave fra toppen av *skal utføres* feltet og gjentar prosessen.

For at denne metodikken skal virke etter hensikten, er det et viktig prinsipp man må ta hensyn til; nemlig at en oppgave kun skal bevege seg fra venstre mot høyre i tavlen. Det vil si at man ikke skal legge en oppgave som er påbegynt tilbake i backlog, eller i *skal utføres* feltet. Dette for at man skal unngå at halvferdige oppgaver ikke blir liggende, men heller utføres i sin helhet i det de er påbegynt. I noen tilfeller lar det seg ikke gjøre å fullføre en påbegynt oppgave, fordi denne blokkeres av en annen oppgave som ikke er fullført eller påbegynt. For å unngå at den påbegynte oppgaven ikke legges tilbake i backlog, eller i *skal* utføresfeltet kan denne legges i et eget felt med navn *blokkert*, og merkes med med en begrunnelse for hvorfor den ligger der. Da oppnår man at blokkeringer synliggjøres for gruppens øvrige medlemmer, og man unngår at en halvferdig oppgave forsvinner i backloggen.

Vi har også benyttet oss av en tilpasset form for SCRUM[8, p. 68-75] metodikk under arbeidet med oppgaven. For å ha kontroll på fremdriften i arbeidet, og for å unngå å bruke for mye tid på enkeltoppgaver valgte vi å dele arbeidet inn i SCRUM inspirerte sprinter med to ukers varighet.

For å ha kontroll på hvor mye arbeid vi realistisk ville klare å utføre i løpet av en sprint, estimerte vi antall påkrevde arbeidstimer for hver enkelt oppgave, samt hvert gruppemedlems arbeidskapasitet i timer i sprint perioden. Denne estimeringen hjalp oss også og holde styr på fremdriften internt i sprintene, ved hjelp av analyseverktøyene i Azure Devops (fig: 3.2). Etter hver endte sprint holdt vi også et sprint retrospektiv møte[8, p. 71] der vi evaluerte sprinten, og planla endringer som kunne hjelpe oss å forbedre neste sprint.

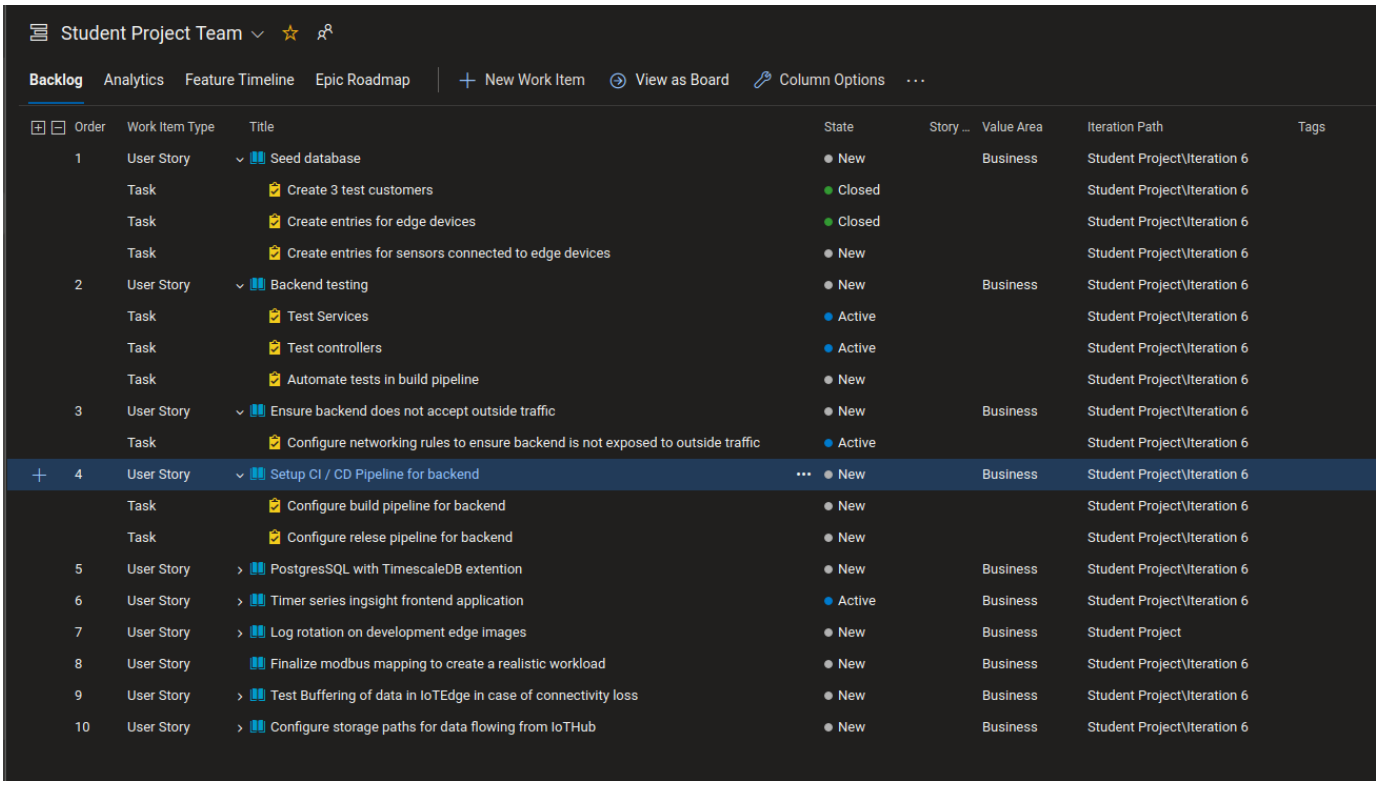


Figur 3.2: Burndown fra sprint 2, her ser vi at det i starten ble utført korrekt mengde arbeid, men at vi i siste halvdel av sprinten havnet på etterskudd.

Et annet element vi hentet fra SCRUM metodikk var daglige standup møter kl 1130. I disse møtene diskuterte vi hva hadde gjort siden forrige møte, hva vi skulle gjøre frem til neste møte, og hvilke utfordringer vi hadde støtt på. På denne måten var det enkelt å tydeliggjøre hva hvert enkelt gruppemedlem arbeidet med, og hvilke utfordringer dette gruppemedlemmet trengte hjelp til å løse.

3.2 Prosjektstyringsverktøy - Azure DevOps

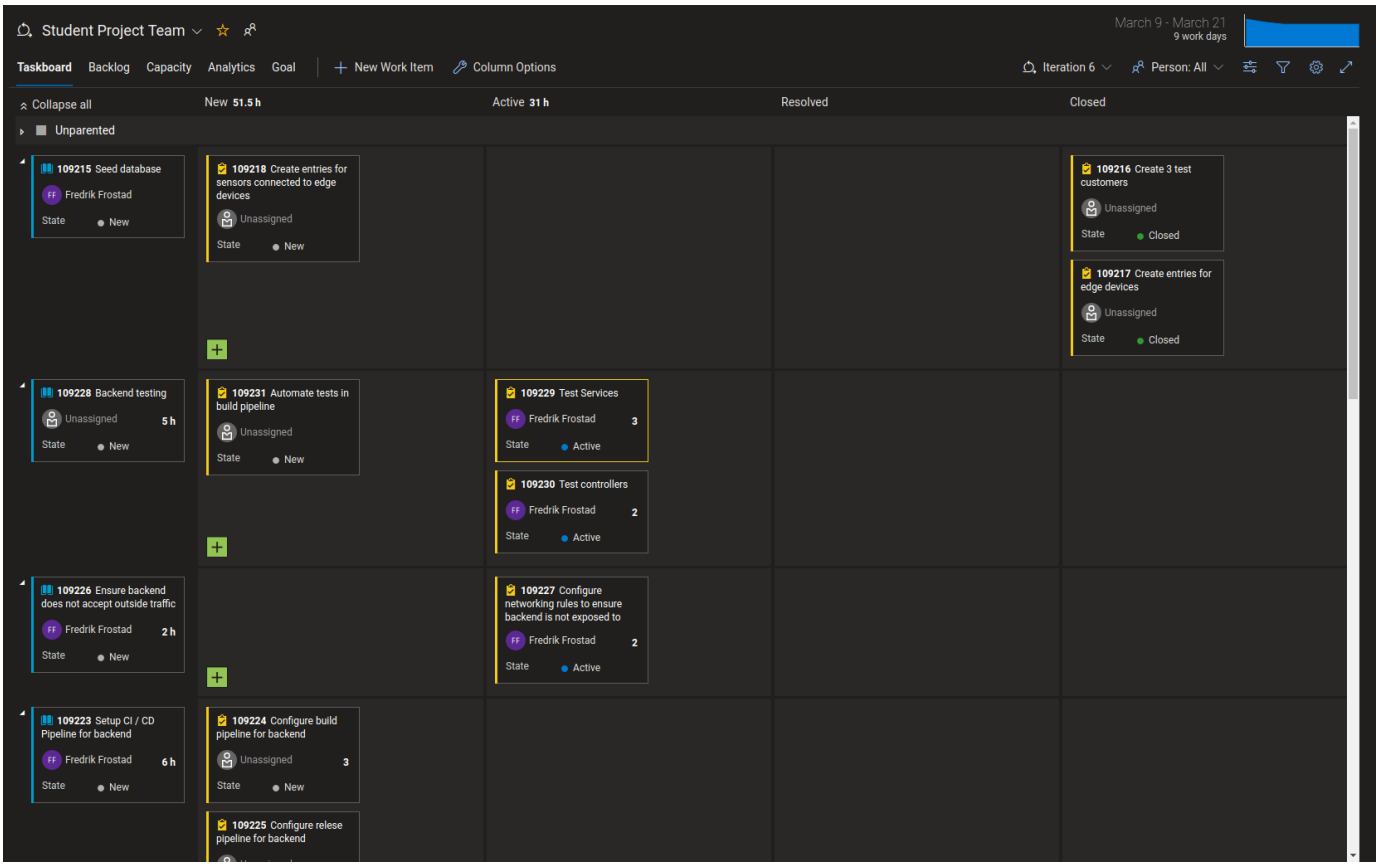
Vi har under mesteparten av prosjektet benyttet oss av Azure Devops som prosjektstyringsverktøy. Dette er et kraftig nettbasert verktøy, som integrerer Kanban brett, backlog, versjonskontroll, sprint planlegging og oppfølging i ett og samme miljø. Kjernen i vår arbeidsflyt under oppgaven var backloggen og kanban brettet. Vi arbeidet for det meste i to ukers sprint sykluser, der hver sprint hadde sitt eget kanban brett.



Order	Work Item Type	Title	State	Story ...	Value Area	Iteration Path	Tags
1	User Story	Seed database	New		Business	Student Project\Iteration 6	
	Task	Create 3 test customers	Closed			Student Project\Iteration 6	
	Task	Create entries for edge devices	Closed			Student Project\Iteration 6	
	Task	Create entries for sensors connected to edge devices	New			Student Project\Iteration 6	
2	User Story	Backend testing	New		Business	Student Project\Iteration 6	
	Task	Test Services	Active			Student Project\Iteration 6	
	Task	Test controllers	Active			Student Project\Iteration 6	
	Task	Automate tests in build pipeline	New			Student Project\Iteration 6	
3	User Story	Ensure backend does not accept outside traffic	New		Business	Student Project\Iteration 6	
	Task	Configure networking rules to ensure backend is not exposed to outside traffic	Active			Student Project\Iteration 6	
4	User Story	Setup CI / CD Pipeline for backend	New		Business	Student Project\Iteration 6	
	Task	Configure build pipeline for backend	New			Student Project\Iteration 6	
	Task	Configure relese pipeline for backend	New			Student Project\Iteration 6	
5	User Story	PostgresSQL with TimescaleDB extention	New		Business	Student Project\Iteration 6	
6	User Story	Timer series ingsight frontend application	Active		Business	Student Project\Iteration 6	
7	User Story	Log rotation on development edge images	New		Business	Student Project	
8	User Story	Finalize modbus mapping to create a realistic workload	New		Business	Student Project\Iteration 6	
9	User Story	Test Buffering of data in IoTEdge in case of connectivity loss	New		Business	Student Project\Iteration 6	
10	User Story	Configure storage paths for data flowing from IoTHub	New		Business	Student Project\Iteration 6	

Figur 3.3: Eksempel på backlog fra prosjektperioden

Når vi skulle planlegge en ny sprint, tok vi utgangspunkt i den prioriterte backloggen, som består av overordnede brukerhistorier som igjen er brutt ned i mindre oppgaver. Alle oppgavene i backlogen er estimert med tanke på forventet tidsbruk, noe som forenkler arbeidet med planlegging av sprinter. Når en sprint skal planlegges begynner vi øverst i backloggen og plukker brukerhistorier som skal være med i sprinten. Vi har på forhånd estimert hvor mange arbeidstimer hvert gruppemedlem kan bidra med inn i sprinten, og unngår dermed å dra for mye, eller for lite arbeid inn i hver sprint.



Figur 3.4: Eksempel på kanbanbrett fra prosjektperioden

I kanban brettene som hører til hver sprint, er alle oppgavene som inngår i sprinten representert ved et kort som har et issue nummer, en tittel, en eier og et estimat over gjenværende arbeidstimer på oppgaven. I kanban brettet er det definert fire ulike kolonner som representerer stegene en oppgave går igjennom. Disse kolonnene er:

- **New:** Her ligger alle oppgaver som ikke er påbegynt. Listen er prioritert, og følger samme rekkefølge som den prioriterte backloggen.
- **Active:** Her plasseres de oppgavene som utføres for øyeblikket. Denne kolonnen gir en god oversikt over hvem som jobber med hva, og hvor lang tid vedkommende har igjen før gjeldende oppgave er fullført.
- **Resolved:** Når et gruppemedlem sier seg ferdig med en oppgave legges den her. Deretter går den som har utført oppgaven igjennom den med et annet gruppemedlem for å forsikre seg om at oppgaven er korrekt og tilfredsstillende utført.
- **Closed:** når en arbeidsoppgave er fulført og sjekket av ett av de andre gruppemedlemmene, kan denne oppgaven lukkes og oppgaven flyttes til Closed kolonnen.

3.3 Versjonshåndtering

Versjonshåndtering er en svært viktig del av utviklingsprosessen og er en av hjørnesteinene i moderne programvareutvikling[8, p. 318-343]. Et versjonshåndteringssystem holder oversikt over uli-

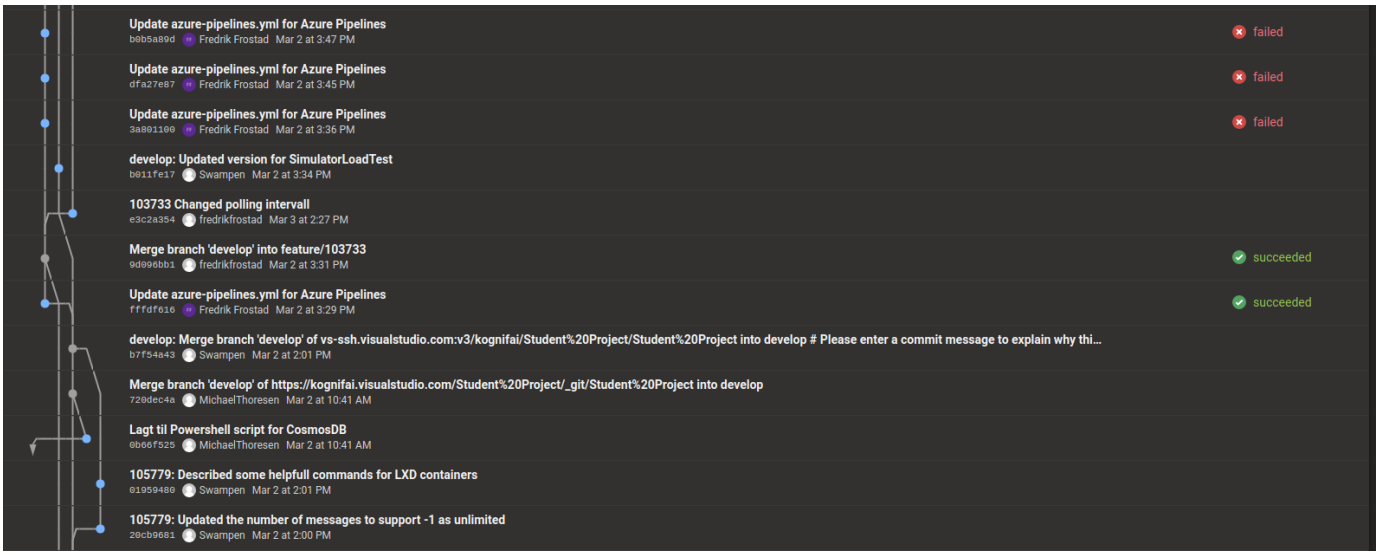
ke versjoner og konfigurasjoner av et system, og tillater flere utviklere å jobbe på samme kodebase uten å ødelegge for hverandre. Moderne versjonshåndteringssystemer tilbyr også funksjonalitet for å løse konflikter når to eller flere utviklere har jobbet på samme kode, og deres endringer skal sammenføyes til en enhetlig kodebase.

Det finnes flere ulike verktøy for versjonshåndtering, men det desidert mest brukte alternativet i dag er Git[9]. Vi har i dette prosjektet valgt å bruke Git som versjonshåndteringsverktøy, og vi har hostet prosjektets Git-repository i Azure Repos, fordi dette gir oss direkte integrasjon mot blant annet Boards og Pipelines i Azure Devops.

3.3.1 Git og Azure Repos

Git er et open-source, distribuert versjonskontroll system. Dette skiller seg fra et sentralisert system, ved at hver utvikler har sin egen lokale kopi av kodebasen som han eller hun jobber på. Dermed kan flere utviklere jobbe på samme kode parallelt. Når endringer skal integreres mot "master"versjonen av koden , gjør utvikler en forespørsel om dette, og de lokale endringene flettes inn i master koden på server. Dersom det er konflikter mellom utviklers kode, og master kode, må disse løses opp i manuelt. For å unngå at slike konflikter oppstår og blir liggende uløste i kodebasen på server, krever Git at utvikler alltid har hentet ned siste versjon av kodebase fra server til sin lokale kopi, før endringer flettes inn i kodebasen på server.

Azure Repos er en plattform som tilbyr lagring og håndtering av Git-repositories i skyen. Dette gjør at alle utviklere som jobber på prosjektet har fjerntilgang til kodebasen, og kan dermed holde sin lokale kopi oppdatert ved å jevnlig flette inn endringer fra repositoriet som ligger lagret i Azure Repos. Azure Repos tilbyr også utvidet funksjonalitet, som oversikt over endringer som er flettet inn i master kode, status på byggprosess som følge av endringer i kodebase konfigurert for Continuous Integration/Delivery via Azure Pipelines, og integrasjon mot Azure Boards (se Figur 3.5). I arbeidet med oppgaven har vi funnet at integrasjonen mot Azure Boards i forening med Git-Flow (se 3.3.2) arbeidsflyt har vært et svært godt verktøy for å holde styr på kodeendringer, og for å kunne gå tilbake å indentifisere og rette bugs i koden.



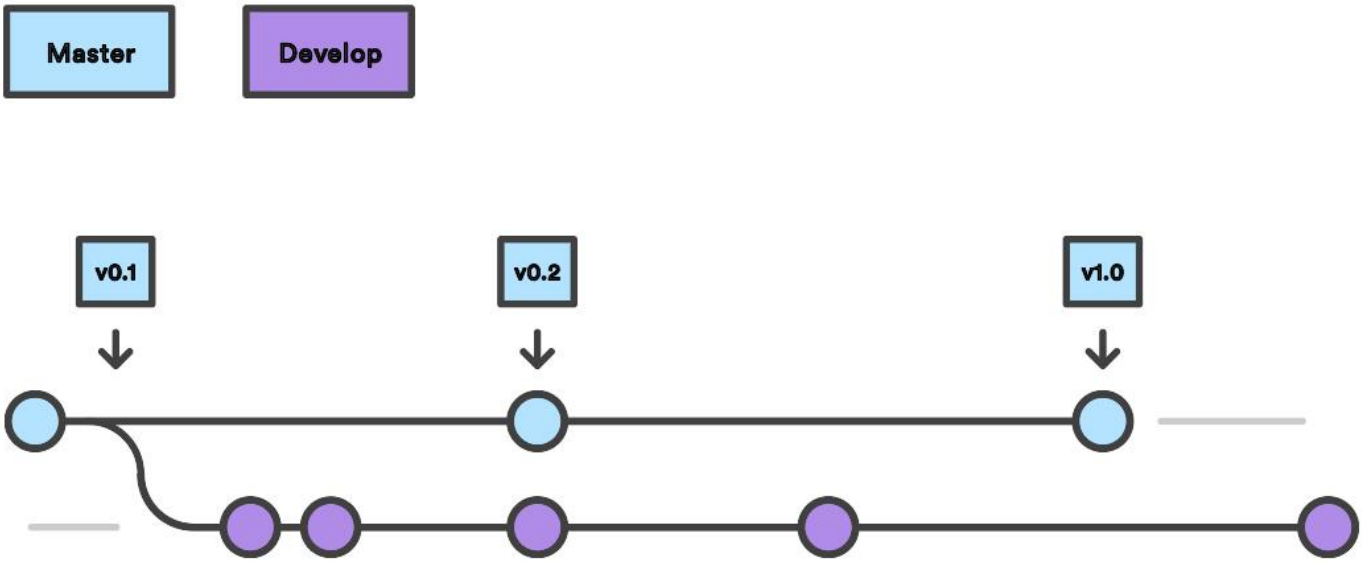
Figur 3.5: Utdrag av commit-historie og bygg-status fra prosjektet

3.3.2 GitFlow arbeidsflyt

GitFlow[10] er en metode for arbeidsflyt i Git som definerer en streng model for forgrening i et Git prosjekt. GitFlow er først og fremst tiltenkt prosjekter som har en jevnlig utgivelsessyklus[11], men det utelukker ikke bruk av denne typen arbeidsflyt i prosjekter hvor dette ikke er tilfellet. Vi har valgt å bruke GitFlow, fordi vi mener denne metoden gir oss svært god oversikt over endringer i kodebase, og gjør det enkelt å knytte grener i Git repositoriet opp mot oppgaver definert i Azure Boards.

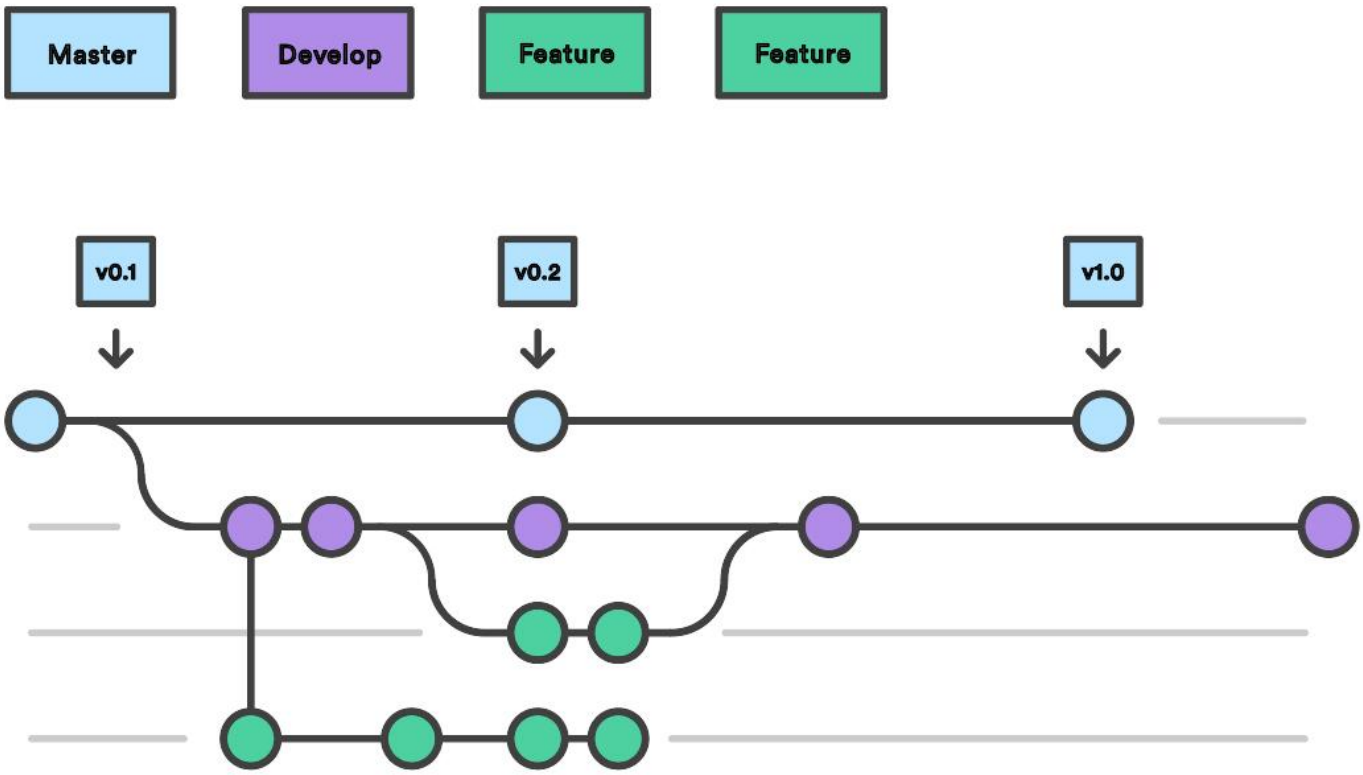
Hvordan GitFlow fungerer: Vi vil i dette avsnittet først gå igjennom den overordnede modellen for hvordan GitFlow fungerer, deretter vil vi gi et steg-for-steg eksempel av hvordan vi har brukt GitFlow i vårt prosjekt.

I all hovedsak er GitFlow en arbeidsflyt som definerer hvordan man strukturerer forgreninger i et Git prosjekt. I stedet for å kun ha en master gren som alle kodeendringer fortløpende flettes inn i, opererer man i GitFlow med to hovedgrener for å holde styr på versjonshistorien i et prosjekt. Disse grenene kalles **master** og **develop**. Master grenen inneholder offisielle versjoner av kodebasen, mens develop grenen brukes som en integrasjonsgren for ny funksjonalitet (heretter kalt features).



Figur 3.6: Develop og master gren

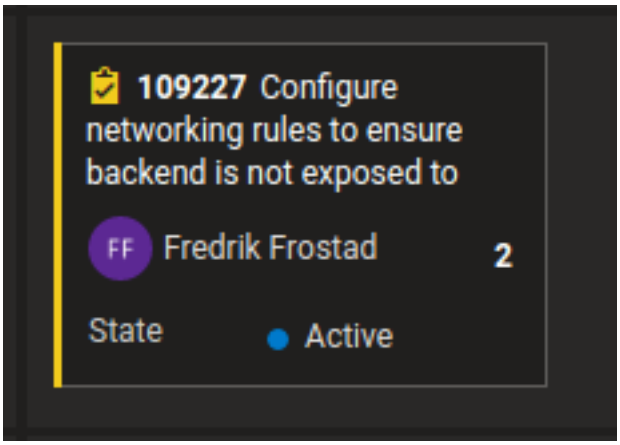
Hver oppgave fra Kanban brettet i Azure Boards representerer en feature i GitFlow modellen. Hver nye feature skal utvikles i en egen gren. Denne grenen forgrenes ut ifra develop grenen og ligger gjerne lokalt hos utvikler, men kan også pushes til det sentrale repositoryet i Azure Repos, slik at man kan samarbeide med andre utviklere, eller for å ha en redundant kopi av arbeidet. Når en feature gren er komplett og arbeidet som er utført er godkjent, kan feature grenen flettes inn i develop grenen. Her er det viktig å huske at et sentralt element i GitFlow tankegangen er at en feature gren aldri skal flettes direkte inn i master grenen.



Figur 3.7: Feature grener forgrenet fra develop gren

Eksempel på arbeidsflyt fra vårt prosjekt:

- Først velger vi en oppgave fra kanban brettet i Azure Boards. Denne oppgaven har et unikt nummer som vi benytter oss av for å knytte oppgaven sammen med tilhørende feature gren.



Figur 3.8: Oppgave fra Kanban brett

- Deretter sørger vi for at vår lokale kopi av develop grenen er oppdatert med de siste endringer fra Azure Repos, og lager en ny feature gren merket med id fra oppgaven.

```
git checkout develop
git pull
git checkout -b feature/109227
```

Figur 3.9: Kommandoer for å opprette ny feature gren

- Nå kan vi implementere ny funksjonalitet i vår nye feature gren. For å letter kunne spore kodeendringer prefikser vi alle commit-beskjeder med id nummeret til oppgaven vi jobber med på denne måten:

```
git commit -m "109228 Implement stricter CORS-policy"
```

Figur 3.10: Kommandoer for å committe kodeendring med beskjed

- Når arbeidet med oppgaven til slutt er ferdig, sett over og godkjent av et annet gruppemedlem, kan vi flette feature grenen inn i prosjektets develop gren. Tilslutt sletter vi den fullførte feature grenen, og gjentar hele prosessen for neste oppgave.

```
# Sjekk ut develop gren og sjekk at den er oppdatert
git checkout develop
git pull
# Flett inn endringer fra feauture gren
git merge feature/109227
# Push endringer til Azure Repos
git push
# Slett feature gren lokalt og i Azure Repos
git branch -d feature/109227
git push origin --delete feature/109227
```

Figur 3.11: Kommandoer for å flette inn endringer og rydde opp

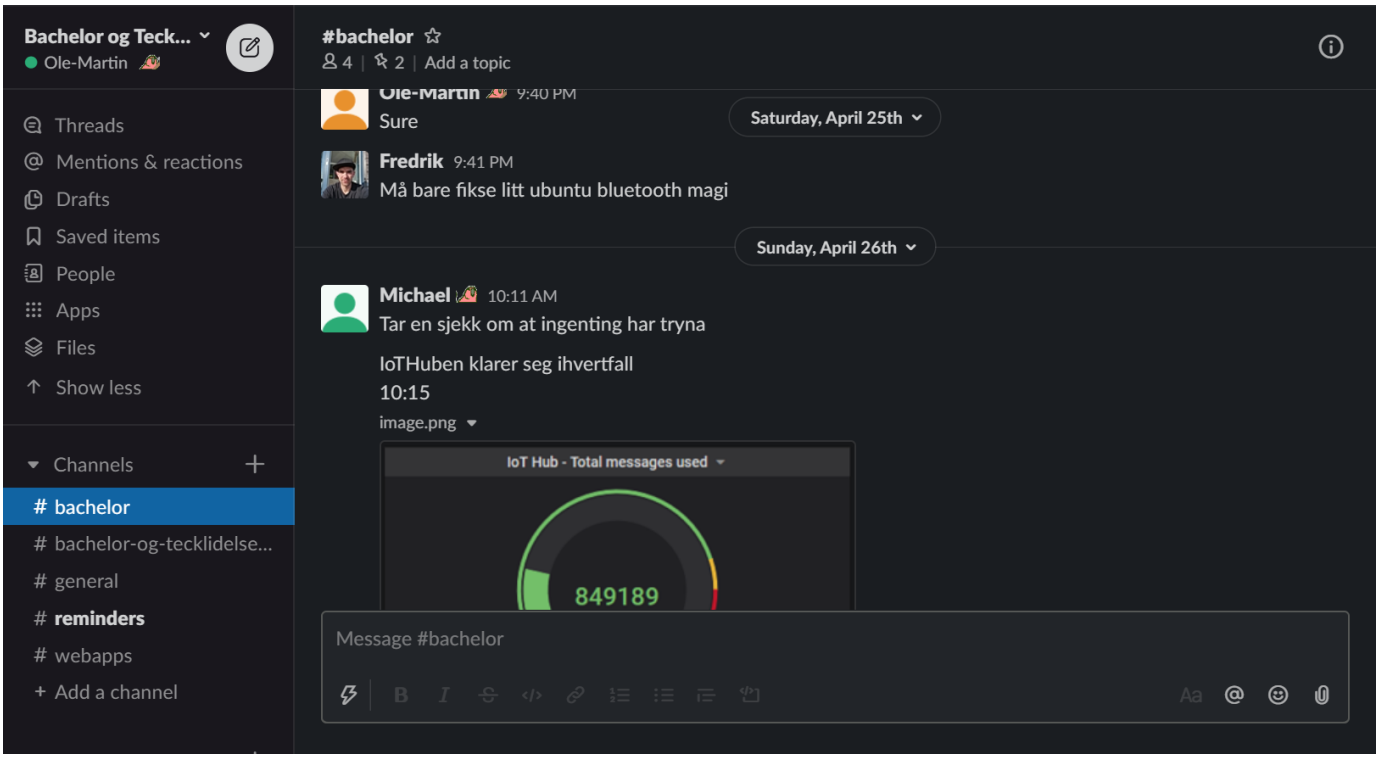
3.4 Kommunikasjonsverktøy

Som følge av smittevernstiltak satt i verk for å begrense omfanget av koronapandemien, har vi under utførelsen av prosjektet benyttet oss av en rekke digitale kommunikasjonsverktøy. Det har vært nødvendig å benytte flere forskjellige plattformer for å holde kontakten med alle involverte

aktører gjennom prosjektets gang. Her presentere vi de viktigste kommunikasjonsverktøyene vi har benyttet oss av.

3.4.1 Slack

[12] er en samarbeidsplatform hvor kjernefunksjonaliteten er en skybasert chattetjeneste. Slack er særdeles godt egnet for samarbeid i grupper og ble opprinnelig utviklet som et samarbeidsverktøy for bruk i programvareutvikling [13]. Plattformen har støtte for fildeling, markup formatering av tekst i chat, automatiserte remindere samt lyd og videosamtaler.



Figur 3.12: Skjermbilde fra gruppens slack kanal

Der slack før de koronarelaterte tiltakene satte inn, var et godt hjelpemiddel for å forenkle kommunikasjon internt i gruppen, ble det etter at tiltakene satte inn et uvurderlig verktøy i den resterende prosjektgjennomførelsen. Slack har tillatt oss å opprettholde god og effektiv kommunikasjon internt i gruppen, til tross for at vi i halve prosjektperioden har arbeidet på forskjellige lokasjoner. Vi har brukt slack for å koordinere arbeidet, feilsøke problemer i fellesskap, og som en kilde til sosial omgang. Vi er svært godt fornøyd med funksjonaliteten slack tilbyr, og har opplevd det som et viktig og produktivt verktøy i utførelsen av prosjektet.

3.4.2 Teams

er samarbeidsplatform fra Microsoft som tilbyr mye av den samme funksjonaliteten som Slack. Teams er Kongsberg Digital sin foretrukne platform for internkommunikasjon, og vi har benyttet oss av denne for å opprettholde kontakten med oppdragsgiver i tiden etter at de valgte å stenge sine kontorer som tiltak for å hindre spredning av koronavirus.

3.4.3 Google Meet

er en videokommunikasjonsplatform utviklet av Google. Google meet er en rebranding og videreutvikling av det som tidligere het google hangouts. Google Meet tilbyr videomøter for opp til 100 deltagere, mulighet for å bli med i møter både fra browser og egen app, og deling av skjerm. Vi har brukt Google meet til daglige standups, samarbeidssesjoner internt i gruppen og for å holde status og demonstrasjonsmøter med oppdragsgiver. Vi har i møter med oppdragsgiver hatt mer enn ti deltakere i samme møte, og dette har fungert svært tilfredsstillende.

3.5 Metode og testdata for gjennomføring av ytelsestester

I dette delkapittelet vil vi beskrive hvordan vi har planlagt å utføre tester av systemet med syntetisk last, med hensikt å samle inn data om systemets ytelse og kostnad. Under utførelsen av disse testene vil vi se på sammenhengen mellom ytelse og kostnad for de enkelte komponenter i systemet.

Under utføringen av testene vil vi i hovedsak evaluere følgende:

- Hva er kostnaden og ytelsen ved bruk av Cosmos DB vs PostgreSQL med Timescale
- Hva er kostnaden og ytelsen ved bruk av Time Series Insight
- Hva er kostnaden og ytelsen ved bruk av IoT Hub.
- Hvor mange IoT Edge enheter kan kjøres på en enkelt VM i Azure, og hvilke ressurser må denne må ha tilgjengelig.

Etter samtaler med oppdragsgiver ble vi enige om at ytelsesmålinger for modulene som kjører på IoT Edge, samt tester mot selve IoT Edge kjøretiden ikke ville være relevant for oppdraget. Vi har derfor valgt å kjøre alle testene med pre-konfigurerte instanser av IoT Edge, som kjører en egen temperatur sensor modul som genererer syntetisk telemetridata.

3.5.1 Metode

I dette delkapittelet vil vi gå gjennom rammene for de ulike testfasene vi planlegger å gjennomføre. For resultater av disse testfasene, se kapittel 5 - Resultater, og 6 - Drøfting og analyse.

Vi vil utføre testene i tre forskjellige faser der vi på forhånd har definert hvor stor belasting vi vil påføre systemet. For hver testfase vil vi øke belastingen på systemet for å undersøke om vi kan finner trender som viser hvordan ytelse og kostnad skalerer.

3.5.1.1 Felles kriterier for alle testfaser

- Hver IoT Edge enhet vil hente data fra 200 simulerte sensorer
- Hver simulerte sensor vil ha en Id og og produsere en randomisert temperaturverdi
- Hver IoT Edge vil lese sensorverdier med 1 Hz frekvens og samle disse i et JSON dokument som sendes direkte til IoT Hub. Det vil altså sendes ett json dokument inneholdende 200 sensorverdier til skyen pr sekund fra hver enkelt IoT Edge. Se testdata for nærmere beskrivelse av dataene.

3.5.1.2 Kriterier for individuelle testfaser

- Test 1: Testen gjennomføres med 10 IoT Edge enheter som kjører i parallell.
- Test 2: Testen gjennomføres med 20 IoT Edge enheter som kjører i parallell
- Test 3: Testen gjennomføres med 50 IoT Edge enheter som kjører i parallell

3.5.1.3 Igangsetting og stansing av tester

Vi så tidlig at det ville være nødvendig og automatisere prosessen for å starte og stoppe test-kjøringer. Dette har vi løst ved å lage en rekke bash script som i samarbeid med Azure device provisioning service starter prekonfigurerte lxc-containere som inneholder en linux avbildning med en ferdig installert IoT Edge instans. Scriptene kan sees i vedlegget:

- Script som verifiserer variabler og iverksetter sekvensiell oppstart av IoT Edge enheter
- Script som genererer symmetriske nøkler og konfigurerer IoT Edge instans
- Script som sletter alle forgjengelige ressurser brukt i testkjøringen

3.5.1.4 Inhenting av data fra test kjøringer

Hver testsyklus kjøres over 24 timer. Vi henter data om testkjøringen fra følgende kilder:

- Azure Monitoring - Ytelsesmetrikker
- Azure Cost Analysis - Kostnadsmetrikker

3.5.2 Testdata

I dette avsnittet vil vi beskrive et standardisert telemetri-meldingsformat som vi vil bruke i alle de tre testfasene.

Telemetrimeldingene som sendes fra IoT Edge enhetene for inntak, prossessering og lagring i skyen vil ha form som et JSON dokument. Dette dokumentet inneholder samt ID og måleverdi for alle de simulerte sensorene som tilhører nevnte IoT Edge enhet og et tidsstempel for avlesnings-tidspunkt. Se Listing 3.1 for utdrag fra en telemetrimelding.

```
1  {
2    "MessageNumber": 522,
3    "Machines": [
4      {
5        "SensorId": 1,
6        "Temperature": 103.93095805050393
7      },
8      {
9        "SensorId": 2,
10       "Temperature": 103.91676739017338
11     },
12     {
```

```
13      .
14      .
15      .
16      .
17      },
18      {
19        "SensorId": 199,
20        "Temperature": 104.07126347328138
21      },
22      {
23        "SensorId": 200,
24        "Temperature": 104.10815670555846
25      }
26    ],
27    "TimeCreated": "2020-05-12T08:30:28.095463Z"
28  }
```

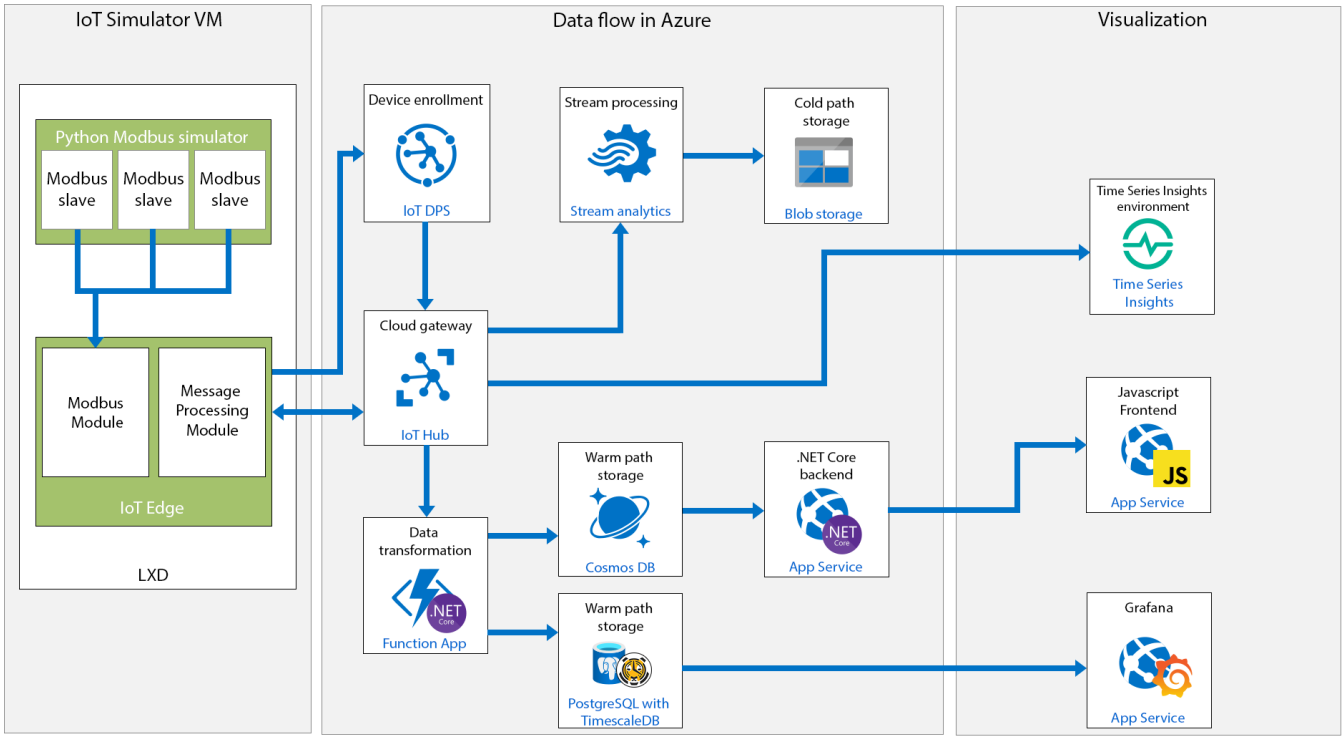
Listing 3.1: Std telemetrimelding brukt i alle tester. Sensor 3 til 198 er ikke vist her.

Hver telemetrimelding har en størrelse på **16.37 KB**. Det vil si at hver IoT Edge enhet genererer **0.98 MB** data pr. minutt.

Kapittel 4

Systembeskrivelse - Proof-of-concept system

For at leseren skal ha nok forståelse om det overordnede systemet, så har vi valgt å ha en kort-fattet beskrivelse systemets enkeltkomponenter og flyt i dette kapittelet. En mer detaljert teknisk beskrivelse kommer i kapittel 9 - Systemet. Under er det vist vår implementasjon av ende-til-ende systemet og dens komponenter, altså proof-of-concept systemet.



Figur 4.1: Systemarkitektur

4.1 IoT Simulator - virtuell maskin

For at dette skal være et komplett ende-til-ende system, så må systemet få data inn. Alt dette gjøres på en virtuell maskin. Den simulerer Modbus data, i tillegg til å sende dette videre inn i systemet. Dette gjøres ved hjelp av LXD containere. På denne måten får vi også simulert flere IoT Edger uten problemer.

4.1.1 Azure IoT Edge

Azure IoT Edge(også referert til som IoT Edge) er et container basert program fra Microsoft som blant annet gjør det mulig flytte prosessering og analyse av data fra skyen til lokasjonen der dataene genereres. IoT Edgen består av tre komponenter

- **IoT Edge Moduler** er containere som kan kjøre Azure tjenester, tredjeparts tjenester eller egen kode. En IoT Edge kan ha en til 20 moduler som enten kjører individuelt eller sammen.
- **IoT Edge runtime** administrerer modulene som blir distribuert til hver IoT Edge.
- Et **cloudbasert grensesnitt** som gjør det mulig å monitorere og administrere IoT Edge enheter [14].

4.1.1.1 Modbus modul

Dette er en modul som er laget av Azure for å kunne lese Modbus data fra en Modbus slave over TCP. Denne har som primærjobb å lese data fra Modbus slaver og publisere dette videre som meldinger, men kan også skrive til Modbus register. Vi har konfigurert denne til å rute meldinger videre til Message Processing modulen.

4.1.1.2 Message Processing modul

Denne modulen tar meldingene som kommer fra Modbus modulen, og legger til informasjon som Edge ID og kunde ID til meldingen før den blir sendt til Azure IoT Huben. Dette kunne vært gjort direkte i Modbus modulen, men vi har valgt å gjøre dette i en egen modul for å vise hvordan meldinger kan rutes mellom moduler.

4.1.2 Python Modbus simulator

Python Modbus simulatoren er et Python-script som vi har laget for å kunne simulere en Modbus slave. For at det skal være data som er gjenkjennelig, skriver den kjente som verdier i form av en sinuskurve eller en fast verdi inn i registrene. Dette skriptet baserer seg på Modbus som er et open source bibliotek for modbus simulering.

4.2 Dataflyt i Azure

I denne seksjonen vil det bli forklart hvordan flyten i Azure vil gå på en forståelig måte, slik at leseren kan enkelt forstå hva som skjer i hvilken rekkefølge.

4.2.1 Azure IoT Hub

Azure IoT Hub er en gateway inn og ut av Azure skyen for IoT Edge enhetene våre. Den håndterer meldinger som kommer fra IoT Edgene, sender konfigurasjonsparametere til IoT Edgene og administrerer disse med stor fleksibilitet[15].

4.2.2 Azure IoT Hub DPS

Azure IoT Hub DPS (Device Provisioning Services) er brukt for å automatisk registrere nye IoT Edger[16]. IoT Hub DPS sjekker en symmetrisk nøkkel som er satt i konfigurasjonen på hver enkelt IoT Edge, og autentiserer IoT Edgen mot IoT Huben hvis den stemmer. I vår implementasjon er dette kun brukt for å kjøre belastningstester av systemet med større mengder nye IoT Edge enheter.

4.2.3 Function App

Azure Functions er en modul i Azure som tillater oss å kjøre små kodesnutter som respons til meldinger som kommer inn til IoT Huben. Den koden vi har laget tar inn disse meldingene hver for seg og transformerer det til et flatt format som er mer rad- og spørrevennlig. Funksjonene markerer dataen med navnet til IoT Edgen som meldingen kommer fra, og deretter settes den transformerte dataen inn som rader i en PostgreSQL database, og som dokumenter i Cosmos DB.

4.2.4 Azure Cosmos DB

Azure Cosmos DB er en av database løsningene vi har tatt i bruk i varm stien av systemet vårt. Cosmos DB er Microsoft sin egenutviklede dokumentdatabase som er designet for å være lett å bruke, og for å kunne raskt hente av store mengder data[17]. I vårt system brukes Cosmos DB som den primære varmlagringsløsningen av dataene som blir transformert i Azure Function.

4.2.5 PostgreSQL med TimescaleDB

PostgreSQL (også kjent som Postgres) er en open source, objektrelasjonell database, mens TimescalDB er en open source, tidsserie SQL database, pakket som en utvidelse av Postgres. Til sammen skaper dette en database optimalisert for rask innsetting, og komplekse spørringer på tidsseriedata[18]. Vi har brukt dette som et alternativ til Cosmos DB for varmlagring av data i vårt system.

4.2.6 Azure App Service

Azure App Service er en HTTP-basert Platform-as-a-Service i Azure som kan kjøre applikasjoner som nettsider og API-er. Med App Sevice er sikkerhet, last-balansering og auto skalering håndtert i Azure slik at vi kan fokusere på utvikling fremfor underliggende arkitektur[19]. I vårt system så er det tre tjenester som kjører på App Service: backend API-et, frontend nettsiden og Grafana.

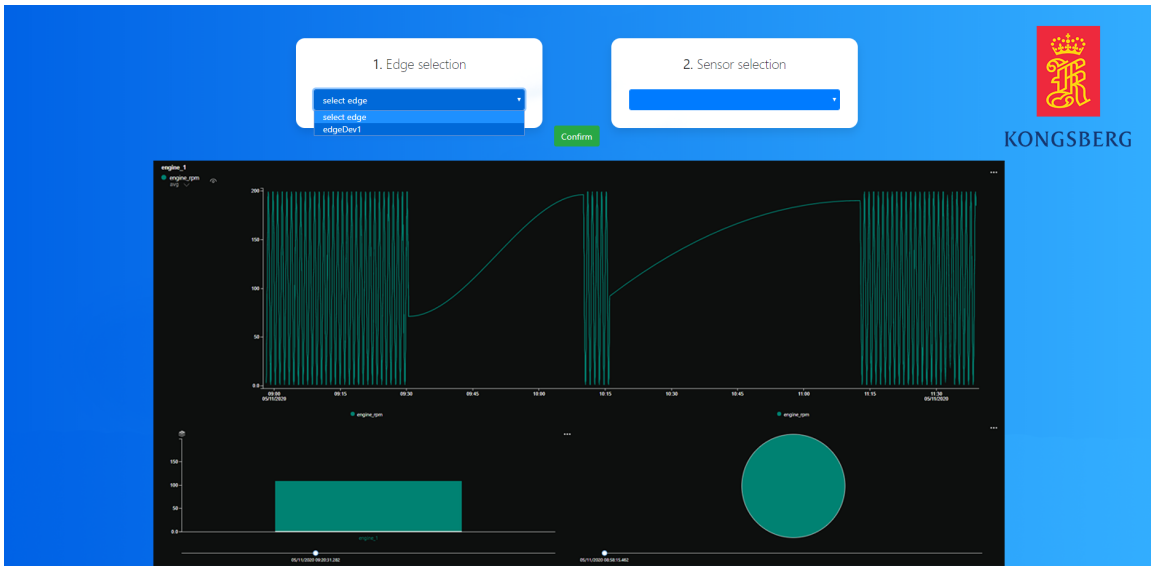
4.2.6.1 Backend

For at vi enkelt skal kunne se på dataene som er blitt lagret i Cosmos DB, har vi laget et backend API som henter data som tilsvarer parametrene som blir oppgitt. API-et er skrevet i C# og krever også autentisering for å hente ut data som tilhører den innloggede brukeren.

4.2.6.2 Frontend

Frontenden er en minimal proof-of-concept implementasjon for å kunne se på data som kan hentes fra backend API-et. Det er også implementert innlogging mot backenden for å kunne hente ut

riktige data i forhold til hvilken bruker som er innlogget. Bilde under (fig 4.2) viser en innlogget bruker og IoT Edger som er knyttet til brukeren.



Figur 4.2: Skjermdump av frontend

4.2.6.3 Grafana

Grafana er en open source analyse og visualiserings løsning. Med Grafana kan en gjøre spørringer på data, visualisere det og sette opp alarmer. Den tilbyr enkel integrasjon med tidsserie databaser, noe som gjorde at vi valgte å bruke Grafana i vårt system mot Postgres databasen, for å visualisere data. Med Grafana har vi også satt opp monitorering av komponenter i Azure. I firgur 4.3 vises det sensordata for den gitte kunden og edge IDen.

Figur 4.3: Skjermdump av Grafana

4.2.7 Azure Stream Analytics

Azure Stream Analytics er et verktøy som kan analysere og prosessere sanntidsdata i store mengder[20]. Vi har brukt dette for å filtrere bort stresstest data mot den faktiske dataen vi sendte inn til IoT Huben, og sender dette videre til Azure Blob storage.

4.2.8 Azure Blob storage

Azure Blob storage er Microsoft sin løsning for lagring av objekter. Den er optimalisert for å lagre store mengder ustrukturerte data i form av filer[21]. Her blir meldinger som kommer igjennom Stream Analytics lagret som store JSON filer.

4.2.9 Azure Time Series Insights

Azure Time Series Insight er en komponent fra Microsoft som gjør lagring, visualisering og spørring på store mengder tidsseriedata [22]. Dette er et alternativ vi har sett på som et komplett system i seg selv, uten behov for å skrive egen kode. Vi har sendt meldinger direkte fra IoT Huben inn i Time Series Insights.

Kapittel 5

Resultater

I dette kapittelet skal vi gjennomgå resultatene som har blitt produsert i løpet av prosjektet. Som beskrevet i kapittel 3.5 Metode og testdata for gjennomføring av ytelsestester, har vi gjennomført tester i tre faser hvor vi har testet systemet med 10, 20 og 50 Azure IoT Edge enheter. Azure Functions er en essensiell komponent i dette systemet, men viste seg tidlig å ikke ha noen begrensninger med tanke på skalering til vårt bruk. Derfor har vi valgt å se bort ifra denne i resultatene. Det forventes at leser har lest kapittel 4 Systembeskrivelse og fått en grunnleggende forståelse av systemet. Her er noen tilleggsdetaljer som kan være nyttige når du skal lese dette kapittelet:

Azure Cosmos DB:

- Request Units (RU) – Er enheten Azure Cosmos DB bruker til å måle gjennomstrømningen inn til sine containere.

Azure Time Series Insights:

- Eventer - Time Series Insights(TSI) baseres rundt det som kalles eventer.
- Nivåer - Time Series Insights deles inn i to nivåer:

Time Series Insights nivåer	Antall eventer pr. enhet	Maksimalt antall eventer
Standard Tier 1	1 000 000	10 000 000
Standard Tier 2	10 000 000	100 000 000

Tabell 5.1: Time Series Insights nivåer med kapasitet

Azure IoT Hub:

- Nivåer – IoT Huben skaleres med hjelp av nivåer. Vi tar i bruk Standard Tier for denne oppgaven og Standard kan deles inn i tre nivåer:

Time Series Insights nivåer	Antall eventer pr. dag
Standard Tier 1	400 000
Standard Tier 2	6 000 000
Standard Tier 3	300 000 000

Tabell 5.2: Time Series Insights nivåer med kapasitet

- Partisjoner – er direkte knyttet til antall instanser som kan lese data samtidig.

Azure PostgreSQL:

- Tilkoblinger – PostgreSQL instansen vi bruker har en restriksjon på 100 samtidig tilkoblinger på en gang
- vCore – Antallet virtuelle CPU kjerner PostgreSQL instansen har blitt tildelt.

5.1 Testfase 1

I første testfase hadde vi en belastning på 10 IoT Edger, hvor hovedfokuset var å teste hvordan systemet oppførte seg under belastning. Delene av systemet som skulle testes var Cosmos DB, PostgreSQL og Time Series Insight (TSI) som beskrevet i testkriteriene. Disse er alle varmlagringsløsninger, og ble testet med samme last som går igjennom IoT Huben. I denne testfasen endte vi opp med 2 tester, hvor den første ble gjennomført 26. april og den andre 29. april.

Test 1.1 Tabell 5.3 viser testoppsettet for den første testen. Den ble kjørt med 1 virtuell maskin som simulerer 10 IoT Edger. Videre er Time Series Insights konfigurert (TSI) til Standard Tier 1(S1) med 3 enheter. Dette gir oss mulighet for å ta inn 3 000 000 eventer per dag. Cosmos DB var satt til automatisk skalering med 4 000 request units per sekund som øvre grense, og PostgreSQL er satt til 2 vCore med automatisk skalering på lagringskapasitet.

Cosmos DB	PostgreSQL	Time Series Insights (TSI)	IoT Hub	Virtuell maskin
4 000 RU/s	Basic tier 2 vCore Lagring: autoskalering	Standard tier 1 (S1) 3 enheter 3 000 000 eventer	Standard tier 2 (S2) 4 partisjoner	1 stk (VM størrelse B2ms) 2 vCPUs, 8GB RAM, 100GB SSD

Tabell 5.3: Testoppsett 26. april

Test 1.2 Den andre testen i denne fasen ble gjennomført 29. april, og kjørte med det samme oppsettet som i test 1.1, men Cosmos DB og Time Series Insights (TSI) er skalert opp (se tabell 5.4). Cosmos DB ble satt til 20 000 RU/s som øvre grense, og TSI ble oppgradert til en Standard tier 2 med 4 partisjoner, noe som vil resultere i 40 000 000 eventer. Dette er for å få nok gjennomstrømning av data.

Cosmos DB	PostgreSQL	Time Series Insights (TSI)	IoT Hub	Virtuell maskin
20 000 RU/s	Basic tier 2 vCore Lagring: autoskalering	Standard tier 2 (S2) 4 enheter 40 000 000 eventer	Standard tier 2 (S2) 4 partisjoner	1 stk (VM størrelse B2ms) 2 vCPUs, 8GB RAM, 100GB SSD

Tabell 5.4: Testoppsett 29. april

5.1.1 Resultater

Under testen utført 26. april, oppdaget vi at vi hadde konfigurert for lite gjennomstrømning til Cosmos DB og dette forårsaket at mange av dokumentene ble satt i kø opp til 6 timer etter endt test. Som vist på figur 5.1 kan vi se at testen stoppet kl. 02:00, men at Azure Function applikasjonen fortsatte å kjøre. Ut ifra Azure Monitor kunne vi se at det er Azure funksjonen prøver å sette inn dokumenter i Cosmos DB, men feiler fordi antall request units har overskredet 4 000 RU/s.

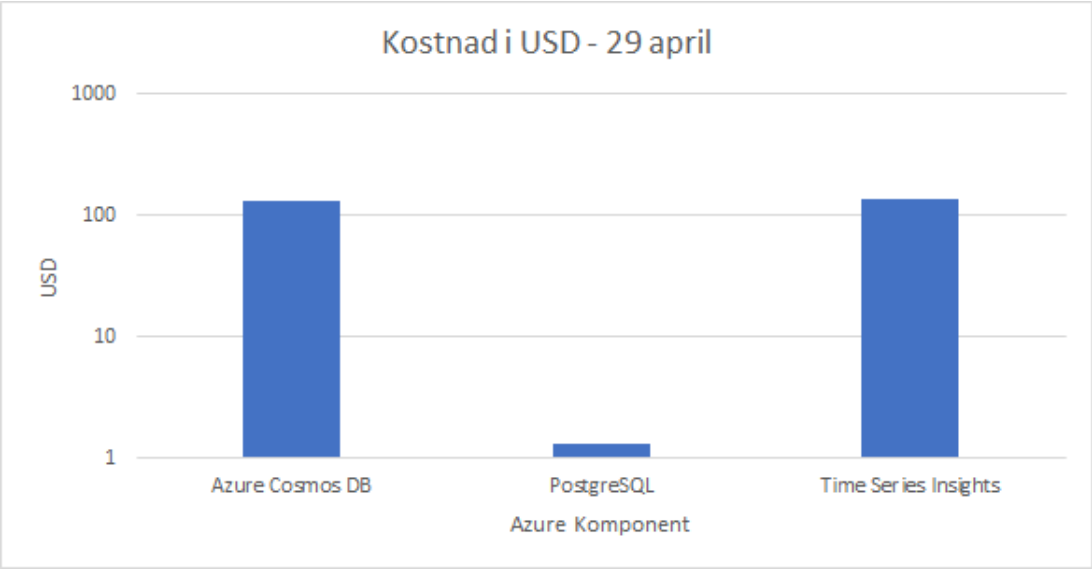
Figur 5.1: Viser tidsrommet 26. april kl.12:00 til 27. april kl. 12:00.
Funksjonen kjører fortsatt etter endt test kl. 02:00

Time Series Insights (TSI) var også konfigurert for lavt i første test. Dette gjorde at vi bestemte oss for å kjøre en ny test av 10 IoT Edger som ble gjort den 29. april som ga oss mer relevante testresultater både for kost og ytelse.

Den andre testen som ble kjørt resulterte i et bedre resultat. Maks antall request units i Cosmos DB ble ikke overskredet. Derimot så var det fortsatt en kø mot Cosmos DB, men mye kortere enn forrige test uten at vi helt kunne forklare hvorfor. Totalt sett et akseptabelt resultat for å se på priser.

Vi hadde litt andre problemer med TSI. Den var dimensjonert til 40 000 000 eventer på andre testen, noe som førte til prosesseringsforsinkelser i TSI. Ved senere kalkulasjoner, oppdaget vi at vi måtte ha konfigurert den til å ta imot mer enn 172 600 000 eventer. TSI kan i utgangspunktet bare konfigureres til å ta imot maks 100 000 000, men det er også en pay-as-you-go plan som ikke tilbyr noen priskalkulasjon på forhånd.

Kostnadene fra den andre testen (se fig 5.2), viste oss at Azure Cosmos DB kostet 131,61 amerikanske dollar(USD) over 24 timer med 20 000 RU/s. TSI kostet 135,47 USD. PostgreSQL kostet 1,31 USD og støttet ikke på noen problemer. IoT Hub S2 kostet 6,29 USD. Dette resultert i en totalkostnad på 274,68 USD



Figur 5.2: Kostnadsoversikt over de forskjellige varmlagringsløsningene vist logaritmisk.

5.1.2 Konklusjon

Etter denne testen besluttet vi å ikke teste Time Series Insights (TSI) videre, fordi TSI kun støtter opptil 100 millioner hendelser per dag som ikke dekker vårt behov med skalering. Vi var ikke villige til å prøve en pay-as-you-go løsning, fordi en konfigurasjon med 100 millioner eventer ville kostet 450 USD per dag. En estimert kostnad for en konfigurasjon for 172 600 000 eventer vil minst koste 774 USD.

Vi besluttet også at Cosmos DB drastisk må skaleres opp for å møte videre krav for testing av 20 IoT Edge enheter. Vi valgte derfor å og skalerer opp til 500 000 RU/s for å kunne håndtere datamengden for neste test. I tillegg ble det besluttet å skrive om funksjonen til å sette dokumenter i bulk operasjoner. Ved å gjøre dette ville forhåpentligvis problemet med køen løse seg. På grunn av ingen synlige problemer i PostgreSQL, gjorde vi ingen endringer med konfigurasjonen.

5.2 Testfase 2

I testfase to testet vi systemet med 20 IoT Edger fordelt på to virtuelle maskiner. Grunnen til at vi brukte to, er fordi i den siste testfasen vil vi bruke to virtuelle maskiner uansett. Prisen for disse lik som en med dobbel i størrelse. Som vi konkluderte med i forrige fase, har vi også valgt å stoppe videre testing av Time Series Insights i tillegg til å endre måten Azure funksjonen setter inn i Cosmos DB.

Test 2.1 Tabell 5.5 viser oppsettet for testen den 4. mai. Her er det satt opp to identiske virtuelle maskiner, samme som i forrige testfase, som kjører 10 IoT Edger hver. Cosmos DB er satt til automatisk skalering med maks grense på 500 000 request units. PostgreSQL og IoT Huben forblir uendret.

Cosmos DB	PostgreSQL	IoT Hub	Virtuell maskin
500,000 RU/s	Basic tier 2 vCore Lagring: autoskalering	Standard tier 2 (S2) 4 partisjoner	2 stk (VM størrelse B2ms) 2 vCPUs, 8GB RAM, 100GB SSD

Tabell 5.5: Testoppsett 4. mai

Test 2.2 I tabell 5.6 er testoppsettet for 6. mai presentert. IoT Huben er skalert opp til 32 partisjoner, mens PostgreSQL og de virtuelle maskinene som kjører 10 IoT Edger hver er uforandret. I tillegg er ikke Cosmos DB tatt med i denne testen.

PostgreSQL	IoT Hub	Virtuell maskin
Basic tier 2 vCore Lagring: autoskalering	Standard tier 2 (S2) 32 partisjoner	2 stk (VM størrelse B8ms) 2 vCPUs, 8GB RAM, 100GB SSD

Tabell 5.6: Testoppsett 6. mai

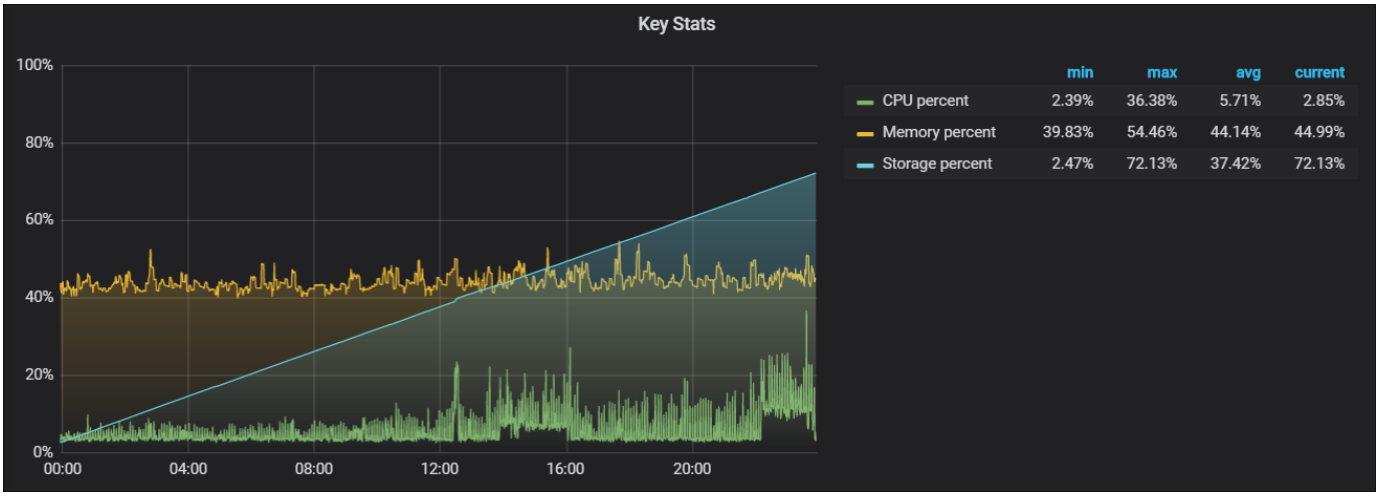
5.2.1 Resultater

Etter forrige fase endret vi metoden for innsetting i Cosmos DB til bulk operasjoner. Dette resulterte i et en mye mer effektiv innsetting av data, slik at køen ble redusert ytterligere fra forrige testfase. Som vi kan se fra figur 5.3 kjører funksjonene fortsatt etter at testet stoppet kl. 02:00 (blå stiplet linje). Vi ser at både PostgreSQL og Cosmos DB hadde bygget opp en kø i løpet av testen. Erfaring fra tidligere tester viser oss at PostgreSQL funksjonen stoppet rundt kl. 07:00 dagen etter, mens funksjonen til Cosmos DB fortsetter å kjøre. Kl. 09:00 ble vi oppringt av Kongsberg Digital, og ble bedt om å skru av Cosmos DB.

Figur 5.3: Viser tidsrommet 4. mai kl.12:00 til 5. mai kl. 12:00. Funksjonen kjører fortsatt etter endt test kl. 02:00

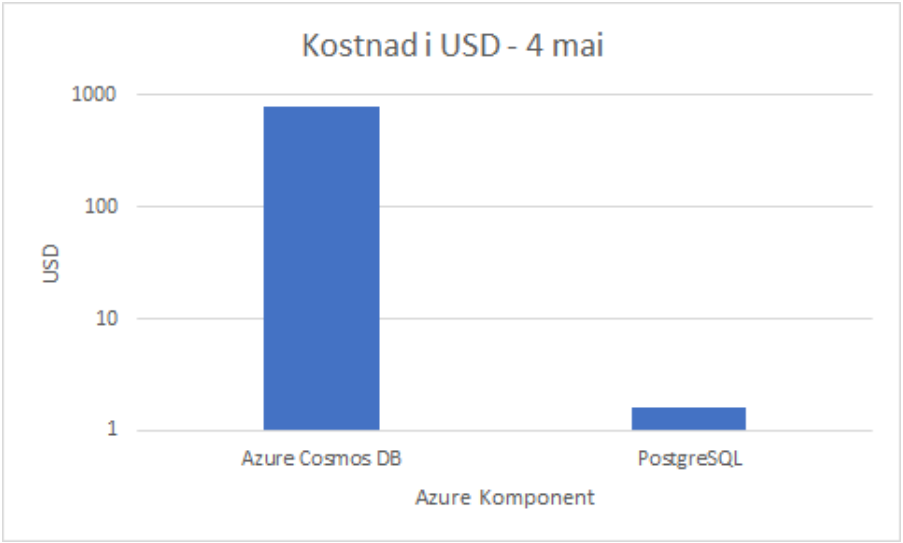
Et nytt problem som oppstod under denne testen, var at funksjonen mot PostgreSQL også hadde bygget opp en kø, noe den ikke gjorde i forrige testfase. Etter grundig lesing av dokumentasjonen til IoT Huben, viste det seg at vi hadde oversett innstillingen for antall partisjoner.

Antall partisjoner er direkte knyttet opp til antall instanser som kan lese data fra IoT Huben samtidig. Denne må settes når IoT Huben opprettes, og ligger under **Advanced settings** seksjonen. Når denne er satt til 4 partisjoner som standard, så betyr dette at det er kun fire instanser av Azure funksjonen som kan starte parallelt. Når vi da testet 6. mai med 32 partisjoner, hadde Azure funksjonen og PostgreSQL ingen problemer med å prosessere alle meldingene (se figur 5.4).



Figur 5.4: Monitorering for PostgreSQL 6. mai

Årsaken til at vi måtte skru av Cosmos DB, var fordi i løpet av testen 4. mai hadde Cosmos DB kostet 779,42 USD (se fig 5.5). Med en slik kostnad, var ikke Kongsberg Digital lenger interessert i flere belastningstester på Cosmos DB. PostgreSQL hadde kun kostet 1,60 USD. IoT Hub S2 kostet 6,29 USD. Dette gav en totalkostnad på 787,31 USD.



Figur 5.5: Kostnadsoversikt over de to forskjellige varmlagrings løsningene brukt i testfase to, vist logaritmisk.

5.2.2 Konklusjon

I denne testfasen tok Kongsberg beslutningen å avslutte testing av Cosmos DB på grunn av den høye kostnaden. Hvis vi antar at antall request units skalerer lineært med antall IoT Edger tilkoblet IoT Huben, ville kostnaden for en 24 timers test i testfase tre kostet 2 400 USD.

Antall partisjoner i IoT Huen viste seg å være en flaskehals hele tiden. Hadde vi oppdaget dette i en tidligere fase, kunne vi ha fått mer nøyaktig resultater. Det er kun prisen som hadde blitt påvirket av denne, og den hadde vært høyere enn det vi har presentert tidligere. Cosmos DB hadde med andre ord brukt flere request units, og ville trolig ha kostet mye mer enn forventet. Derimot vil dette ikke gjelde for PostgreSQL, siden det er kun langringsbegrensningen i antall GB som har noe å si på prisen.

5.3 Testfase 3

Testfase tre ble gjennomført 8. mai med 50 IoT Edger, hvor 2 virtuelle maskiner simulerte 25 IoT Edger hver. Vi valgte å bruke to maskiner med 8 vCPUer og 32 GB RAM, fordi vi ikke hadde tid til å prøve oss frem på et lavere nivå, og vi ville sikre på at maskinene hadde nok ressurser. Siden Cosmos DB i forrige fase ble utelukket som løsning i denne skalaen, ble siste test gjennomført kun med PostgreSQL. I denne fasen er IoT Huben skalert opp til en Standard Tier 3 for å kunne håndtere antallet meldinger som er forventet. Tabell 5.7 viser oppsettet for denne testfasen.

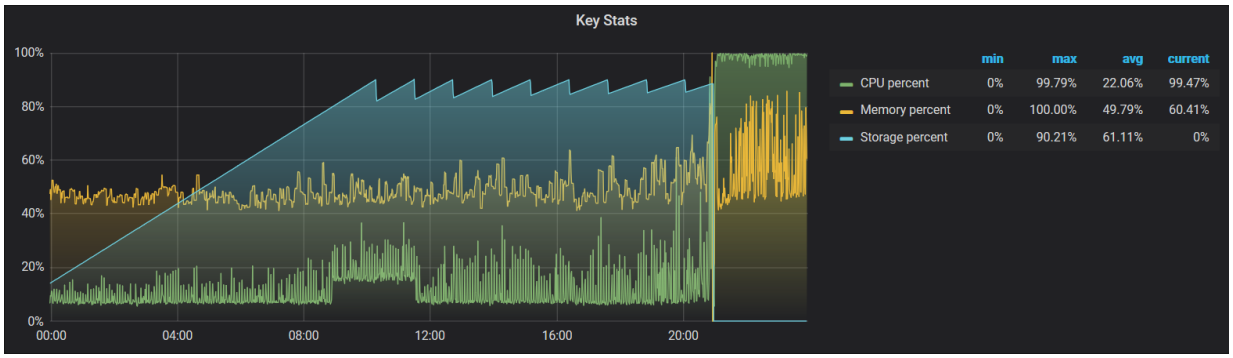
PostgreSQL	IoT Hub	Virtuell maskin
Basic tier 2 vCore Lagring: autoskalering	Standard tier 3 (S3) 32 partisjoner	2 stk (VM størrelse B8ms) 8 vCPUs, 32GB RAM, 100GB SSD

Tabell 5.7: Testoppsett 8. mai

5.3.1 Resultater

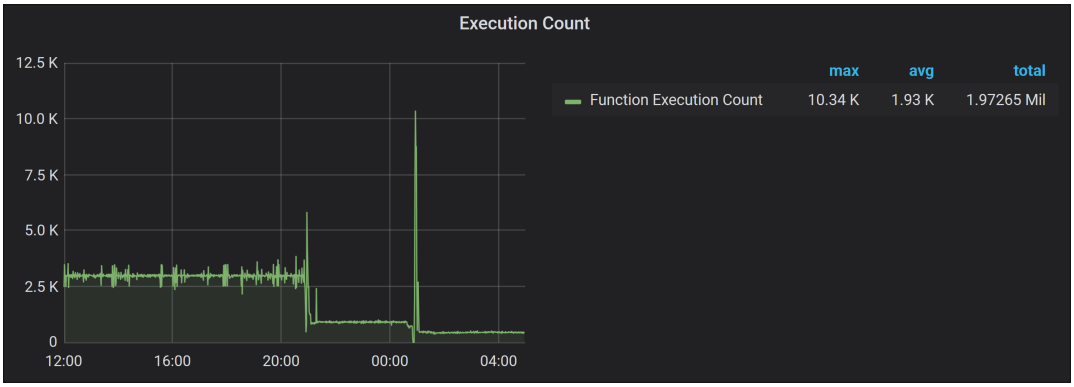
I testfase tre hadde PostgreSQL en kostnad på 1,70 USD og IoT Huben 62,90 USD. Dette resulterer i en totalkostnad på 64,60 USD.

I løpet av denne testen, fikk vi se hvordan systemet håndterer problemer som uforutsett nedetid. Azure database for PostgreSQL Basic tier har en oppetid på 99.99% [23], som medfører at det ikke er garantert at servicen er tilgjengelig til enhver tid. Klokken 20:54 fikk vi et varsel på at PostgreSQL servicen var nede, og at de undersøker problemet. Vi kan se på figur 5.6 hvor CPU bruken gikk ned til 0%.



Figur 5.6: Monitorering av PosgreSQL 8. mai

I figur 5.7 kan vi se at på samme tid som databasen gikk ned, begynte antall kjøringer for Azure Function å synke og stabilisere seg på et langt lavere nivå enn tidligere. Dette er en direkte bieffekt at PostgreSQL databasen gikk ned. Da vil instansene av Azure funksjonen forsøke å koble seg til PostgreSQL et visst antall forsøk, men som til slutt feiler. Dette fører til at det danner seg en kø til Azure Function, siden instansene bruker lengre tid på å sette inn i databasen enn tidligere, og antall kjøringer går også ned. Når da PostgreSQL starter igjen, blir den overveldet av antall nye tilkoblinger, noe som resulterer i at CPU og minnebruken øker kraftig. Det nivået av PostgreSQL vi hadde valgt, støttet kun 100 samtidige tilkoblinger. På denne måten danner det seg en flaskehals i PostgreSQL siden den ikke klarer å håndtere alle innsettingene.



Figur 5.7: Azure Function kjøringer 8. mai

5.3.2 Konklusjon

Vi kan anta at med en kraftigere instans av PostgreSQL med mer overhead for å kunne håndtere en større mengde med innsettinger som en følge av uforutsett nedetid. Overhead referer til overflyt av ressurser. Ved eventuell nede tid er det sentralt å ha ressurser som kan håndtere lagrings køen som bygges opp. Hvis man ikke har overhead i et produksjonssystem, er det risiko for at dataen blir forsinket inn i databasen. Dette kan medføre at dataen ikke vil vises i sanntid til brukeren. Nedetid på kritiske systemer kan potensielt skape store problemer for kunder som er avhengig av en konstant datastrøm med sanntidsdata. Det å kunne hente seg inn så raskt som mulig er høyst nødvendig.

Kapittel 6

Drøfting og analyse

I forrige kapittelet er det presentert resultater av det vi har implementert og belastningstestene. I dette kapittelet skal vi drøfte implementasjonen vår som en helhet og gjennomføringsprosessen.

Vi i gruppen ser oss fornøyd med hva vi har fått implementert med tanke på oppgavens totale omfang. Det har vært en utfordring å skalere oppgaven ned til et skop som er gjennomførbart på en bachelorskala. Ikke minst har det vært utfordrende å sette seg inn i nye teknologier og rammeverk som vi ikke har brukt før.

Det er mange komponenter vi gjerne skulle ha implementert i proof-of-consept systemet vårt, og testet hvordan systemet hadde oppført seg med litt annerledes arkitektur. Siden vårt system er et komponentbasert ende-til-ende system, vil vi også drøfte muligheten for videreutvikling.

6.1 Utvikling og testing av systemet

I denne seksjonen drøftes systemet vi har utviklet og hvordan det har vært å jobbe med det.

6.1.1 Valg av komponenter i systemet

Da vi skulle velge komponentene, fulgte vi referansearkitekturen (fig 2.1) til Microsoft mest mulig. Under prosjektet, har dette valget vist seg å være til stor fordel med tanke på tidsaspektet på oppgaven. I stedet for å bruke mye tid på å undersøke alle mulige komponenter og hva de kan tilby til systemet, kunne vi nesten gå rett på utviklingen av de enkeltkomponentene i referansearkitekturen. Ingen av medlemmene i gruppen har jobbet med Azure tidligere, så dette valget endte med å være kritisk for at vi kom i mål med proof-of-concept systemet som en helhet.

6.1.2 Identifisering av flaskehalser

Resultatene av systemet vi har utviklet har vist seg at flaskehalser kan oppstå når det skal testes med store laster. Det å kunne oppdage hvor flaskehalsene oppstår har vært essensielt, så her har Azure Monitoring vært til stor hjelp. Et eksempel her er at Azure funksjonene fortsatte å kjøre etter at belastningstestene våre hadde stoppet i testfase 2, pekte på at flaskehalsen kunne ligge i IoT Huben. Testen oppskalering av IoT Huben med flere partisjoner, kunne vi se at Azure funksjonene stoppet med en gang belastningstesten stoppet.

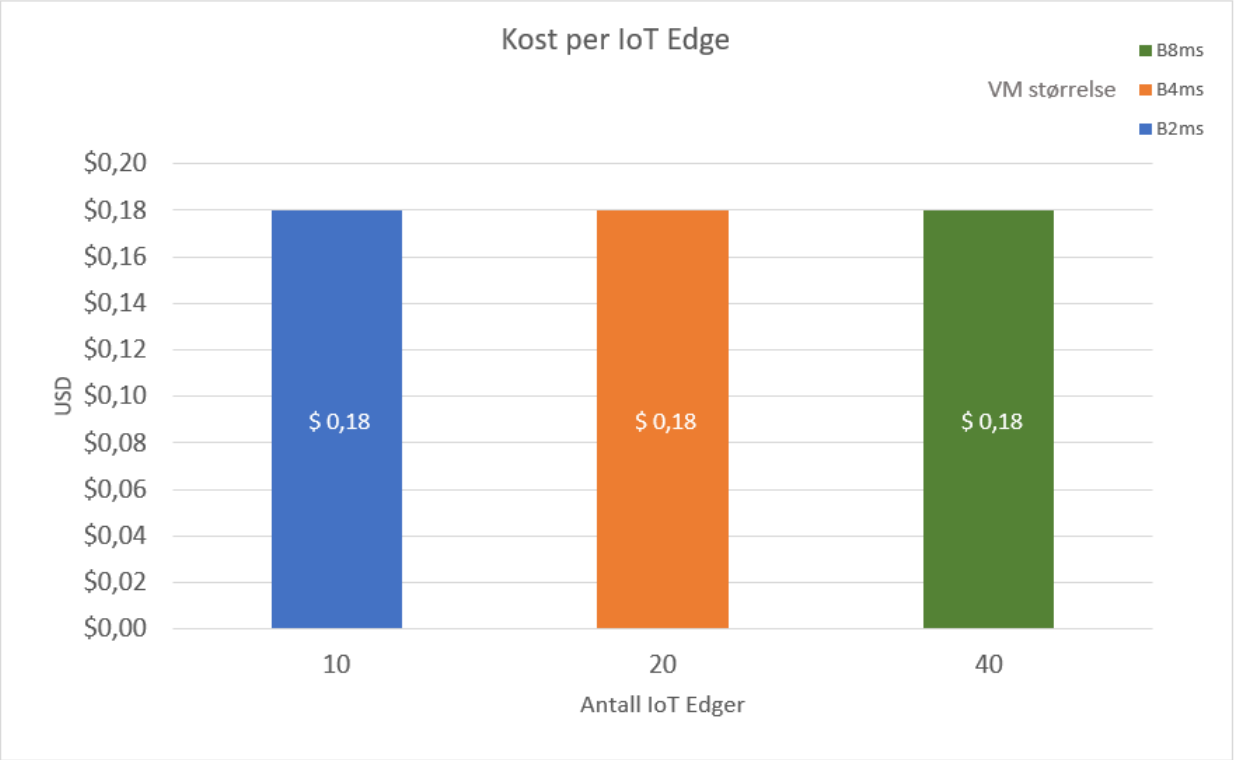
På samme tid som Azure Monitoring er et godt verktøy, tilbyr den ikke all nødvendig informasjon. I overvåkning av ressursbruken på virtuelle maskiner, er det kun status på CPU, nettverk og diskoperasjoner som blir rapportert. Dette medførte at vi ikke fant årsaken til at en maskin med 2 vCPUer og 8 GB RAM, ikke klarte å simulere mer enn 15 IoT Edger. Ut ifra hva Azure Monitoring rapporterte, kunne det se at CPUen gikk i 100%, noe som førte til at vi trodde at det var CPUen som var årsaken. Dette stemte ikke helt med hva vi hadde testet tidligere, hvor den samme virtuelle maskinen kunne kjøre 10 IoT Edger med kun 12% last på CPUen.

I senere tester viste det seg at det ikke var tilstrekkelig med RAM som forårsaket problemet. Hvis vi hadde brukt Monitoring Insights, som et ekstra verktøy for overvåkning av ressurser, kunne vi ha oppdaget det tidligere. Det negative med Monitoring Insights, er at det kan føre til en del ekstra kostnader.

6.1.3 Simulering av IoT Edger i containere

Det å kunne simulere større mengder med IoT Edger, har vært viktig for utviklingen og belastningstesten av systemet. Kongsberg Digital hadde tidligere brukt en dedikert virtuell maskin til en IoT Edge. Ved bruk av containere har vi kuttet store kostnader ved å kjøre alle IoT Edgene fordelt på en eller to virtuelle containere. I våre tester har vi primært brukt to maskiner, hvor hver maskin simulerer halvparten av IoT Edgene. Dette ble gjort fordi det tar tid å starte opp 50 nye instanser av en IoT Edge. Ved å da fordele det over to maskiner, har effektivisert igangsetting av en belastningstest.

Som tidligere nevnt har bruk av containere redusert kostander. Vi har undersøkt hva kostnadene per IoT Edge vil bli dersom vi dobler antall IoT Edger og størrelsen av den virtuelle maskinen. For å kalkulere kostnaden, har vi sett bort fra kostnaden for IP-adresse og disk, grunnet at disse kostnaden er for små til å gjøre en forskjell i resultatet. I figur 6.1 viser den blå stolpen at det koster 0.18$ å simulere 10 IoT Edger på en virtuell maskin med 2 vCPUer og 8 GB RAM (B2ms). Når antall IoT Edger dobles til 20 og den virtuelle maskinen dobles til 4 vCPUer og 16 GB RAM (B4ms), kan vi se at prisen forblir det samme per IoT Edge. Det samme skjer hvis vi dobler igjen til 40 IoT Edger og B8ms. Dette er resultatet av at prisen på den typen virtuelle maskiner vi har brukt skalerer linjert med resursene den får tildelt. Derfor vil også pris per IoT Edge forbli den samme når antallet dobles.



Figur 6.1: Kostnadsoversikt per IoT Edge for gitt et gitt antall som kjører på en virtuell maskin.

6.1.4 Læringskurven

Vi valgte denne oppgaven tidlig i bachelorprosessen, og var allerede hos Kongsberg Digital og fikk gjennomgang allerede tidlig i desember. Dette førte til at vi fikk startet tidlig med å lese oss opp på dokumentasjonen. En av de utfordringene vi var klar over før vi startet på prosjektet, var at læringskurven potensielt kunne bli et problem, så det å starte tidlig var viktig. Ingen av oss hadde jobbet med Azure tidligere, så det har vært mye dokumentasjon å lese på kort tid. Microsoft sin dokumentasjon viste seg å være svært omfattende til tider, noe som har ført til en lang læringsperiode. Det var spesielt vanskelig å få en god forståelse av referansearkitekturen til Azure (se figur 2.1) og hvordan den kan brukes i vår implementasjon. Vi som enkeltpersoner har hatt stort læringsutbytte av å utvikle dette systemet, men det har vært tidskrevende.

6.2 Arbeidsmetodikk

Prosjektet ble gjennomført med smidig utviklingsmetodikk utdypet i kapittelet om Gjennomføringsprosess, hvor vi forklarte hvordan vi ved å fokusere på en smidig utviklingsprosess. Vi bestemte oss tidlig i prosessen at vi ønske å jobbe smidig ved å bruke Scrum og Kanban som hjelpemidler til å holde god flyt i prosjektet. Vi som gruppe var allerede godt kjent med siden vi har brukt denne metodikken i tidligere prosjekter.

Metodikken vår gikk ut på å gjøre utviklingen mest mulig oversiktlig og strømlinjeformet ved å sette sprinter med 2 ukers intervaller. Dette viste seg å være en god ide, ettersom en del av komponentene vi skulle bruke var komplekse og tidkrevede å forstå. Ved å ha lengre enn den normale

1 ukers sprinten vi har hatt på tidligere prosjekter var det klart at å forlenge denne perioden på et komplekst prosjekt var svært nyttig.

Noe som var mer krevende var å beregne tid til oppgaver. Vi hadde ikke noe fast tak på hvor lang tid en oppgave maksimalt skulle ta, men det var nyttig å gjør et estimat ved sprint planleggingen. Siden Microsoft Azure var ny teknologi for gruppen, var læringskurven til tider flat. Dette førte til at det kunne ta lengre tid enn først antatt og det oppstod dominoeffekt hvor oppgaver måtte skyves over på neste sprint, siden vi ikke estimerte tiden godt nok.

Etter Kongsbergs kontorer stengte 10. Mars på grunn av covid-19 utbruddet, ble arbeidsflyten vår forandret. Ved å ikke sitte sammen og kunne diskutere problemer direkte, begynte vi i større grad å kommunisere over Slack og Google Meet. Vi prøvde så godt vi kunne å opprettholde den daglige standupen, dette viste seg til tider å være krevende, men vi ga i disse tilfellene oppdatering over Slack. Vi begynte også å ha tekniske samtaler over Google Meet hvor vi sammen prøvde å løse problemer som dukket opp.

Alt i alt mener vi at prosessen fungerte godt, og vi var i tillegg tilpasningsdyktige i takt med samfunns forandringene som fant sted under prosjektet. Ved å gjøre små forandringer i takt med prosjektutførelsen, tror vi at utfordringene ble løst på en hensiktsmessig måte.

Kapittel 7

Konklusjon

Målet med dette prosjektet har vært å utvikle et ende-til-ende IoT system ved bruk av Microsoft Azure standardkomponenter. I tillegg skulle prosjektet evaluere disse for å bidra til å gi oppdragsgiver et bedre beslutningsgrunnlag ved en eventuell migrering av sine eksisterende systemer til Azure komponenter.

Vi kan etter endt prosjektperiode vise til implementasjon av et skalerbart system, basert på standard Azure platform-as-a-service komponenter. Denne skal tillate en full ende-til-ende flyt av tidsseriedata fra sensorer i et Modbus nettverk, gjennom dataprosessering, mellomlagring i skyen og hele veien frem til visualisering i en frontend komponent.

Vi vil si at denne implementasjonen er et vellykket konseptbevis for et IoT system bygget på Azure standardkomponenter. Dette vil sammen med resultatene som presenteres i denne rapporten, være av stor verdi for oppdragsgiver som et solid beslutningsgrunnlag ved en eventuell migrering fra eksisterende egenproduserte systemer, til administrerte standardtjenester fra Microsoft Azure.

Denne rapporten vil også være av verdi for andre som ønsker å implementere et liknende system, da vi under arbeidet med prosjektet har kartlagt nødvendige komponenter, metoder og teknologier nødvendige for å konstruere et fungerende system for håndtering av sanntids tidsseriedata. Rapporten gir en innsikt i utviklingsprosessen, herunder utfordringer og tidsbruk knyttet til utviklingen av et slikt system. Dette kan brukes til mer presis planlegging og risikovurdering for andre lignende prosjekter, eller ved en evt. videreutvikling av det eksisterende systemet.

Gjennom vårt system har vi også vist at det er fullt mulig å ha ett større antall IoT Edge instanser kjørende på samme virtuelle maskin ved bruk av LXC containere. Resultatet kan medføre betydelig kostnadsreduksjon under utvikling mot IoT Edge.

Til slutt vil vi påpeke at prosjektet har hatt stor verdi for alle medlemmene av prosjektgruppen. Vi har fått verdifull erfaring med teknologier og konsepter vi ikke har tidligere kjennskap til fra studiene, og vi har lært mye om strukturering og gjennomføring av større prosjekter. Vi er stolte av resultatet vi kommet frem til, og har lært svært mye i prosessen.

Kapittel 8

Testing

I dette kapittelet vil vi vise og forklare hvordan vi har gjennomført testing av kode og systemet i sin helhet. Det har blitt gjort enhetstesting, kodeanalyse og systemtester. Det har vært spesielt krevende å gjennomføre systemtestene, med tanke på hvordan systemet vårt er spredt ut i forskjellige tjenester som må testes både enhetlig, hver for seg og integrert. Ved å gjennomføre testingen gradvis, har vi gjennom utviklingsprosessen kunnet validere at systemet fungerer og oppfører seg som forventet. Testingen av systemet har hatt stor verdi for oss, da dette har gitt oss grunnlag for å stole på kvaliteten til de resultatene vi har fått som følge av belastningstestene vi har utført.

8.1 Enhetstester

Ved å gjøre enhetstester underveis i utviklingsprosessen, har vi sikret oss en god flyt i testingen og oppnådd god kontroll over hvordan miljøet gradvis har utviklet seg. Vi har enhetstestet tre distinkte deler av prosjektet: Azure Function funksjonene, backenden og frontenden.

8.1.1 Frontend

Frontend enhetstestingen er testing av funksjonene som håndterer spørringer mot backend API-et og serverer websider. Uten enhetstesting ville vi ikke kunnet sikre kvaliteten på viktige funksjoner som manipulerer data eller håndterer innloging.

8.1.1.1 Rammeverk

På frontend applikasjonen vår har vi brukt Mocha som test rammeverk. Mocha gjennomfører automatisk tester og setter opp testmiljøet. Chai.js er et påstandsbiblotek som brukes for å kunne sette forventede verdier, HTTP-status koder og generell verdi sammenligning. Vi brukte også istanbul som er et testdekningsrammeverk laget for monitorering av testdekning og linje-for-linje test analyse.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	97.67	75	100	97.62	
providers	96.77	75	100	96.67	
apiRequestProvider.js	100	100	100	100	
authenticationProvider.js	92.86	66.67	100	92.86	56
providers/controller	100	100	100	100	
controller.js	100	100	100	100	

Figur 8.1: Testdekning for frontend implementasjonen

8.1.1.2 Testing

Enhetstestene som ble gjennomført på frontenden baserte seg hovedsakelig rundt datahåndtering og testing av forventet oppførsel, slik at man kan se hvordan programmet håndterer ikke forventet oppførsel og feilmeldinger. Ved å gjøre HTTP-request testing mot Backend API-et, kan man se hvordan frontenden håndterer ikke støttede HTTP statuskoder og tilgangshåndtering.

```
chai.use(chaiHttp);
describe("Edges", () => {
    it("Edges Test for user: frostad.fredrik@gmail.com", done => {
        chai
            .request('localhost:8080/api')
            .get("/getEdges")
            .set('Accept', 'application/json')
            .end((err, res) => {

                expect(res).to.have.status(200);
                expect(res.body[0].iotEdgeName).to.equals("edgeDev1");
                done();
            });
    });
});
```

Listing 8.1: Eksempel på enhetstest for testing av et lokokalt endepunkt for edge enheter

8.1.2 Backend tjenester

Backend og Azure Function sin testingen ble gjort i visual studio ved implementasjon av nUnit som test rammeverk og dotCover som testdekning rammeverk.

8.1.3 Backend API

Backend testene er hovedsakelig sentrert rundt CRUD operasjoner som refererer til database operasjoner hvor man gjør datahåndtering. Dette gjør at man må lage en test database internt i applikasjonen for å kunne gjennomføre tester.

8.1.3.1 Enhetstesting av backend

I dette avsnittet vil vi gi en beskrivelse av hvordan enhetstestingen i prosjektets backend applikasjon er utført. Vi vil gå gjennom bruk av dependency injection for å underlette tester, se på rammeverk vi har brukt for å skrive og kjøre testene, samt vise noen eksempler på tester.

Mocking av avhengigheter Vi valgte Moq som mocking rammeverk i backend applikasjonen fordi vi har god erfaring med dette rammeverket fra tidligere prosjekter, dokumentasjonen er god og kildekoden er open-source. Moq tillater oss å skape og konfigurere imitasjoner av klasser og deres tilhørende funksjonalitet. Dette og bruken av dependency injection gjør at vi kan erstatte en avhengighet i en funksjon vi tester, med en imitasjon av nevnte avhengighet hvor vi har full kontroll på dennes oppførsel og returverdier. Se avsnitt Eksempel på enhetstest.

Inversjon av kontroll - Dependency Injection er et designmønster som tillater oss å utvikle løst koblede programvarekomponenter.[24] Vi har i implementasjonen av backend applikasjonen benyttet oss av Konstruktør Injeksjon, da funksjonaliteten for dette i er bygget inn i .NET Core, som er rammeverket vi benyttet. Bruken av dependency injection og den løse koblingen dette skaper mellom komponenter, har gjort det enkelt å skrive gode enhetstester. Dette tillater oss å bruke mocking rammeverk for å erstatte avhengigheter en komponent har til en annen og dermed isolere funkjsonalitet i en enkeltkomponent under testing. I neste avsnitt vil vi vise et eksempel fra vår kildekode som beskriver dette.

Eksempel på enhetstest. Vi vil i dette avsnittet se på en eksempel test. Som eksempel har vi valgt en test som tester funksjonalitet i metoden `CreateCustomer` (fig 8.2) i klassen `CustomerController`.

```
[HttpPost]
[Consumes( contentType: MediaTypeNames.Application.Json)]
[ProducesResponseType( statusCode: StatusCodes.Status201Created)]
[ProducesResponseType( statusCode: StatusCodes.Status400BadRequest)]
1 usage  fredrikfrostad *
public async Task<ActionResult<Customer>> CreateCustomer(Customer customer)
{
    if (ModelState.IsValid)
    {
        customer.customerId = Guid.NewGuid().ToString();
        customer.Id = customer.customerId;
        await _cosmosCustomerService.AddItemAsync(customer);
        return Ok(customer);
    }

    return BadRequest();
}
```

Figur 8.2: Metode i CustomerController vi ønsker å teste

Dette er en enkel kontroller metode, som sjekker gyldigheten til en entitet som skal opprettes og persisterer entiteten i databasen. All funksjonalitet for dette er delegert til en service klasse som utfører forretningslogikken. Når vi skal teste kontrollermetoden, ønsker vi ikke å samtidig teste funksjonaliteten den er avhengig av. Vi må derfor mocke en instans av service klassen og dens metode `AddItemAsync()`. Ved hjelp av Moq kan vi lage en etterlikning (fig: 8.3) av avhengigheten, som returnerer en kjent verdi og dermed frikobler testresultatet fra implementasjonen av avhengigheten.

```
fredrikfrostad
public static ICosmosService<Customer> GetCustomerServiceMock()
{
    var mockService = new Mock<ICosmosService<Customer>>();
    mockService.Setup( expression: m :ICosmosService<Customer> =>
            m.GetItemAsync( id: It.IsAny<string>(), partitionKey: It.IsAny<string>())) // ISetup<ICosmosService<...>,...>
        .Returns(Task.FromResult(Customer1));
    mockService.Setup( expression: m :ICosmosService<Customer> =>
        m.GetItemsAsync( query: It.IsAny<string>())) // ISetup<ICosmosService<...>,...>
        .Returns(Task.FromResult(Customers));
    mockService.Setup( expression: m :ICosmosService<Customer> =>
            m.UpdateItemAsync( id: It.IsAny<string>(), item: It.IsAny<Customer>())) // ISetup<ICosmosService<...>,...>
        .Returns(Task.FromResult(UpdatedCustomer));
    mockService.Setup( expression: m :ICosmosService<Customer> =>
            m.AddItemAsync(It.IsAny<Customer>())) // ISetup<ICosmosService<...>,...>
        .Returns(Task.FromResult(Customer1));
    mockService.Setup( expression: m :ICosmosService<Customer> =>
            m.DeleteItemAsync( id: It.IsAny<string>(), partitionKey: It.IsAny<string>())) // ISetup<ICosmosService<...>,...>
        .Returns(Task.FromResult(Customer1));
    mockService.Setup( expression: m :ICosmosService<Customer> =>
            m.DeleteAllItemsByCustomerId(It.IsAny<string>())) // ISetup<ICosmosService<...>,...>
        .Returns(Task.FromResult(DeletedCustomers.ToList()));
    return mockService.Object;
}
```

Figur 8.3: Metode fra utility klasse som lager imitasjoner av CustomerService og returnerer kjente verdier

Siden vi bruker dependency injection gjennom kontroller, kan vi i oppsettet av testklassen for `CustomerController`, bytte ut `CustomerService` avhengigheten med vår mockede etterligning. I figur 8.4 ser vi at vi oppretter en ny instans av `CustomerController`, og at vi injiserer våre imiterte avhengigheter i konstruktørøren.

```
namespace BackendTest.Controllers.CosmosControllers
{
    [TestFixture]
    fredrikfrostad *
    public class CustomerControllerTest
    {
        private CustomerController _customerController;

        [SetUp]
        fredrikfrostad
        public void SetUp()
        {
            _customerController = new CustomerController(
                new Logger<CustomerController>(new LoggerFactory()),
                Mock.ServiceMock.CosmosCustomerServiceMock.GetCustomerMock(),
                cosmosUserService: new CosmosService<User>(),
                Mock.ServiceMock.CosmosEdgeServiceMock.GetEdgeservice(),
                Mock.ServiceMock.CosmosSensorServiceMock.GetSensorServiceMock(),
                cosmosAssetNodeService: new CosmosService<AssetNode>());
        }

        [TearDown]
        new *
        public void TearDown()
        {
            _customerController = null;
        }

        [Test]
        [TestCase( id: null,  partitionkey: "")]
        [TestCase( id: "",  partitionkey: null)]
        [TestCase( id: "",  partitionkey: "")]
        [TestCase( id: null,  partitionkey: null)]
        fredrikfrostad
```

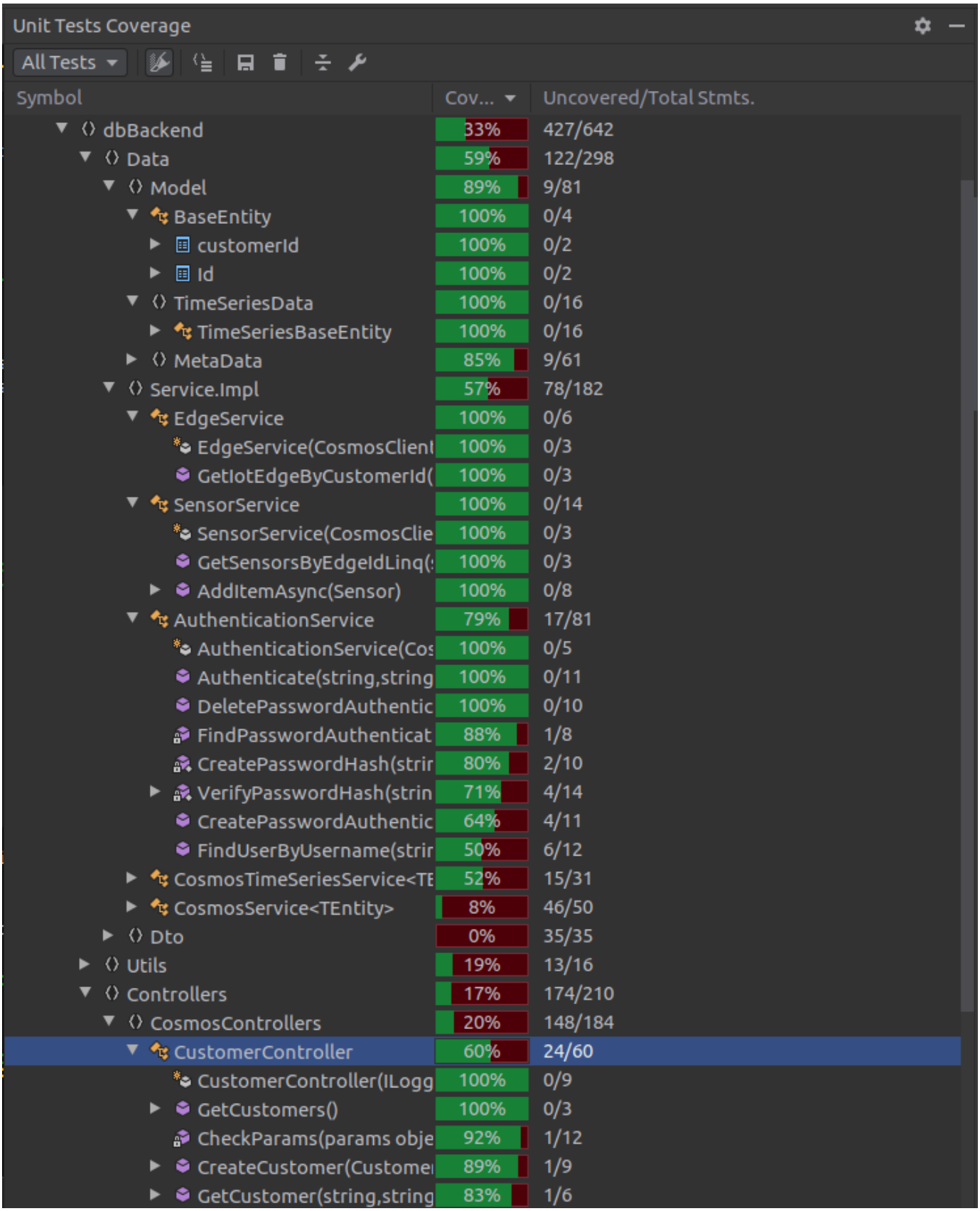
Figur 8.4: Metode fra utility klasse som lager imitasjoner av CustomerService

Til slutt har vi selve testmetoden (fig: 8.5). Denne testen oppretter et gyldig `Customer` objekt, og bruker dette for å teste funksjonaliteten i `CustomerController` sin `CreateCustomer` metode. Assert utsagnene sjekker at vi får tilbake et objekt av type `Customer` og at HTTP resultatet er av type `200-OK`

```
[Test]
fredrikfrostad *
public void CreateCustomer_shouldReturnCustomer_shouldReturnOkResult()
{
    // Arrange
    Customer customer = new Customer
    {
        CompanyName = "TestCompany",
        customerId = "TestCustomerId",
        DocumentType = "Customer",
        Id = "TestId",
        OpenIdConnectUrl = "TestOpenIdConnectUrl",
        TenantId = "TestTenantId",
        TenantShortName = "TestTenant"
    };

    // Act
    var res :Task<ActionResult<...>> = _customerController.CreateCustomer(customer);

    // Assert
    Assert.IsInstanceOf<Customer>(res.Result.Value);
    Assert.IsInstanceOf<OkResult>(res.Result.Result);
}
```

Figur 8.5: Testmetode som sjekker opprettelse av gyldig Customer objekt i CustomerController

Test-dekning Det kan være vanskelig å holde kontroll på hvor stor del av kodebasen som er dekket av enhetstester. For å lette dette arbeidet, har vi brukt DotCover som er en plugin tilgjengelig for det integrerte utviklingsmiljøet Jetbrains Rider. Dette er et svært godt verktøy, som gir god oversikt over testdekningen i et prosjekt.



Figur 8.6: Utdrag fra oversikt over testdekning i backend applikasjon

8.1.4 Azure Functions

Testingen av Azure funksjonene er sentrert rundt det å teste at tranformasjonen i funksjonene fungerer som den skal. Dette gjøres ved å for eksempel sjekke at JSON-objekter som kommer inn fra en consumer gruppe parses korrekt til objekter og flates ut slik at objektene kan settes inn i databaseløsningene våre.

Symbol	Coverage (%)	Uncovered/Total Stmts.
Total	64%	65/182
N00418-Function-apps	64%	65/182
N00418_Function_apps	64%	65/182
JsonParserFunction	0%	18/18
JsonTransformationPostgreSQL	0%	47/47
Content	100%	0/4
Data	100%	0/6
InputData	100%	0/8
JsonParserFunction_PostgreSQL	100%	0/9
HelperFunctions	100%	0/9
HelperFunctions()	100%	0/1
ParseTimestamp(string)	100%	0/8
Values	100%	0/10
SensorData	100%	0/20
SharedFunctions	100%	0/60
CreateSensorData(string,string,string,string,string,string,string,string,stri	100%	0/4
ParseJson(EventData)	100%	0/5
CheckEdge(SystemPropertiesCollection,ILogger)	100%	0/16
FlattenJson(InputData,ILogger)	100%	0/35

Figur 8.7: Utdrag fra oversikt over testdekning i Azure funksjoner

8.2 Statisk analyse av kode

Vi har brukt statisk kodeanalyse på både backend, frontend og funksjons applikasjonene for å gjøre at koden har konsistent formattering. Vi benyttet oss av ESLint [25] på frontend applikasjonen, som er et tilleggs bibliotek lagt til ved bruk av NPM(node package manager), intellicode, som er en del av visual studio og intellisense som er Visual Studio codes statiske kode analyse verktøy.

Statisk kodeanalyse hjelper til med å finne feil i kodenbasen, før den kompileres eller kjøres. Dette gjør at man sparer mye tid på rekompilering og kjøretidsfeil. Analysen blir gjort ved å definere regler som programmet skal følge i en config fil, som konstant kjører analyse på koden som blir produsert. Et typisk eksempel hvor statisk kodeanalyse er en stor fordel, er når man gjør komplekse boolske operasjoner, hvor man med en feil kan produsere falske positive utrykk [26].

Ved å bruke kodeanalyse verktøy som krever konsistent formatering får man bedre kode kvalitet i følger en satt standard, som igjen gjør koden lettere å forstå. Dette er spesielt viktig når flere personer jobber på samme applikasjon. Det konsistente formatet gjør det lettere for nye utviklere å sette seg inn i kodebasen når tiden for potensiell videreutvikling eller vedlikehold kommer.

8.3 Integrasjonstester

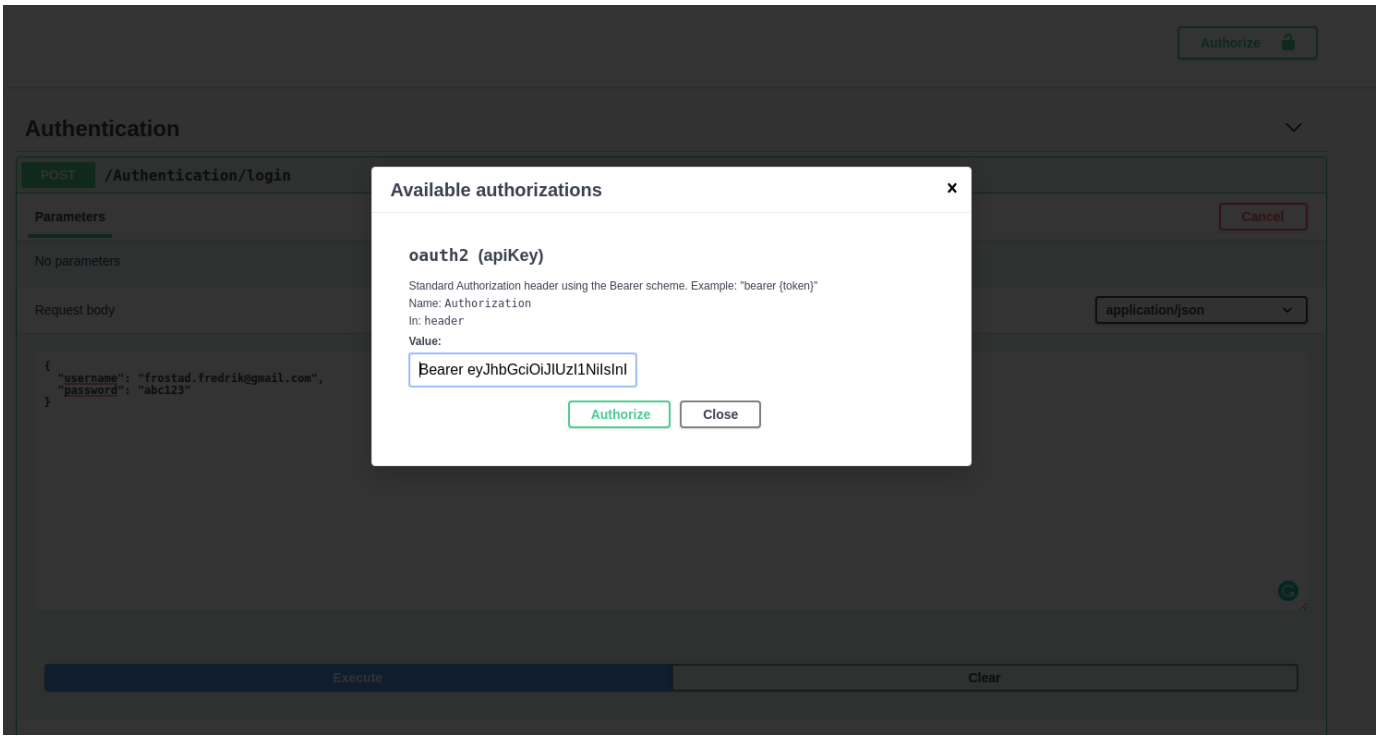
Integrasjonstesting går ut på å teste at alle komponentene i en applikasjon interagerer slik de tilsammen oppfyller de oppgavene systemet er satt til å utføre. Integrasjonstestingen av backend applikasjonen har blitt gjort ved hjelp av swagger mot produksjonsdatabasen.

8.3.1 Frontend

Ved å gjøre tester mot applikasjonen kunne vi se hvordan den håndterte å få uforventede verdier. For eksempel ble det gjort tester hvor passord eller email var feil for å se hvordan applikasjonen håndterte dette. Det ble også gjort tester hvor vi logget inn som en bruker som ikke hadde noen registrerte edge enheter, dette betyr da at brukeren ikke har noen sensor enheter hvor det ikke var mulig å navigere seg videre til visualiseringsplattformen. Tidsintervall ble også testet, hvor vi sjekket hvordan TSI-Client håndterte å ikke få noe data siden en gitt sensor ikke hadde produsert data i det valgte intervallet.

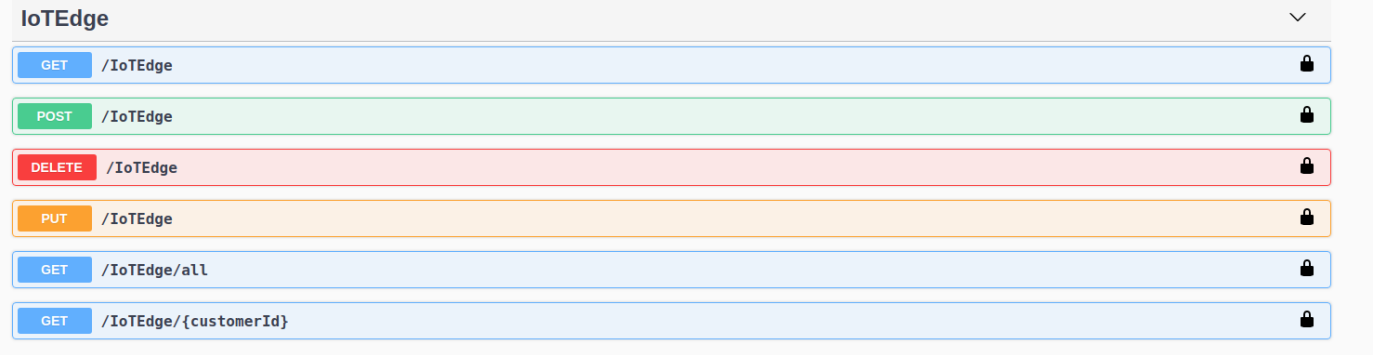
8.3.2 Backend

Swagger er et nyttig verktøy som krabber gjennom alle kontrollere i koden og genererer grafiske brukergrensesnitt for alle endepunkter som eksponeres av API-et. Disse endepunktene vi så tilgjengeliggjøres på `https://<applikasjonens-base-url>/swagger-ui`. Swagger tillater også testing av atorisasjonsløsning ved at autorisasjonstokenet fra backendapplikasjonen kan legges på request-headeren fra swagger (fig 8.8)



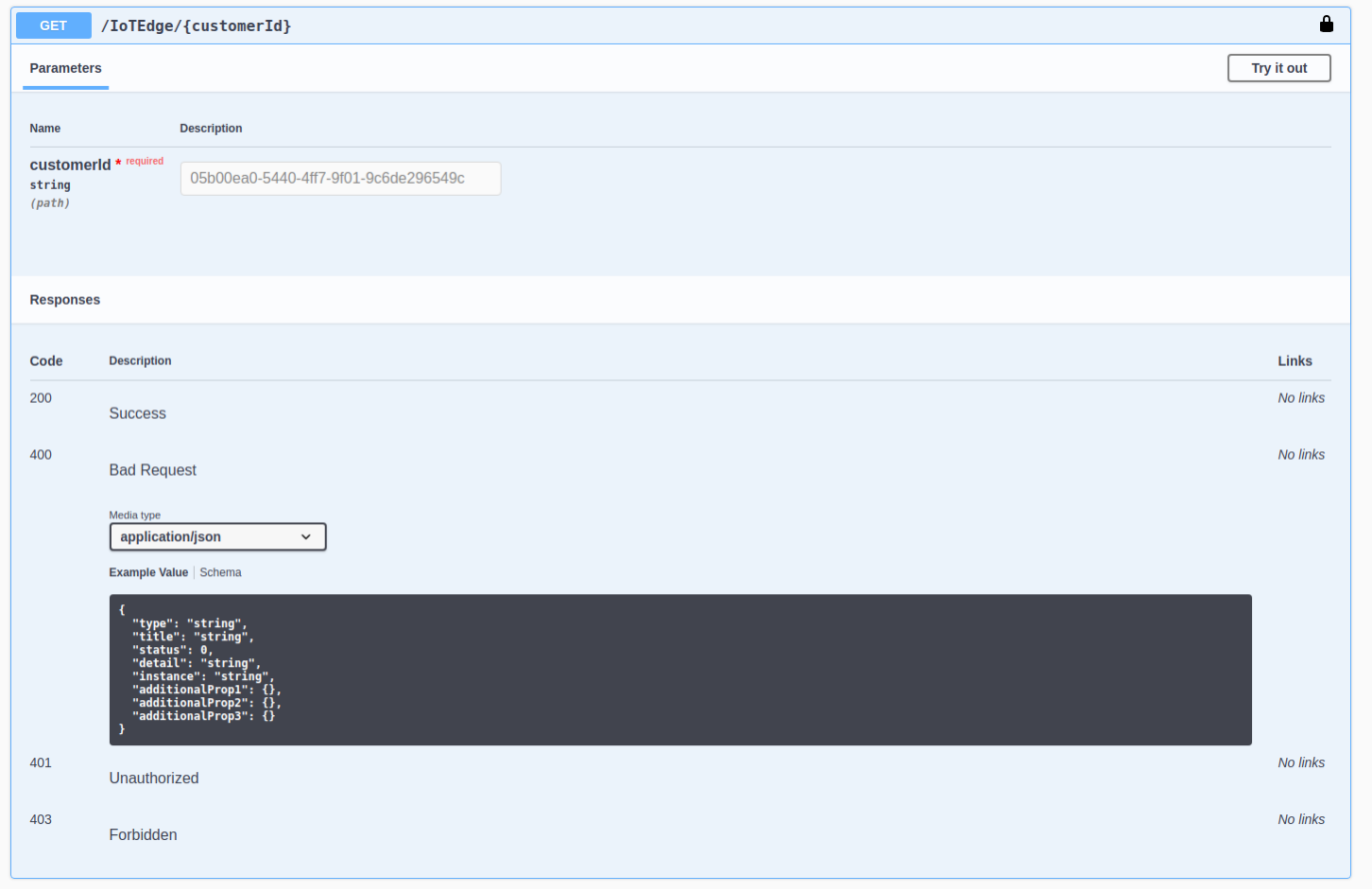
Figur 8.8: Registrering av Bearer token i swagger

Swagger grupperer endepunkter etter hvilken kontroller de tilhører (fig 8.9). Vi vil her vise prosessen for å teste endepunktet for å hente alle IoT Edge enheter som er registrert på en gitt `customerId`. Som vist i fig 8.8 har vi allerede logget inn en bruker og registrert brukeren sin token i swagger.



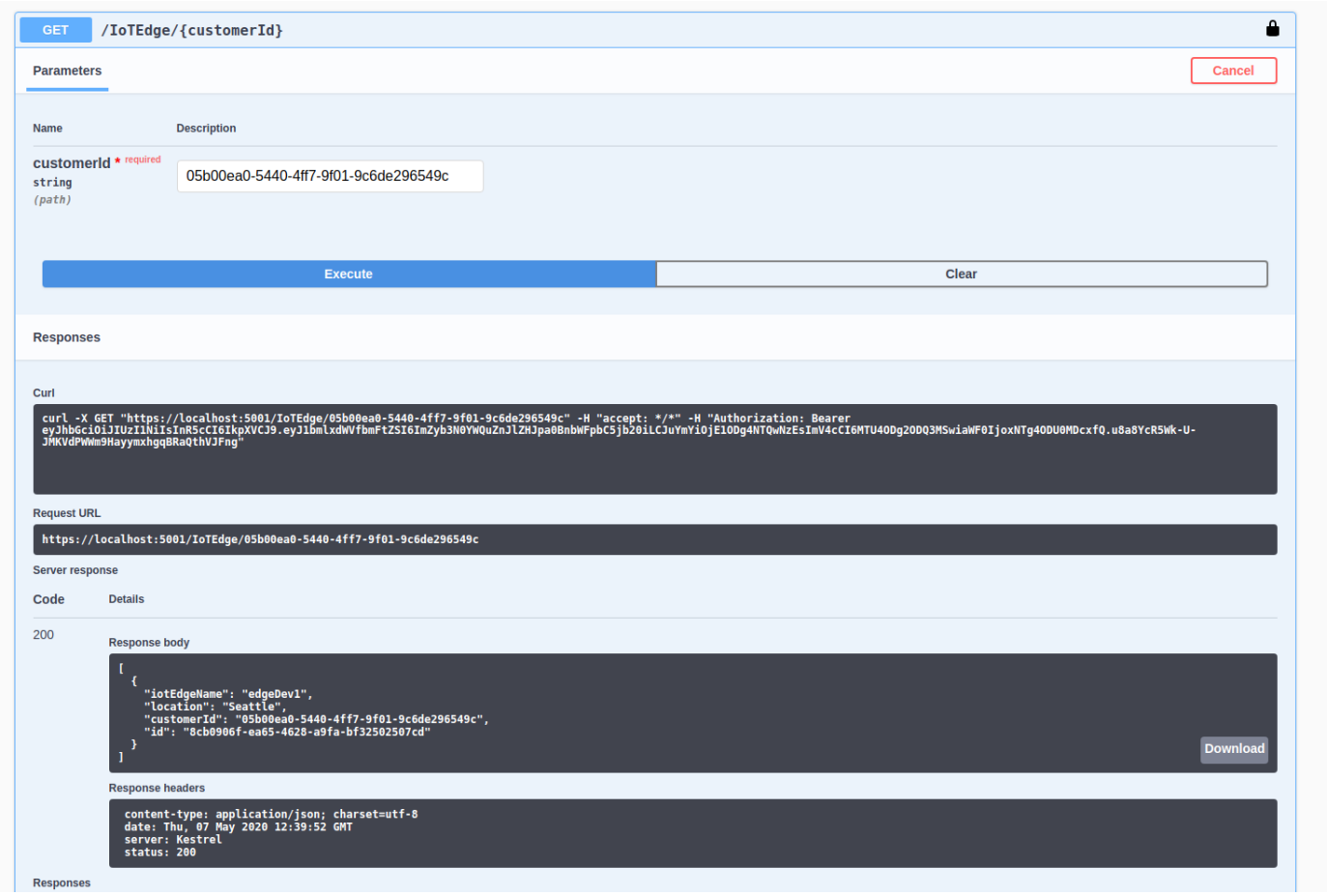
Figur 8.9: Alle endepunkter som hører til IoTEdge kontrolleren

Vi åpner fanen til endepunktet vi ønsker å teste, og trykker på **Try it out**. Vi ser at fanen også inneholder informasjon om endepunktet, slik som hva slags dataformat som returneres, og hvilke statuskoder endepunktet produserer.



Figur 8.10: Beskrivelse av endepunkt for å hente IoT Edge enhet basert på userId

Vi skriver inn en kjent `customerId`, og sender en spørring mot API-et ved å trykke **execute**. Da vil vi få tilbake en response med en statuskode, og alle objekter i databasen som matcher spørringen som gjøres av API-et.

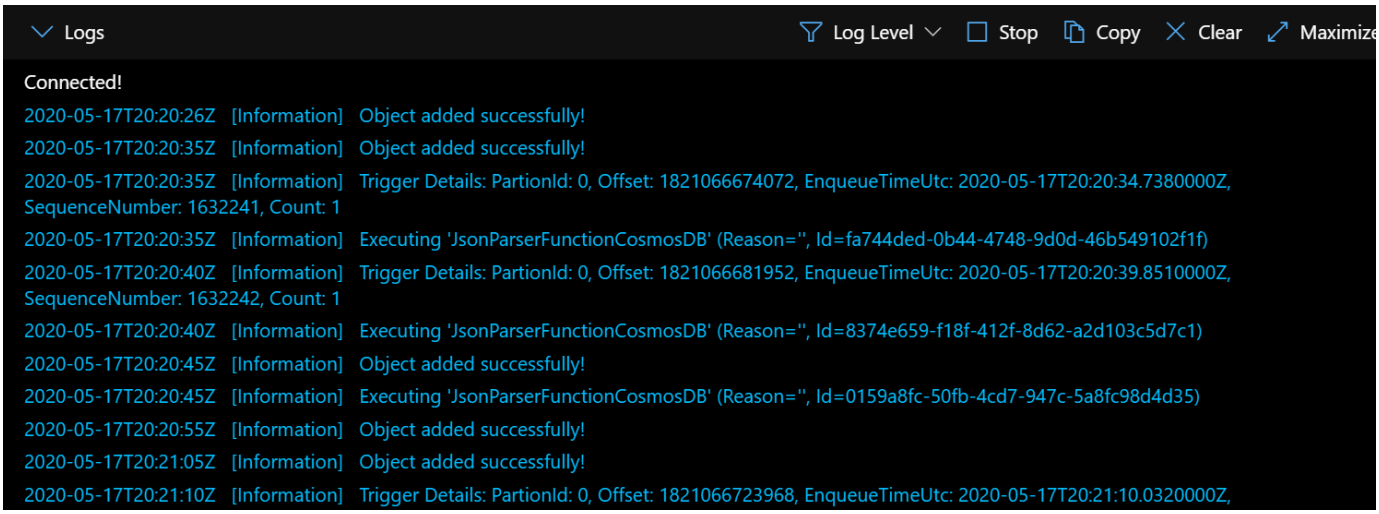


Figur 8.11: Resultat av spørring mot endepunkt

Oppsummering Integrasjonstestingen av backendapplikasjonen viste seg å være et svært godt verktøy for å finne feil i integrasjonen mellom backend og database, samt mellom de interne komponentene i backendapplikasjonen. Swagger har vist seg å være et godt verktøy for dokumentasjon av endepunkter ovenfor frontendutvikler og gjør det lettere for frontend å feilsøke integrasjonsproblemer mellom frontend og backend applikasjon.

8.3.3 Azure Functions

Integrasjonstestingen av Azure Functions foregår ved å sjekke at funksjonene blir kjørt og data blir lagt inn i lagringsløsningene når data kommer inn i IoT Huben.



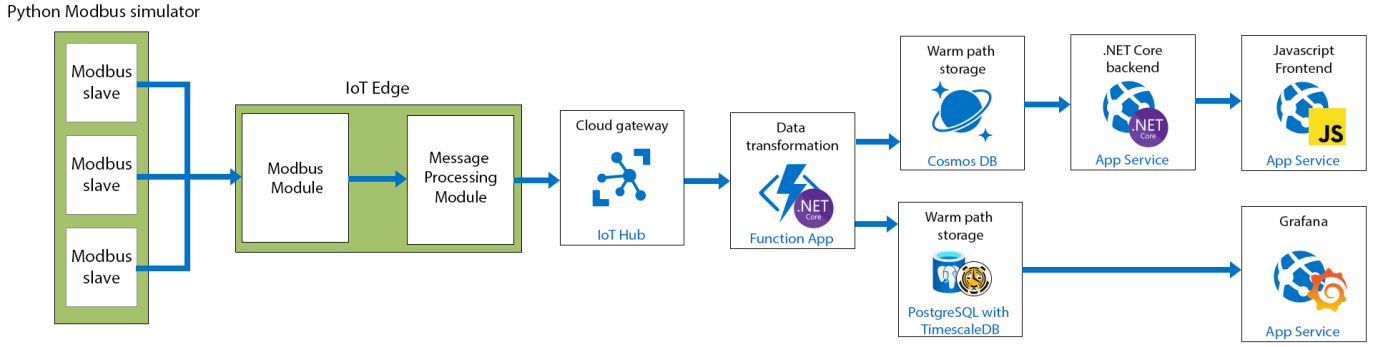
Figur 8.12: Skjermdump fra Azure Function App om suksessfull kjøring av funksjon

Som vi ser i figuren 8.12 har funksjonen blitt kjørt og lyktes med å sette objekter inn i databasen.

8.4 Systemtester

Verifisering av systemet har vært en sentral del av oppgaven hvor vi har testet at alle komponentene i systemet fungerer og sender korrekt data. Det at systemet produserer korrekt data gjennom hvert eneste bindeledd som til slutt gjør at vi produserer det forventede sluttresultatet. Vi har i stor grad benyttet oss av Postman som har gitt oss tilgang til å teste API endepunktene som knytter koden sammen.

Vi har gjort tester ved å sende inn data fra datasimulatoren som produserte et stort antall datapunkter fra IoT Edge instansene til test brukeren vår. Deretter vil dataene bli transformert av Azure Function som sender data videre inn i Cosmos DB og PostgreSQL. Deretter vil vi se i databasen at den forventede mengden har ankommet i riktig format. Til slutt kan vi gå inni frontend implementasjonene (Time Series Insights client eller Grafana), for å se om dataen gikk gjennom backend og frontend API-ene. For frontend var et testkriterie at dataen kun skal aksesseres av brukeren som eier den. Dette var for å simulere et produksjonsprogram hvor kundene av systemet bare har tilgang til deres egen data.



Figur 8.13: Ende-til-ende system test med simulert data

Det ble også gjennomført tester mot API-ene fra hver service hvor vi testet ikke-forventet formatering og autorisasjon. Dette ble gjort med Postman slik at det skulle være enklere å se om man fikk rett feilmelding, for eksempel ved at vi brukte feil innloggingsinformasjon og at vi forventet å få en HTTP statuskode `401` [27]. Denne testmetoden ble også mye brukt under utviklingen og er en av flere etablerte måter man kan teste REST API-er [28].

8.5 Akseptansetester

Akseptansetesting er det siste testnivået innen programvareutvikling. Formålet med akseptansetesting er å validere at systemet samsvarer med forretningskravene som stilles av oppdragsgiver og evaluere om det er klart for sluttleveranse.

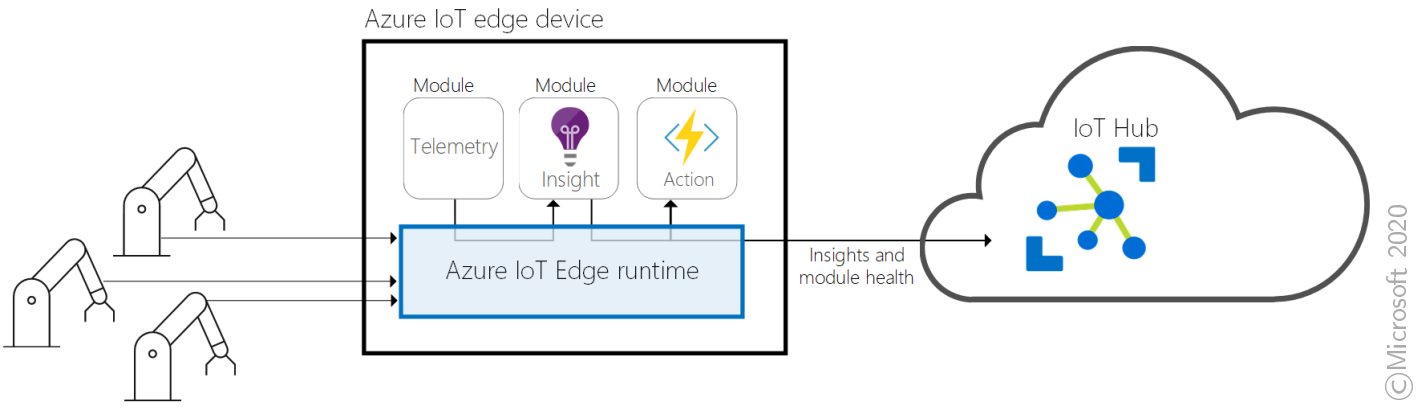
Systemet som er implementert i dette prosjektet, er kun ment som et demonstrasjonssystem. Dette systemet har som oppgave å underbygge at det funksjonelle kravet, ende-til-ende flyt av telemetrimeldinger, er oppfylt. Det har ikke vært noen krav om brukeropplevelse fra oppdragsgivers side, da dette ikke har vært prioritert i det kontekstsystemet har blitt utviklet. Dermed er det ingen reell sluttbruker for systemet, utover prosjektgruppen selv. Av denne grunn har vi valgt å ikke utføre noen akseptansetester for systemet utover å verifisere ende-til-ende meldingsflyt og separasjon av data basert på eier.

Kapittel 9

Systemet - Teknisk beskrivelse, arkitektur og implementasjonsdetaljer

I dette kapittelet gir vi en inngående teknisk beskrivelse av komponentene som inngår i systemet og konfigurasjonen av disse. I tillegg til å være teknisk dokumentasjon som grunnlag for en videreutvikling av systemet, vil dette kapittelet være av interesse for alle som ønsker å sette seg inn i teknologiene det bygger på.

9.1 IoT Edge



Figur 9.1: Konsept diagram for IoT Edge

Azure IoT Edge er en fullstendig administrert tjeneste bygget på Azure IoT Hub. IoT Edge tillater oss å flytte arbeidsoppgaver fra skyen til den fysiske lokasjonen der dataene genereres. Det kan være en fabrikk, et skip eller en oljeinstallasjon. Denne modellen har mange fordeler satt opp i mot å sende all rådata direkte til skyen for prosessering og lagring:

- Responstiden for kritiske hendelser blir lavere
- Ved bortfall av tilkobling kan telemetridata mellomlagres på IoT Edge frem til tilkoblingen er gjenopprettet.

- Telemetridata kan preprosesseres på IoT Edge og unødvendig data kan forkastes. Dermed kan man oppnå betydelige besparelser i båndbredde.
- Man kan trene maskinlagringsmodeller på data som er lagret i skyen, for så å bruke den ferdige modellen som en modul på IoT Edge

En IoT Edge enhet kan være en x86 eller ARM64 basert datamaskin som kjører enten Windows eller Linux som operativsystem. Vi har i denne oppgaven kun konsentrert oss om IoT Edge kjørende på Linux.

9.1.1 IoT Edge kjøretid

[29] Kjernen av IoT Edge er en kjøretidskomponent som tillater kjøring av logikk rettet mot skyen, og egenutviklet kode på en IoT Edge enhet. Vi vil i dette avsnittet gå gjennom de ulike modulene som til sammen utgør IoT Edge kjøretidskomponenten. Kjøretidskomponenten er ansvarlig for følgende funksjonalitet i en IoT Edge enghet:

- Kommunikasjon mellom IoT Edge og tilkoblede enheter nedstrøms.
- Kommunikasjon mellom IoT Edge og skyen (IoT Hub)
- Kommunikasjon mellom moduler som kjører på IoT Edge enheten
- Tilstandsrapportering til skyen for fjernovervåkning
- Sørge for konstant oppetid på alle moduler som kjører på IoT Edge enheten.
- Opprettholde sikkerhetsstandarder fastsatt i Azure Cloud
- Tillate fjerninstallasjon og oppdatering av programvare og arbeidsoppgaver på IoT Edge enheten

Arbeidsoppgavene til IoT Edge kjøretiden kan i all hovedsak deles inn i to kategorier: kommunikasjon mot skyen og modulhåndtering. Disse ansvarsområdene håndteres av to moduler som utgjør i IoT Edge kjøretidskomponenten: **IoT Edge hub** og **IoT Edge agent**.

9.1.1.1 IoT Edge cloud grensesnitt

Denne modulen håndterer kommunikasjon både internt i IoT Edge enheten og mot IoT Hub i skyen. Den fungerer som en lokal proxy for IoT Hub ved at den eksponerer de samme endepunkter og protokoller som IoT Hub. Dette tillater dermed klienter, i form av IoT Edge enheter eller moduler, å koble seg til IoT Edge kjøretiden på samme måte som om de skulle ha koblet seg direkte til IoT Hub i skyen. Tilkobling til IoT Edge hub støtter både **MQTT** og **AMQP**, men ikke **HTTP**.

Det er verdt å merke seg at IoT Edge cloud grensesnittet ikke er en fullverdig instans av IoT Hub. IoT Edge cloud grensesnittet har en del funksjonalitet som finnes i IoT Hub, men svært mange oppgaver vil også håndteres av IoT Hub. For eksempel autentisering. IoT Edge cloud grensesnittet vil videreformidle autentiseringsforespørsler fra en tilkoblet enhet til IoT Hub i skyen og

enheten autentiseres der. Deretter vil IoT Edge cloud grensesnittet cache autentiseringsinformasjonen lokalt slik at enheten som forsøker å koble seg til IoT Edge ikke må autentiseres via skyen ved neste tilkobling.

Dersom IoT Edge hub merker at tilkoblingen til skyen faller bort, vil den sørge for at telemetridata lagres lokalt og sykroniseres til skyen ved tilkoblingens tilbakekomst. I dette scenariet vil IoT Edge hub prioritere ny telemetridata fremfor eldre cachet data.

Iot Edge hub håndterer også kommunikasjon mellom moduler internt på IoT Edge enheten. Hver modul trenger bare å spesifisere endepunkter for sending og mottak av telemetridata og IoT Edge hub vil håndtere ruting mellom disse endepunktene.

9.1.1.2 IoT Edge agent

Denne modulen utgjør sammen med IoT Edge hub den andre halvparten av IoT Edge kjøretidskomponenten. IoT Edge agent har følgende ansvarsområder:

- Instansiering av moduler.
- Sørge for modulers oppetid.
- Rapportering av modulers status til IoT Hub.

Modul tvilling er et konsept fra Microsoft for å håndtere konfigurasjonsdata for IoT Edge Agent. En modul tvilling er et strukturert JSON dokument som beskriver konfigurasjonen av IoT Edge enheten, og alle moduler som kjører på den. Modul tvillingen lagres i IoT Hub, og holder tilstandsinformasjon for alle moduler som kjører på en gitt IoT Edge enhet.

Figur 9.2: Utdrag fra modul tvilling JSON

Ved oppstart av en IoT Edge enhet, vil en deamon prosess som kjører på enheten starte opp IoT Edge agenten. Agenten vil deretter hente sin modul tvilling fra IoT Hub, og parse denne for å se hvilke moduler som skal startes opp. Disse modulene er spesifisert i et *deployment manifest*, som er et JSON dokument som holder på all nødvendig konfigurasjonsinformasjon om modulene som skal startes. Disse modulene har form som Docker images, og vi har valgt å sette opp et eget avbildnings repository i Azure for å lagre alle modulene vi bruker i systemet. Dette repositoryet er integrert med Azure Pipelines og tillater oss å rulle ut nye konfigurasjoner basert på DevOps-prinsipper[30].

9.1.1.3 IoT Edge moduler

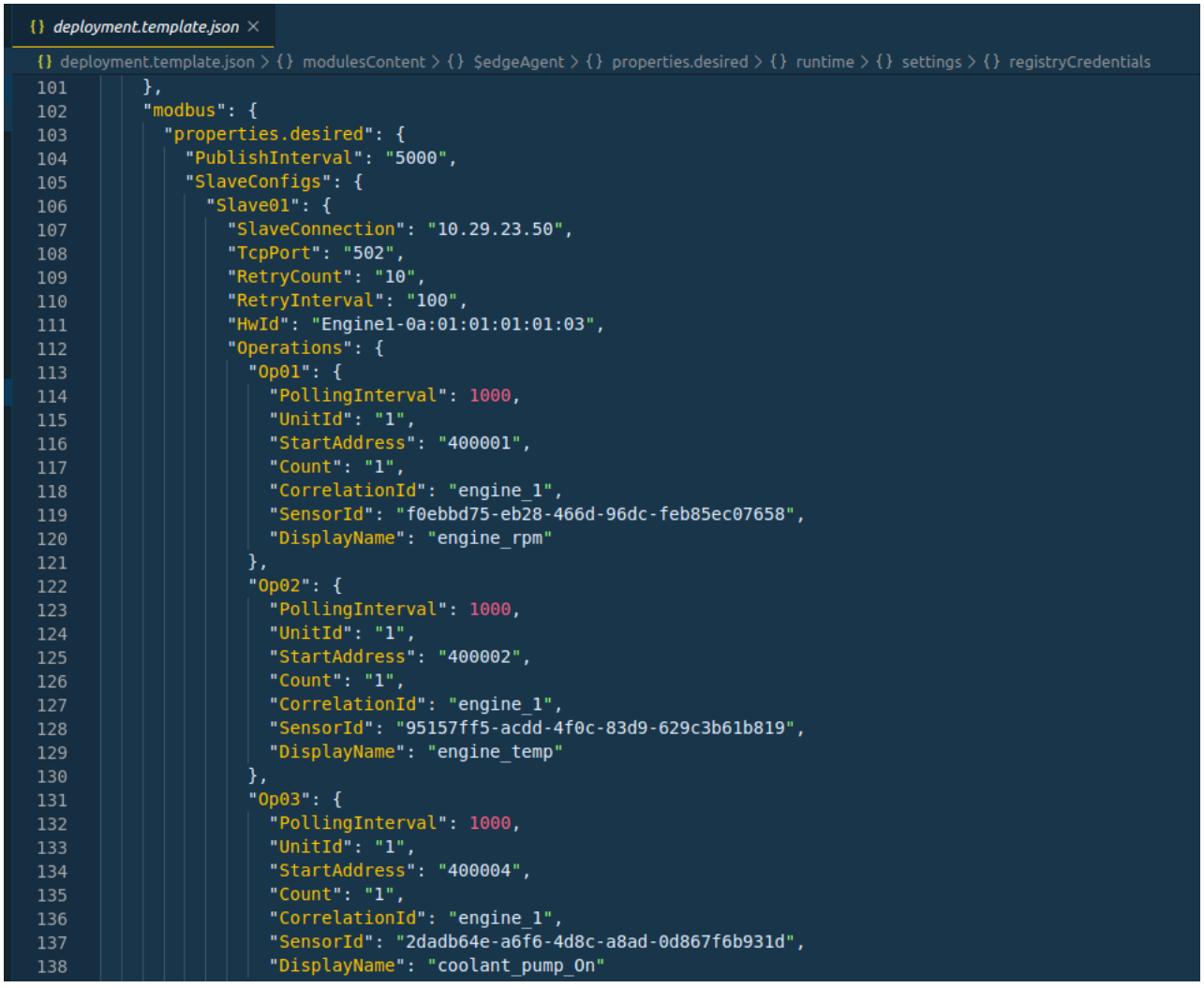
IoT Edge tillater å kjøre kode i form av moduler. En modul kan både være administrerte tjenester fra Microsoft, eller en tredjeparts utvikler, eller det kan være egenutviklet kode som er skreddersydd for brukers behov.

Man kan kjøre optil 20 moduler på en enkelt IoT Edge, og fritt rute data mellom disse modulene internt.

Vi har i vårt system benyttet oss av standard moduler fra Microsoft som vi har tilpasset til å passe inn i vårt system. Fordi oppdragsgiver ønsker at vi i så stor grad som mulig skal basere systemet på standardkomponenter har vi gjort så få endringer i modulenes kodebase som mulig.

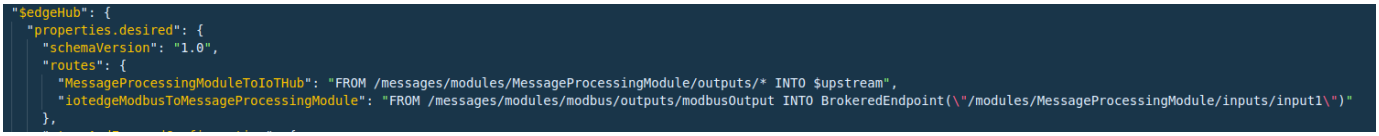
Avsnittene nedenfor beskriver modulene vi har brukt i vårt system:

Modbus Module er en standardmodul som er tilgjengelig fra Azure marketplace. Denne modulen tillater IoT Edge å fungere som en master enhet i et Modbus nettverk. Modulen gjør det mulig å mappe enheter fra Modbus nettverket, til en JSON tresktruktur som representerer enhetenes fysiske tilhørighet i nettverket. Mappingen gjøres via IoT Edge enhetens modul tvilling. Se figur 9.3 som eksempel.



Figur 9.3: Mapping av Modbus enheter i Modbus module

Message Processing Module er en modul med egen kode. Vi har inkludert denne modulen i vårt system for å demonstrere prosessen ved utvikling av egne moduler for IoT Edge. Vi har konfigurert IoT Edge slik at telemetridata fra Modbus modulen rutes til message processing modulen ved hjelp av Iot Edge Hub, (se fig 9.4) slik at denne kan overvåke alle telemetrimeldinger og utføre handlinger på disse, basert på konfigurasjonen i IoT Edge sin modul tvilling.



Figur 9.4: Ruting mellom moduler i IoT Edge

Temperatur simulator modul er en modul som er basert på Microsoft sin IoT Edge modul som heter SimulatedTemperatureSensor[31]. Siden Microsoft har kildekoden liggende ute på GitHub, har dette gjort det mulig for oss å modifisere denne modulen til å være tilpasset våre belastningstester. Det som er modifisert er fjerning av overflødig kode vi ikke vil få bruk for, og å legge til en ny funksjonalitet. Denne nye funksjonaliteten gjør det mulig skalere meldingsstørrelsen opp og ned gjennom modulens modul tvilling.

```
1 {
2   "properties.desired": {
3     "NumberOfMachines": 200,
4     "SendInterval": 1,
5     "NumberOfMessages": -1
6   }
7 }
```

Listing 9.1: Eksempel modul tvilling innstillinger for temperatur simulator modulen

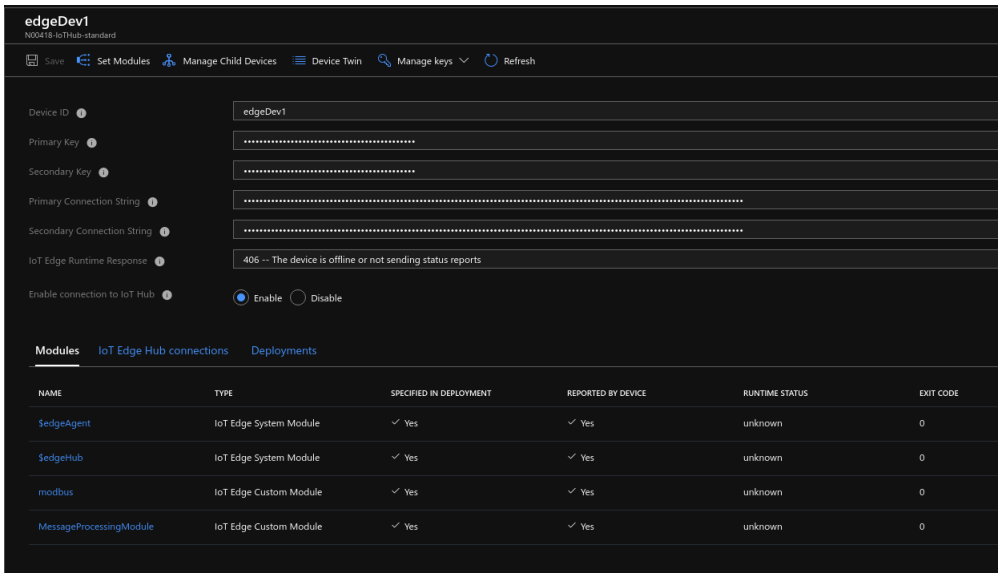
Det er tre variabler er nyttig å endre på (se listing 9.1):

- **NumberOfMachines** er antallet sensorer som vil bli generert. På denne måten øker og minker meldingsstørrelsen med antall sensorer.
- **SendInterval** er intervallet som det skal sendes telemetridata.
- **NumberOfMessages** er antall meldinger som skal sendes før modulen stopper. Hvis den er -1 vil den kjøre uendelig.

9.1.2 Oppsett av IoT edge - bruk av LXC containere

I dette avsnittet vil vi vise de viktigste elementene i prosessen med å sette opp en instans av IoT Edge i en LXC container som kjører på en Linux vm i Azure Cloud.

- Først må det legges til en IoT Edge enhet i IoT Hub. Når dette er gjort har det blitt opprettet flere nøkler og tilkoblingsstrenger som er knyttet til IoT Edge enheten, og som brukes for å autentisere denne mot IoT Hub.



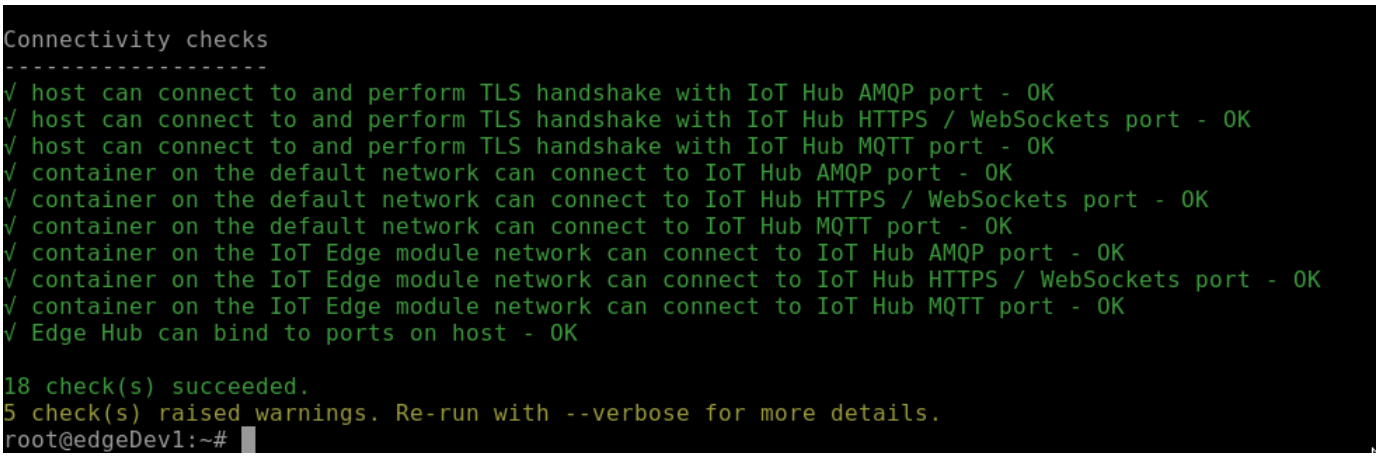
Figur 9.5: Statusside med id, nøkler og tilkoblingsstrenger

- Første trinn er å installere og sette opp LXD. Opprette en Ubuntu 18.04 container og installere IoT Edge kjøretiden på containeren. Dette gjøres enkelt med følgende kommandoer:

```
1  # Opprett og start en ny Ubuntu 18.04 container, vi gir containeren navnet edgeDev1
2  lxc launch ubuntu:18.04 edgeDev1
3
4  # Oppdater alle pakker i container
5  lxc exec edgeDev1 apt-get update
6  lxc exec edgeDev1 apt-get upgrade
7
8  # Installer Moby engine (brukes av IoT Edge for å kjøre containere)
9  lxc exec edgeDev1 apt-get install moby-engine
10
11 # Installer IoT Edge kjøretid på container
12 sudo apt-get install iotedge
```

Figur 9.6: Kommandoer for å opprette en Ubuntu 18.04 container og installere IoT Edge kjøretid

- Deretter kan vi logge inn i containeren og konfigurere IoT Edge kjøretiden ved å editere filen `/etc/iotedge/config.yaml`. Vi legger inn tilkoblingsstreng (fig 9.5) og deviceId (fig 9.5) fra IoT Hub .
- Etter å ha editert `/etc/iotedge/config.yaml` restarter vi IoT Edge kjøretiden med kommando `systemctl restart iotedge`, og sjekker at konfigurasjonen er korrekt med `iotedge check`. Se figur 9.7.



Figur 9.7: Utskrift fra `iotedge check` kommando på korrekt konfigurert enhet

- Utskriften fra `iotedge check` kommandoen (fig 9.7) viser at IoT Edge instansen er korrekt konfigurert og kommuniserer med IoT Hub.

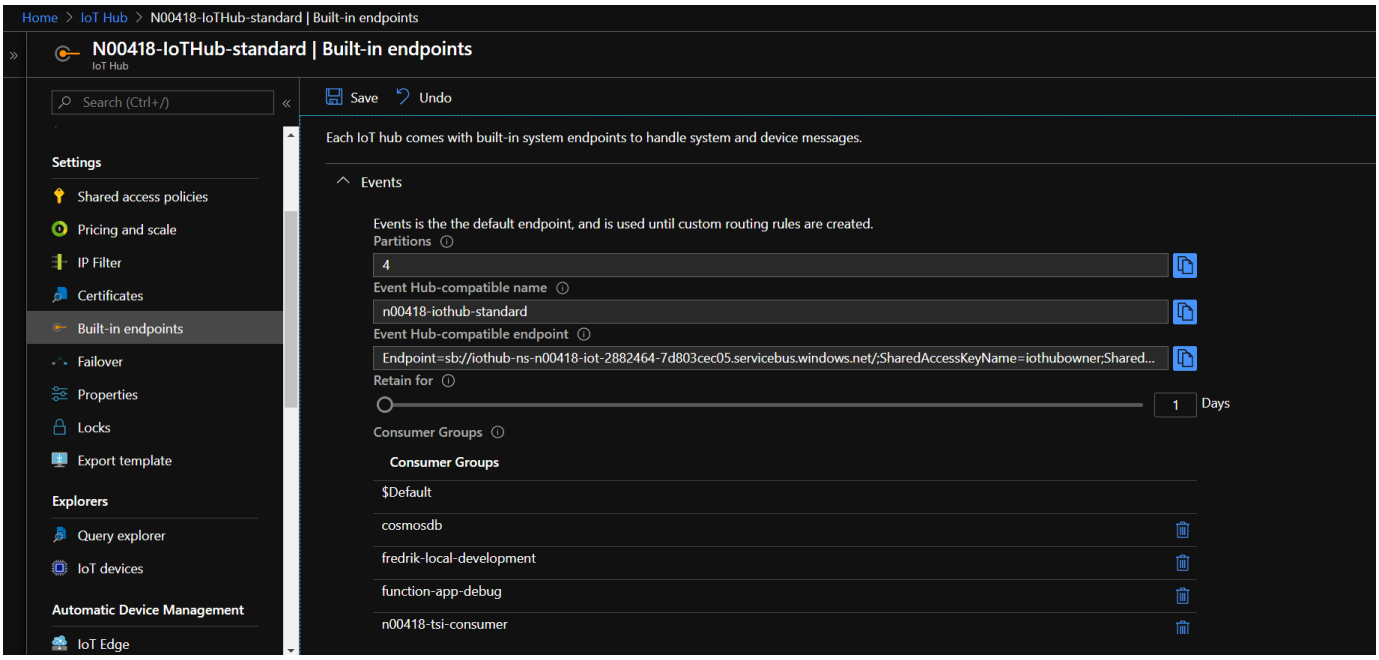
9.2 Azure IoT Hub

Azure IoT Hub er en tjeneste fra Microsoft som tilgjengelig i Azure. Rollen dens er å håndtere toveiskommunikasjons mellom IoT enheter og Azure[15]. Den håndterer meldinger som kommer fra IoT Edgene vi har, og sender konfigurasjonsparametere som module twin innstillinger eller deployments. For at IoT Hub skal kunne håndtere IoT Edge instanser, så må det være en Standard tier IoT Hub, siden Basic tier ikke har støtte for IoT Edge. Noe å ta notat av her er at hostname på IoT Edgen vil fungere som ID i IoT Huben, så dette må være en unik parameter.

9.2.1 Meldinger til IoT Hub

Antall meldinger som blir sendt til IoT Hub fra IoT Edger vil definere hvilken prisklasse av IoT Hub som må velges. En melding regnes som 4 KB, så for vårt bruk passer prisklasse S2, som tilbyr 6,000,000 meldinger per dag. Når meldingene fra IoT Edger kommer til IoT Hub er det flere muligheter for hva som kan skje dem. Den første er meldingsruting, hvor det blir definert et endepunkt hvor alle meldingene blir lagret. Dette kan for eksempel være en lagringscontainer som Blob storage. Den andre måten er den vi har benyttet oss av, som er ved hjelp av Consumer Groups.

Consumer Groups er en definert gruppe som skal kunne ta til seg data som er lagret i IoT Huben sin interne lagringsløsning. IoT Huben kan holde på meldinger opp til 7 dager. Når en Consumer Group har lest en melding, kan den ikke leses flere ganger. Den samme meldingen vil fortsatt være tilgjengelig for de andre gruppene til de også har lest den, eller til lagringsperioden har gått ut. For at meldingene skal kunne gå flere steder, så må det opprettes en Consumer Group for hver sti.



Figur 9.8: Konfigurasjon av Consumer Group i Azure

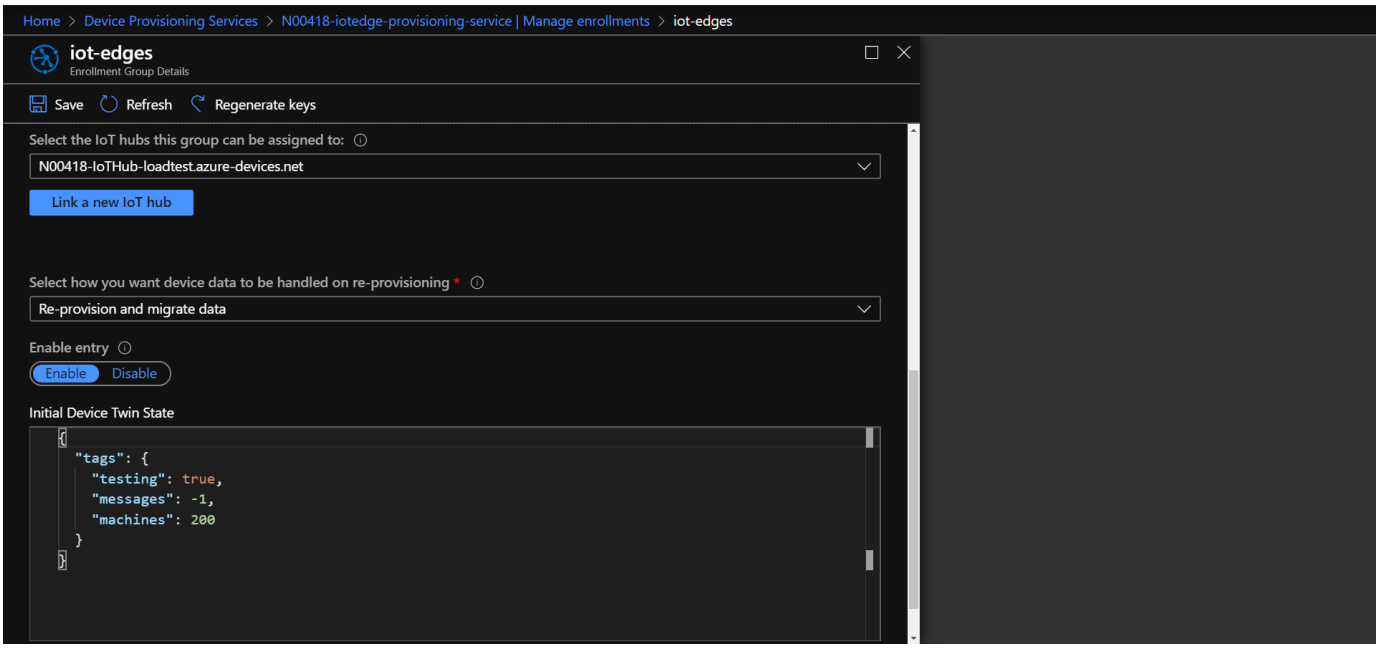
Partisjoner er en data organiserings mekanisme, som er relatert til parallelle nedhentinger av data for applikasjoner som bruker Consumer Groups. Antall partisjoner er direkte knyttet til antallet samtidige lesere som en IoT Hub er forventet å ha. For at Azure Functions funksjonene skal kunne konsumere data fort nok, er det viktig å ha et høyt antall partisjoner, for at Azure Functions skal kunne skalere riktig. Partisjonskonfigurasjonen må vurderes når det skal opprettes en ny Azure IoT Hub. Antall partisjoner kan ikke endres etter at IoT Huben er opprettet, så det er viktig å tenke frem i tid når dette skal settes.

9.2.2 Azure IoT Hub Device Provisioning Service

For å slippe manuell konfigurasjon av mange IoT Edger, tilbyr Azure IoT Hub Device Provisioning Service (DPS) registrering av nye IoT Edger og lastbalansering mellom flere IoT Huber hvis en har det. Vi har benyttet oss av registreringsfunksjonen i DPS når vi har gjort belastningstester. Ved hjelp av denne funksjonen, kan vi registrere større antall nye IoT Edger. I tillegg til å markere de med tagger for å kunne bruke deployments senere uten å måtte gjøre noe manuelt i Azure.

Det hadde vært ønskelig å bruke X.509 sertifikater for autentisering. Da vi implementerte registreringssystemet var dette kun støttet delvis i en tidlig utgivelse av IoT Edge kjøretiden, men ikke i Azure DPS. Derfor har vi benyttet oss av symmetriske nøkler, brukt som autentisering av IoT Edger ovenfor Azure DPS. For at IoT Edge kjøretiden skal vite at den skal bruke symmetriske nøkler, må konfigurasjonen endres. For at dette skal være mest brukervennlig for oss, så har vi laget et Bash skript som gjør denne konfigurasjonen for oss. Alt dette skriptet trenger, er IDen som IoT Edgen skal ha, primærnøkkelen til Azure DPS som kan hentes fra Azure Portal, og scope IDen til Azure DPS. Da vil skriptet ta hånd om generering av den symmetriske nøkkelen, i tillegg til å endre konfigurasjonen til IoT Edge kjøretiden.

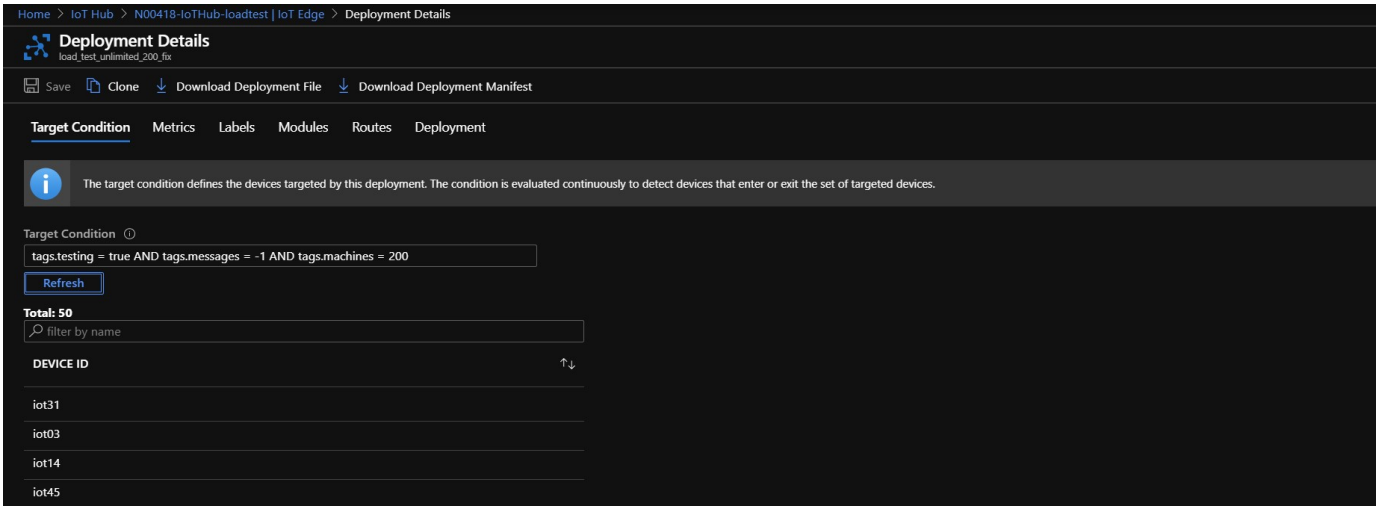
Når IoT Edgen har fått autentisert seg ovenfor Azure DPS, vil den bli tagget med noen variabler, (se figur 9.9) slik at Deployments i IoT Huben senere kan bruke disse.



Figur 9.9: Azure DPS registreringskonfigurasjon

9.2.3 Deployments

Deployments i IoT Huben er predefinerte konfigurasjoner for IoT Edger som kan tilpasses helt fritt. De kan konfigureres med moduler som skal ut til IoT Edger med ferdig konfigurasjon. For at IoT Huben skal vite hvilke IoT Edger som hører til hvilke Deployments, blir egenskapene som tagger eller navn til IoT Edgene brukt til å definere hvilken Deployment som skal sendes ut til en gitt IoT Edge.



Figur 9.10: Konfigurasjon av en Deployment mot IoT Edger

9.3 Kaldlagring

Kaldlagring er en del av lagringsløsningen vår hvor vi lagrer all data som sjeldent skal aksesseres og som ikke trenger rask aksesstid på. Kaldlagring brukes vanligvis til statistikk og maskinlæring. I vårt system inneholder kaldlagringen av eldre data som ikke skal aksesseres så ofte fordi de ikke lenger er relevante. Data lagres hovedsakelig for historikk.

9.3.1 Azure Stream Analytics

For filtrering av data til kaldstien har vi en Azure Stream Analytics jobb kjørende. Stream Analytics er et sanntidsanalyse og begivenhets-prosesserings verktøy designet for å kunne håndtere data fra flere kilder på en gang[20]. Stream Analytics har støtte for å ta imot data fra mange kilder samtidig, prosessere og sende til forskjellige konsumenter via SQL spørringer.

I vårt system brukes Azure Stream Analytics til å ta data fra Azure IoT Huben og sende det inn i Azure Blob for langstidslagring.

9.3.2 Lagring - Azure blob storage

Azure Blob Storage er Microsoft sin objekt lagringsløsning for skyen. Blob storage er optimalisert for å kunne lagre svært store mengder med ustrukturert data, det vil si at dataen ikke følger noen spesiell modell eller definisjon.[21]

I vårt system har vi tatt i bruk Azure Blob storage til langtidslagring av data fra den kalde stien. Dataene blir satt inn i blobber ved bruk av Stream Analytics uten noe videre transformasjon. Dataene blir i vårt system bare liggende der for langtids lagring, men datagrunnlaget som opparbeides her kan blant annet brukes til å drive maskinlæring, ved å kjøre dataene fra Blob storage inn i Azure Machine Learning.

9.4 Varmlagring

Varmlagring er en del av lagringsløsningen vår hvor det lagres data som vi trenger rask tilgang til og som skal brukes ofte. I vårt system tilsvarer dette ny data ikke er eldre enn 24 timer. Dette er fordi det meste av data som er eldre enn dette vil være mindre interessant å se på.

9.4.1 Datatransformasjon - Azure Function

Siden meldingene som kommer inn i IoT Huben er i JSON-format og flere nivåer dypt, er det ikke hensiktsmessig å sette disse direkte inn i en varmlagringsløsning. For å få meldingene på riktig format, er transformasjonen gjort ved hjelp av Azure Functions. Begrunnelsen for dette valget var at da har vi kontroll over hvordan dataen transformeres. Vi har også tatt hensyn til at Microsoft foreslår det i sin referansearkitektur (fig 2.1).

Azure Functions kjører kode som en serverless applikasjon, som betyr at vi som utviklere kun trenger å tenke på koden, mens Microsoft vedlikeholder den underliggende infrastrukturen. Vi valgte å gå for en Consumption plan hvor Microsoft skalerer funksjonens ressurser opp og ned

etter bruken, og prisen vil bli kalkulert kun ut i fra den tiden funksjonen kjører[32]. For at Azure Function skal kunne håndtere store mengder med data, så vil den skalere opp ved å starte maksimum 200 nye instanser av samme funksjon. En instans av en funksjon kan prosessere flere meldinger samtidig, så det er ingen grense på hvor mange ganger en instans kan kjøre[33].

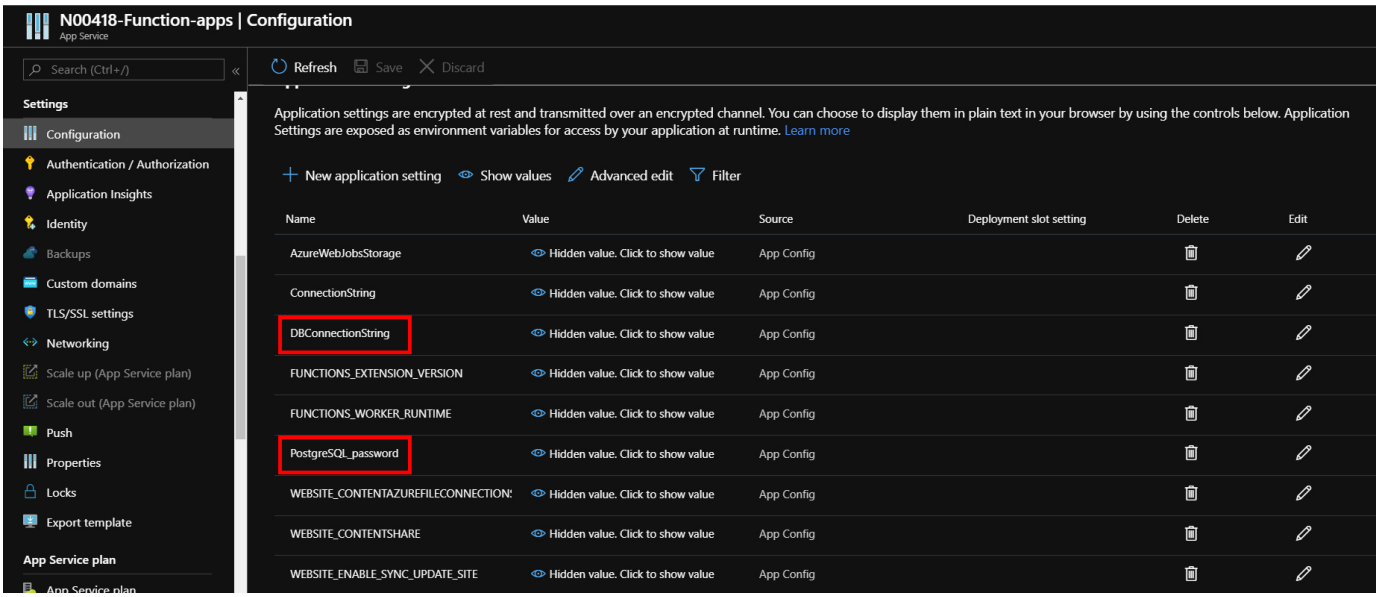
Vi har implementert to funksjoner som vil kjøre parallelt. Disse funksjonene benytter en del felles metoder for å transformere meldingene de konsumerer, men forskjellen kommer når de skal gjøre innsetting i hver sin respektive varmlagringsløsning, en til PostgreSQL og en til Azure Cosmos DB. Funksjonene vi har laget blir trigget hver gang det kommer en ny melding til IoT Huben. Da vil funksjonene først sjekke hvilken IoT Edge som meldingen kommer fra. Dette er for å separere POC-løsningen og belastningstestene. Hvis meldingen kommer fra en gyldig IoT Edge vil den deserialisere JSON-formatet til ett C# objekt som er identisk med trestrukturen. Deretter vil funksjonen plukke ut sensordataene i meldingen, og lage en liste som inneholder objekter med strukturen som vist under. Grunnen til at vi har valgt akkurat dette formatet, er for at dataen skal være mer effektivt å gjøre spørringer mot i etterkant. I tillegg til at det gjør det mulig å sette direkte inn i PostgreSQL.

```
1  public class SensorData
2  {
3      public string sourceTimestamp { get; set; }
4      public string publishedTimestamp { get; set; }
5      public string customerId { get; set; }
6      public string edgeId { get; set; }
7      public string correlationId { get; set; }
8      public string value { get; set; }
9      public string sensorId { get; set; }
10     public string displayName { get; set; }
11     public string hardwareId { get; set; }
12     public string quality { get; set; }
13 }
```

Listing 9.2: SensorData objektet

Når formatet er på listeform, er det på tide å sette inn i Cosmos DB og Postgres. Cosmos DB funksjonen er blitt konfigurert med en output binding gjennom en connection string. Det vil si at funksjonen har direkte kobling til Cosmos DB uten mer konfigurasjon. I Cosmos DB blir hvert element i listen satt inn som et eget dokument i JSON-format. Postgres derimot må bruke vanlige SQL kommandoer for å sette inn. For at dette skal gjøres mest mulig effektivt, blir dette gjort i bulker ved hjelp av Postgres sin COPY kommando. Da blir alle objektene i listen lagt inn i samme COPY kommando, før kommandoen til slutt blir utført. COPY kommandoen er vanligvis brukt for migrering av data, men det fungerer veldig bra for dette bruksområdet også, siden vi arbeider med større menger med data.

For å opprettholde cybersikkerhetsretningslinjene har vi holdt sensitiv informasjon som Cosmos DB connection strengen og PosgreSQL passord i miljøvariabler. Disse blir satt i Azure portalen (se figur 9.11) for å unngå at disse blir hardkodet i kildekoden, noe som vil føre til at disse vil bli committet til Git.



Figur 9.11: Azure Function miljøvariabler

9.4.2 Lagring

Kongsberg Digital ville at vi skulle se på flere alternativer for varmlagringsløsninger. I kapittelet under er det skrevet om de to lagringsløsningene vi har testet.

9.4.2.1 Azure Cosmos DB

Cosmos DB er Microsoft sin globalt distribuerte, multi-modell database tjeneste. Cosmos DB er laget for å kunne skalere både gjennomstrømning og lagringskapasitet etter behov, og for å kunne aksessere data lynraskt gjennom et utvalg API-er som inkluderer: SQL, MongoDB, Cassandra, Tables og Gremlin.[17]

Dokumentdatabase er en type database bygget rundt JSON-liknende dokumenter, dokumentdatabaser er laget for å være fleksible og enkle å jobbe med for utviklere. En av hovedforskjellene dokumentdatabaser har fra relasjonsdatabaser, er at skjemaet til dokumentdatabaser er dynamisk. Dette betyr at du ikke trenger å forhåndsdefinere skjemaet i en tabell slik du trenger i en tradisjonell relasjonsdatabase. Dette gjør det mulig at feltene kan variere fra dokument til dokument og at du kan modifisere strukturen når du vil.[34]

Databaser er Cosmos DB sin ekvivalent til SQL databaser, men disse inneholder containere og ikke tabeller.

Containere brukes av Cosmos DB til å holde på dokumenter. Disse containerne kan sammenlignes med en tabell i en helt vanlig SQL-database. Hver container krever at en partisjonsnøkkel spesifiseres ved oppretting av containeren.

I systemet vårt bruker vi 2 containere:

- Customer
- Data

I Customer lagrer vi informasjon om hver enkelt kunde, IoT Edger assosiert til hver kunde og sensorer assosiert til hver IoT Edge. Alt dette lenkes sammen med hjelp av en customerId som er unik for hver kunde og forteller deg hvilken kunde hver IoT Edge og sensor tilhører. Denne containeren er partisjonert på `customerId` for å gjøre det enklere og billigere å hente ut data. Det ligger også User dokumenter i containeren som brukes til innlogging i frontend klienten vår og autorisering mot backend API-et.

Customer containeren brukes til å lagre all informasjon relatert til kunder i bruk av henting av data. Containeren inneholder tre forskjellige typer dokumenter: Customer, Edge, Sensor og User. Customer dokumenter inneholder:

- `companyName` - Navnet på kundefirmaet
- `customerId` - En unik ID for å identifisere kunden og dataene tilhørende kunden
- `tenantId` - ID brukt i et Multi Tenant system
- `tenantShortName` - Navn brukt i Multi Tenant system

```
{
  "documentType": "Customer",
  "companyName": "Customer 1",
  "tenantId": "Tenant 1",
  "tenantShortName": "Tenant1",
  "openidConnectUrl": "openid.connect.url",
  "customerId": "05b00ea0-5440-4ff7-9f01-9c6de296549c",
  "id": "05b00ea0-5440-4ff7-9f01-9c6de296549c",
  "_rid": "YHgDAImGlXIWAAAAAAAAAA==",
  "_self": "dbs/YHgDAA==/colls/YHgDAImGlXI=/docs/YHgDAImGlXIWAAAAAAAAAA==/",
  "_etag": "\"7301f873-0000-0d00-0000-5e7b2c470000\"",
  "_attachments": "attachments/",
  "_ts": 1585130567
}
```

Listing 9.3: Eksempel på et customer dokument i customer containeren

Edge dokumenter inneholder:

- `iotEdgeName` - Et beskrivende navn til IoT Edge enheten
- `location` - Geografisk lokasjon av edge enheten
- `customerId` - IDen til kunden edgen tilhører
- `id` - Generert av Cosmos DB, brukes som edgeId

```
{
    "documentType": "IoTEdge",
    "iotEdgeName": "edgeDev3",
    "location": "Tokyo",
    "customerId": "ba3b912d-8be3-4fca-b985-9113a2262f1e",
    "id": "25e9e328-15fb-4323-9044-2d0e73ec3e68",
    "_rid": "YHgDAImGlXIbAAAAAAAAAA==",
    "_self": "dbs/YHgDAA==/colls/YHgDAImGlXI=/docs/YHgDAImGlXIbAAAAAAAAAA==/",
    "_etag": "\"740173ea-0000-0d00-0000-5e7b2f760000\"",
    "_attachments": "attachments/",
    "_ts": 1585131382
}
```

Listing 9.4: Eksempel på et edge dokument i customer containeren

Sensor dokumenter inneholder:

- `edgeId` - ID til edgen sensoren tilhører
- `HwId` - MAC-adresse
- `correlationId` - Id som brukes for å gruppere sensorer i edge enheten.
- `displayName` - Beskrivende navn av hva sensoren måler
- `customerId` - IDen til kunden sensoren tilhører
- `id` - Generert av Cosmos DB, brukes som sensorId

```
{
    "documentType": "Sensor",
    "edgeId": "8cb0906f-ea65-4628-a9fa-bf32502507cd",
    "HwId": "Engine1-0a:01:01:01:01:03",
    "correlationId": "engine_1",
    "assetnode_id": "none",
    "displayName": "engine_rpm",
    "customerId": "05b00ea0-5440-4ff7-9f01-9c6de296549c",
    "id": "f0ebbd75-eb28-466d-96dc-feb85ec07658",
    "_rid": "YHgDAImGlXI1AAAAAAAAAA==",
    "_self": "dbs/YHgDAA==/colls/YHgDAImGlXI=/docs/YHgDAImGlXI1AAAAAAAAAA==/",
    "_etag": "\"3905e2df-0000-0d00-0000-5e96cc7d0000\"",
    "_attachments": "attachments/",
    "_ts": 1586941053
}
```

Listing 9.5: Eksempel på et sensor dokument i customer containeren

I Data-kontaineren lagres all sensor data som kommer inn fra Azure Functions. Denne dataen er partisjonert på `customerId`.

Dokumentene vi tar i bruk for lagring av data følger et standard format for å kunne lett knytte hvert enkelt dokument opp til hvilken kunde det kom fra via en `customerId` og hvilken edge eller sensor data kom fra via `edgeId` og `sensorId`.

```
{
    "sourceTimestamp": "2020-05-11T08:58:20.518312Z",
    "publishedTimestamp": "2020-05-11T08:58:26.4370574Z",
    "customerId": "05b00ea0-5440-4ff7-9f01-9c6de296549c",
    "edgeId": "8cb0906f-ea65-4628-a9fa-bf32502507cd",
    "correlationId": "engineroom_climate",
    "value": "164",
    "sensorId": "1383e41c-261e-4aa4-ab82-eaa641c9f457",
    "displayName": "coolant_temperature",
    "hardwareId": "ClimateControl-0a:01:01:01:01:04",
    "quality": "good",
    "id": "749e2ac9-1729-465b-a13f-f422cd44cd42",
    "_rid": "YHgDAJ3jN2n4kRkAAAAAAA==",
    "_self": "dbs/YHgDAA==/colls/YHgDAJ3jN2k=/docs/YHgDAJ3jN2n4kRkAAAAAAA==/",
    "_etag": "\"7b03ed87-0000-0d00-0000-5eb913be0000\"",
    "_attachments": "attachments/",
    "_ts": 1589187518
}
```

Listing 9.6: Eksempel på et dokument i data containeren

Hvert dokument inneholder også noen andre felter:

- correlationId - Id som brukes for å gruppere sensorer i edge enheten.
- value - Den avleste verdien av gitt sensor
- displayName - Et navn som gjør sensoren lettere å kjenne igjen for mennesker
- sourceTimestamp - Tidspunktet målingen blir tatt
- publishedTimestamp - Tidspunktet målingen blir sendt mot IoT Huben

I tillegg til dette har dokumentet også noen verdier generert av Cosmos DB når dokumentet blir satt inn i containeren. Hvorav de 2 viktige av disse er id som er en unik id Cosmos DB genererer for hvert dokument, og ts som er tidspunktet dokumentet blir lagt til i Cosmos DB i Epoch tid.

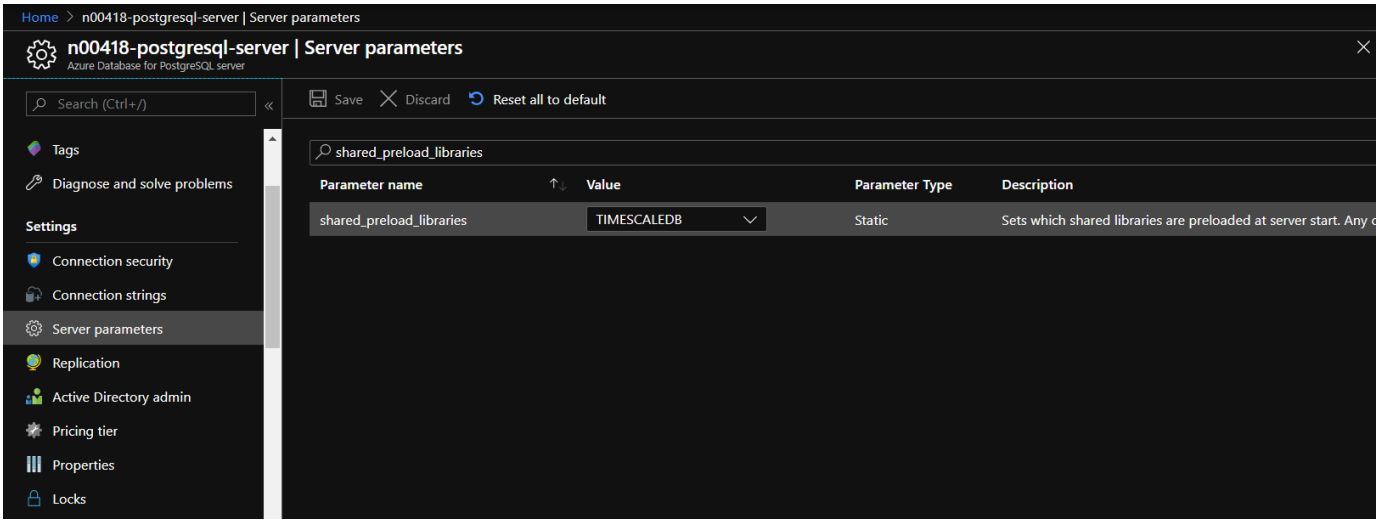
Request Units er Cosmos DB sin måte å måle gjennomstrømning. En Request Unit(RU) er definert som mengden beregninger krevd til å lese et dokument av størrelse 1KB. Request Units er definert på database eller container basis, i form av allokerte RU per sekund(RU/s). Hvis definert på database nivå, vil databasen ha en viss mengde å distribuere mellom containerne i databasen. Hvis det er satt på container nivå i stedet, setter du av en viss mengde til den containeren. RU/s fordeles også på to forskjellige måter, statisk og autopilot. Statisk gir en fast mengde RU/s mens autopilot vil skalere opp og ned etter behov opp til en øvre grense bruker har satt.

Partisjonsnøkkel er et felt som er satt ved oppretting av en Cosmos DB container, dette feltet kan ikke endres i ettertid uten å slette hele containeren. Partisjonsnøkkelens oppgave er å fortelle Cosmos DB hvordan den skal fordele dokumentene utover de forskjellige shard-ene for mest mulig effektiv spørring mot partisjonsnøkkelen. Dette vil si at hvis du f.eks. har ti dokumenter på vei inn til en Cosmos DB container, som er partisjonert på et felt kalt country, hvor fem dokumenter har country feltet satt til Norway, og de fem andre har Sweden, vil alle dokumenter med lik country havne i samme shard. I vårt system er alt partisjonert på en globalt unik identifikator som er customerId for å lett kunne hente ut all data tilhørende en kunde.

9.4.2.2 PostgreSQL med TimescaleDB

PostgreSQL (også kjent som Postgres) er en open-source objektrelasjonell database, mens TimescalDB er en open-source tidsserie SQL database pakket som en utvidelse av Postgres. Tilsammen skaper dette en database optimalisert for rask innsetting og komplekse spørringer på tidsseriedata[18]. Azure tilbyr PostgreSQL som en database-as-a-service, og i systemet vårt har vi brukt dette som en mer tradisjonell database sammenligning mot de andre lagringsalternativer som Azure Cosmos DB.

Oppsettet som er gjort for å få TimescaleDB til å kjøre sammen med Postgres ganske rett frem. Man må først å fremst ha en Postgres server kjørende i Azure. Det er viktig at denne er enten versjon 9.6 eller 10, siden dette er de siste versjonene i Azure som er støttet. For å installere TimescaleDB må dette gjøres i Azure portalen. Gå til Postgres databasen, velg server parameters, søk etter `shared_preload_libraries` og sett den til `timescaledb`



Figur 9.12: PostgreSQL konfigurasjon i Azure

Når endringene er blitt lagret må serveren restartes for å laste inn endringene. Når Postgres databasen har restartet må man logge inn på databasen, og kjøre følgende kommando:

```
CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;
```

Når dette er gjort, er TimescaleDB klar til bruk i databasen.

Tabellene som skal ta i bruk TimescaleDB må lages som et Hypertable, som optimaliserer tabellene for tidsseriedata. Dette gjøres ved å kjøre kommandoen `create_hypertable` med tabellnavnet og tidskolonnen som parameter. Postgres typen `TIMESTAMP` er mest egnet for dette, men hypertable støtter også datoer og integere, så lenge den er inkrementell.

```
loadtestdb=> CREATE TABLE testdata (
sensor_id INTEGER NOT NULL,
publish_time TIMESTAMP NOT NULL,
enqueued_time TIMESTAMP NOT NULL,
temperature DOUBLE PRECISION NULL,
edge_id TEXT NOT NULL,
message_number INTEGER NULL
);

loadtestdb=> SELECT create_hypertable('testdata', 'publish_time');
```

Listing 9.7: Kommandoer for å lage hypertable

Spørringen gjøres som vanlige SQL spørringer, men med TimescaleDB blir det enklere å gjøre spørringer på større datasett. Vi har blant annet brukt `time_bucket` funksjonen for å kun få tilbake data med ett minutts mellomrom, i stedet for å få returnert alle verdiene som også inngår i hvert minutt.

Som tilgangskontroll til Postgres er det konfigurert brannmurfilter som kun gir tilgang til våre tjenester internt i Azure, i tillegg til våre offentlige IP-adresser for lokal tilgang. Passordet som brukes å logge inn i databasen er et sterkt passord som er generert av en passordgenerator. På denne måten opprettholder vi cybersikkerhetsretningslinjene til Kongsberg Digital.

9.5 Backend

Backend applikasjonen i vårt system er implementert i C# og følger REST-arkitekturen for å eksponere funksjonalitet mot systemets frontend (klient) applikasjon. Backenden kobler klientens HTTP forespørsler opp mot funksjonalitet i applikasjonens kontrollere. Applikasjonen bruker ASP.NET Core Web API som rammeverk for å tilby REST-tjenester som kan konsumeres av klient. Backend applikasjonen i dette prosjektet har kun som oppgave å vise at en fullstendig ende til ende dataflyt i den valgte systemarkitekturen fungerer, og vi har derfor valgt en forholdsvis enkel implementasjon.

9.5.1 Rammeverk

Dette kapittelet gir en oversikt over de rammeverkene som er brukt for implementasjon av backend API-et.

9.5.1.1 ASP.NET Core

er et populært, kryss plattform, åpen-kildekode, web-utviklingsrammeverk for .NET platformen. Som utvikler er det enkelt å sette opp et .NET Core prosjekt og komme i gang med utvikling uten å måtte bruke for mye tid på å konfigurere applikasjonen.

Ved oppstart bygger en .NET Core applikasjon en vert som innkapsler alle ressurser som benyttes av applikasjonen, herunder:

- HTTP server implementasjon
- Mellomvare
- Logging
- Dependency Injection (Inversjon av kontroll)
- Konfigurasjon av applikasjonen

C# Program.cs

```
1   using Microsoft.AspNetCore.Hosting;
2   using Microsoft.Extensions.Hosting;
3
4   namespace dbBackend
5   {
        fredrikfrostad
6       public class Program
7       {
            fredrikfrostad
8           public static void Main(string[] args)
9           {
10              CreateHostBuilder(args).Build().Run();
11          }
12
            1 usage  fredrikfrostad
13          public static IHostBuilder CreateHostBuilder(string[] args)
14          {
15              return Host.CreateDefaultBuilder(args)
16                  .ConfigureWebHostDefaults(webBuilder =>
17                  {
18                      webBuilder
19                          .UseStartup<Startup>();
20                  }); // IHostBuilder
21          }
22      }
23  }
```

Figur 9.13: Oppstartsklasse for applikasjonen

Metoden `CreateHostbuilder` kalles av Main og inkapsler metodene `CreateDefaultBuilder` og `ConfigureWebHostDefaults` som konfigurerer verten med et sett grunnleggende opsjoner:

- Kestrel velges som webserver implementasjon
- Konfigurasjon lastes fra filene `appsettings.json` og `appsettings.{utviklingsmiljø}.json`
- Utskrift fra loggverktøy sendes til konsoll og debug providere.

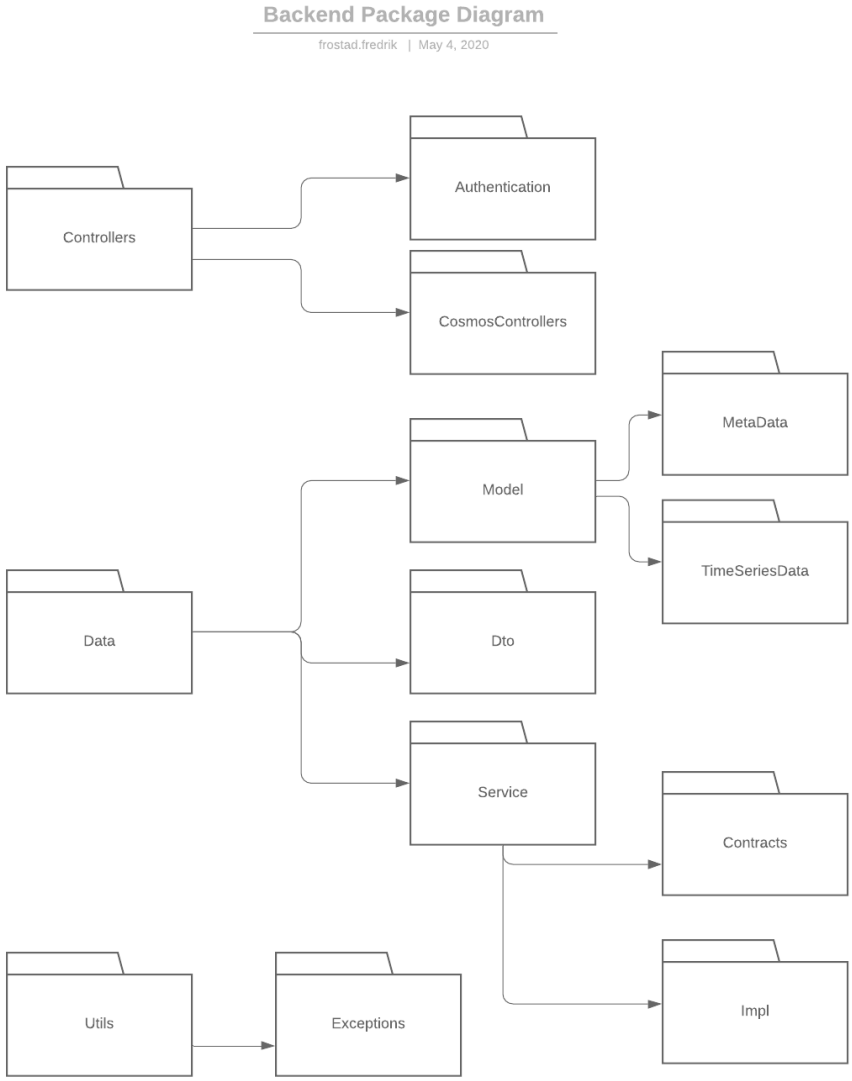
9.5.1.2 Azure Cosmos DB .NET SDK v3

er et rammeverk som tilbyr en intuitiv objektmodell for integrasjon mot Cosmos DB. I objektmodellen som benyttes av rammeverket er det en `CosmosClient` klasse på toppen, som igjen har to underliggende klasser: `Database` og `Container`. Disse klassene inneholder metoder som eksponerer databasen sin funksjonalitet for utvikler.

- `Database` klassen tilbyr operasjoner for å lese og slette data fra en eksisterende database
- `Container` klassen tilbyr operasjoner for å lese, endre eller slette objekter fra en spesifikk container i en database basert på objektets id.

9.5.2 Arkitektur

Dette delkapittelet beskriver de ulike klassene som inngår i API-et og interaksjonen mellom disse.



Figur 9.14: Applikasjonens Pakkestruktur

Videre i dette kapittelet går vi gjennom hver pakke med tilhørende klasser og presenterer dennes plass i arkitekturen.

9.5.2.1 Modeller (entiteter)

Klassene som ligger i `Model` pakken utgjør applikasjonens domenemodell. Domenemodellen består av flere modeller som persisteres og leses fra databasen ved hjelp av spørringer utført av `CosmosClient`. Modellene i applikasjonen er organisert i to sub-pakker:

- Metadata: all data som ikke er telemetridata
- TimeSeriesData: all telemetridata

I avsnittene nedenfor beskrives klassene som utgjør domenemodellen.

Model/BaseEntity er en baseklasse for alle entitsklasser i domenemodellen. Den inneholder attributter som er felles for alle andre entiteter. I denne implementasjonen vil det si CustomerId, som brukes som partisjonsnøkkel i databasen, og en unik Id som identifiserer entiteten. Vi har valgt å la alle entitetsklasser implementere en baseklasse fordi dette tillater utstrakt bruk av generics i implementasjonen av service klassene, og dermed kan vi unngå å skrive en masse duplikatkode for funksjonalitet som er svært lik mellom klassene, da CRUD funksjonalitet mot databasen vil være mere eller mindre identisk for de forskjellige entitetene.

`CustomerId` attributten som tilhører `BaseEntity` brukes for å knytte alle tilhørende entiteter til en enkelt kunde, og denne atributten brukes i tillegg som partisjonsnøkkel i Cosmos DB (se partisjonering av data).

Model/Metadata/Customer representerer en tenkt kunde i vårt demostrasjonssystem. Denne entiteten implementerer `BaseEntity` entiteten og får `customerId` og `Id` attributter fra denne. Alle andre entiteter i domenemodellen knyttes til en tilhørende kundeentitet, ved at disse implementerer `BaseEntity` entiteten og dermed får en `CustomerId` attributt som knytter disse mot en kunde.

Model/Metadata/User representerer en bruker tilknyttet en `Customer` entitet. En bruker kan kun være tilknyttet en `Customer` entitet. En bruker kan ha flere roller som er definert ved en indre `Role` klasse. `Role` er implementert med nøkkelordet `sealed` som forhindrer subklassing, og har privat setter for å unngå at en satt rolle kan muteres etter instansiering. User entiten brukes for autentisering og autorisering i systemet, i fellesskap med entiteten `PasswordAuthenticationData`

Model/Metadata/PasswordAuthenticationData er en entitet som knytter en bruker til sin autentiseringsinformasjon. Denne entiteten inneholder felter for hashet passord, salt og unikt brukernavn (epost).

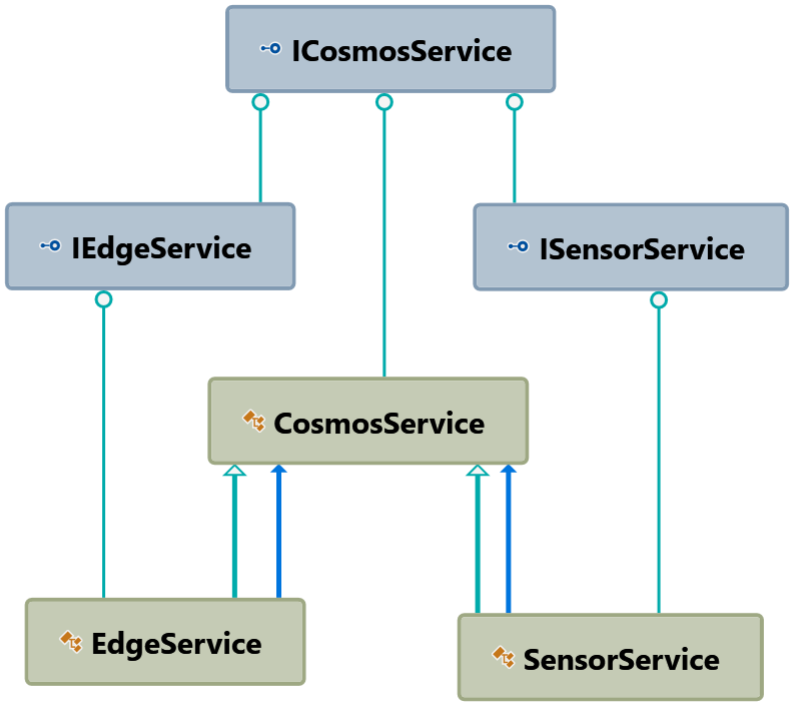
Model/Metadata/IoT Edge entiteten representerer en fysisk IoT Edge boks som eies av en kunde. Denne knyttes til kunden via `CustomerId` attributten.

Model/Metadata/Sensor Hver IoT Edge entitet er tilknyttet en lang rekke sensorer. Disse er representert ved `Sensor` entiteten.

Model/TimeSeriesData/TimeSeriesBaseEntity entiteten er representasjonen av telemetridata i systemet. Denne entiteten er eies av en `Customer` entitet og er knyttet til denne via `CustomerId` attributten. Hver oppføring av en tidsserieentitet i databasen representerer en diskret måleverdi knyttet til en bestemt `Sensor`, som igjen hører til en bestemt `IoTEdge`. `IoTEdge` entiten er igjen knyttet til en og bare en `Customer` entitet.

9.5.2.2 Service klasser

Service klassene i applikasjonen utfører CRUD operasjoner mot databasen, mapper entiteter mot DTOer og autentiserer brukere. Hensikten med service klassene er å forhindre at controllerne utfører for mye logikk. Vi ønsker å holde disse så *tynne* som mulig, og flytter derfor logikk og databasetilgang til egne service klasser. Dette tillater oss også å implementere disse klassene ved hjelp av C# sin støtte for generiske typer. Vi bruker også den innebygde støtten for Dependency Injection[35] i .NET Core for å binde opp riktig implementasjon av et service interface ved kjøretid. Dette tillater oss å skrive mer generiske og dermed også gjenbrukbare klasser, og det underletter enhetstesting av koden ved hjelp av mocking rammeverk.



Figur 9.15: Type Dependency Graf for service klasser

For å unngå å skrive en egen service klasse for hver eneste entitet i domenemodellen, har vi latt alle entiteter implementere en base-klasse: `BaseEntity` som har de identifiserende attributtene `Id` og `CustomerID`. Dette tillater oss å definere ett generelt interface `ICosmosService` (fig 9.16) med en generisk typeparameter `TEntity`.

```
C# ICosmosService.cs
1   using ...
3
4   namespace dbBackend.Data.Service.Contracts
5   {
        18 usages  5 inheritors  fredrikfrostad  1 exposing API
6       public interface ICosmosService<TEntity>
7       {
            4 usages  1 implementation  fredrikfrostad
8           Task<IEnumerable<TEntity>> GetItemsAsync(string query);
            6 usages  1 implementation  fredrikfrostad
9           Task<TEntity> GetItemAsync(string id, string partitionKey);
            5 usages  2 implementations  fredrikfrostad
10          Task<TEntity> AddItemAsync(TEntity item);
            4 usages  1 implementation  fredrikfrostad
11          Task<TEntity> UpdateItemAsync(string id, TEntity item);
            5 usages  1 implementation  fredrikfrostad
12          Task DeleteItemAsync(string id, string partitionKey);
            4 usages  1 implementation  fredrikfrostad
13          Task<List<object>> DeleteAllItemsByCustomerId(string customerId);
14      }
15  }
```

Figur 9.16: Generelt interface for interaksjon mot database

Dette interfacet implementeres av en konkret klasse `CosmosService`. Her definerer vi basetype for den generiske typerameteren og vi setter opp dependency injection gjennom klassens konstruktør (fig 9.17).

```
C# CosmosService.cs
1   using ...
10
11  namespace dbBackend.Data.Service.Impl
12  {
        7 usages  2 inheritors  fredrikfrostad
13      public class CosmosService<TEntity> : ICosmosService<TEntity> where TEntity : BaseEntity
14      {
15          protected readonly Container Container;
16
            7 usages  fredrikfrostad
17          public CosmosService(
18              CosmosClient cosmosClient,
19              string databaseName,
20              string containerName)
21          {
22              Container = cosmosClient.GetContainer(databaseName, containerName);
23          }
24
```

Figur 9.17: Klassesignatur og konstruktør for CosmosService

Vi bruke .NET Core sin innebyggede funksjonalitet for dependency injection til å injisere en implementasjon med korrekt typeparameter utifra hvilken entitet vi skal behandle ved kjøretid. Dette gjøres av `Startup` klassen som kalles når programmets Main metode bygger en vert ved oppstart (fig 9.18).

```
services.AddSingleton<IAuthenticationService>(new AuthenticationService(
    cosmosClient: InitCosmosClientAsync(Configuration.GetSection( key: "CosmosDbCustomer")).GetAwaiter().GetResult(),
    databaseName: "N00418-CosmosDB-Data",
    containerName: "Customer",
    new CosmosService<PasswordAuthenticationData>(
        cosmosClient: InitCosmosClientAsync(Configuration.GetSection( key: "CosmosDbCustomer")).GetAwaiter().GetResult(),
        databaseName: "N00418-CosmosDB-Data",
        containerName: "Customer")));
services.AddSingleton<ICosmosService<Customer>>(new CosmosService<Customer>(
    cosmosClient: InitCosmosClientAsync(Configuration.GetSection( key: "CosmosDbCustomer")).GetAwaiter().GetResult(),
    databaseName: "N00418-CosmosDB-Data",
    containerName: "Customer"));
services.AddSingleton<ICosmosService<User>>(new CosmosService<User>(
    cosmosClient: InitCosmosClientAsync(Configuration.GetSection( key: "CosmosDbCustomer")).GetAwaiter().GetResult(),
    databaseName: "N00418-CosmosDB-Data",
    containerName: "Customer"));
services.AddSingleton<IEdgeService<IoTEdge>>(new EdgeService(
    cosmosClient: InitCosmosClientAsync(Configuration.GetSection( key: "CosmosDbCustomer")).GetAwaiter().GetResult(),
    databaseName: "N00418-CosmosDB-Data",
    containerName: "Customer"));
services.AddSingleton<ISensorService<Sensor>>(new SensorService(
    cosmosClient: InitCosmosClientAsync(Configuration.GetSection( key: "CosmosDbCustomer")).GetAwaiter().GetResult(),
    databaseName: "N00418-CosmosDB-Data",
    containerName: "Customer"));
services.AddSingleton<ICosmosService<AssetNode>>(new CosmosService<AssetNode>(
    cosmosClient: InitCosmosClientAsync(Configuration.GetSection( key: "CosmosDbCustomer")).GetAwaiter().GetResult(),
    databaseName: "N00418-CosmosDB-Data",
    containerName: "Customer"));
services.AddSingleton<ICosmosTimeSeriesService<TimeSeriesBaseEntity>>(
    new CosmosTimeSeriesService<TimeSeriesBaseEntity>(
        cosmosClient: InitCosmosClientAsync(Configuration.GetSection( key: "CosmosDbTimeSeries")).GetAwaiter().GetResult(),
        databaseName: "N00418-CosmosDB-Data",
        containerName: "Data"));
```

Figur 9.18: Binding av implementasjon til interface og definering av typeparameter

Dersom det for en entitetstype i domenemodellen er behov for å utvide funksjonaliteten som tilbys gjennom den generelle klassen `CosmosService`, lar dette seg enkelt gjøre ved å følge det samme mønsteret for dependency injection. Vi ser på implementasjonen av interfacet `IEdgeService` som et eksempel. Vi har i applikasjonen behov for å hente ut `IoTEdge` entiteter basert på hvilken `CustomerId` de er knyttet til. Vi ønsker ikke å utvide interfacet `ICosmosService` med funksjonalitet som kun gjelder for en entitetstype. Dette løser vi på følgende måte:

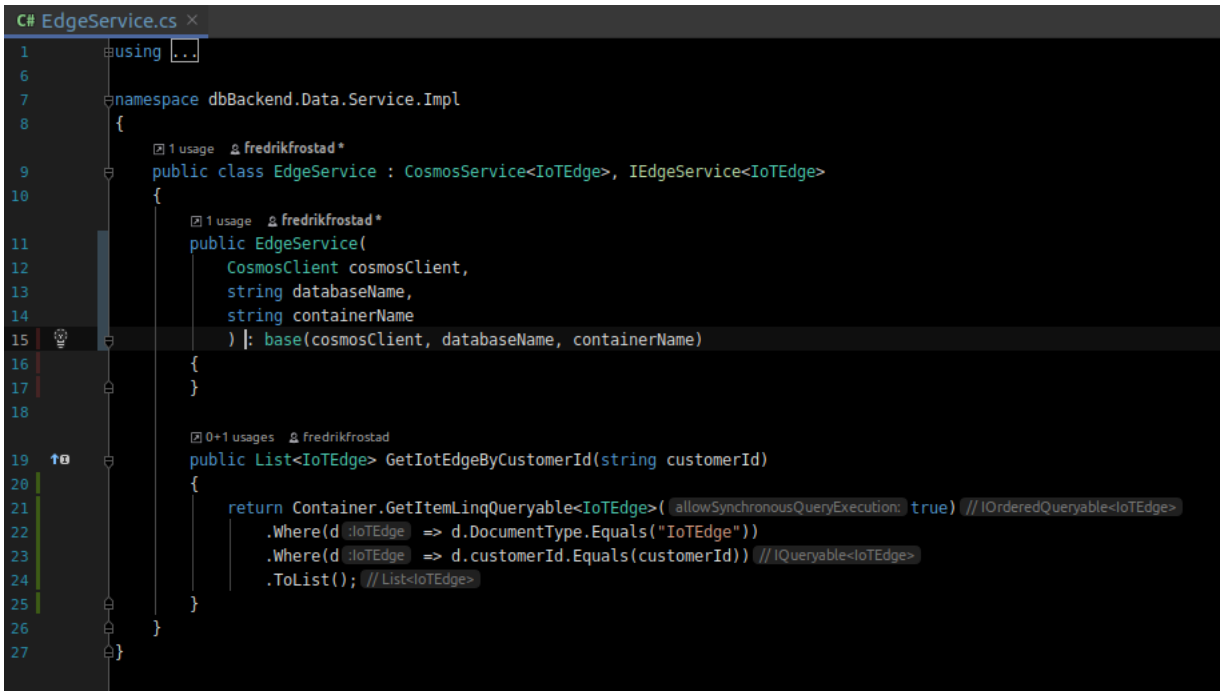
- Vi definerer ett nytt interface IEdgeService som implementerer det generelle interfacet `ICosmosService`:

C# IEdgeService.cs

```
1   using ...
3
4   namespace dbBackend.Data.Service.Contracts
5   {
        6 usages  1 inheritor  fredrikfrostad
6       public interface IEdgeService<TEntity> : ICosmosService<TEntity> where TEntity : IoTEdge
7       {
            1 usage  1 implementation  fredrikfrostad
8           public List<IoTEdge> GetIotEdgeByCustomerId(string customerId);
9       }
10  }
```

Figur 9.19: IEdgeService.cs

- Vi definerer en konkret klasse som implementerer både det nye interfacet `IEdgeService` og den konkrete implementasjonen av det generelle interfacet, `CosmosService`:

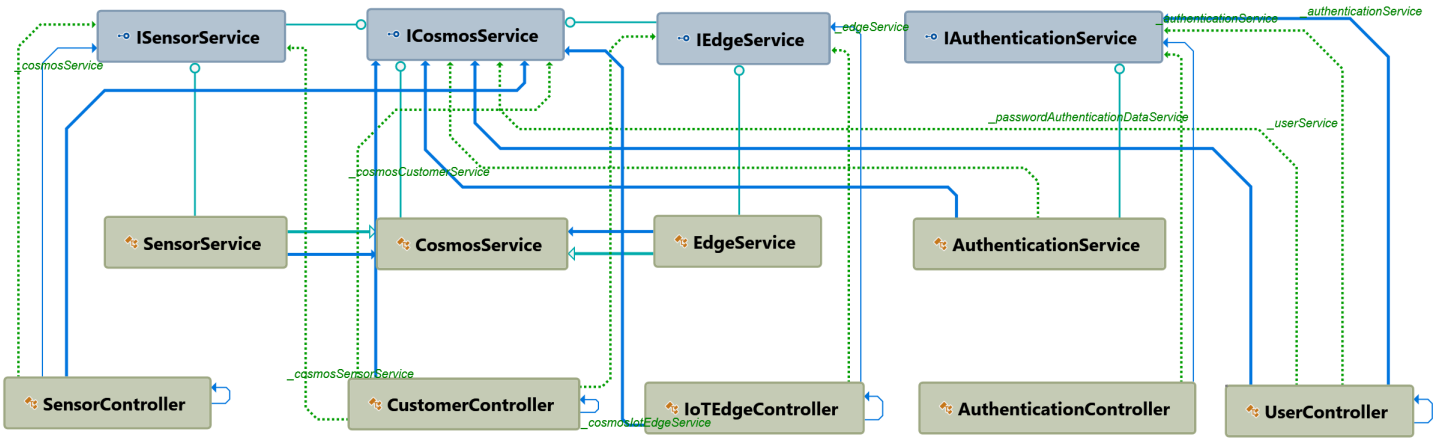


Figur 9.20: EdgeService.cs

- Vi oppnådd det vi ønsket; vi har lagt til ny funksjonalitet som kun er tilgjengelig for entiteten IoTEdge uten å bryte med dependency injection prinsippet. Interfacet bindes opp mot implementasjonen i applikasjonens `Startup` klasse (fig 9.18)

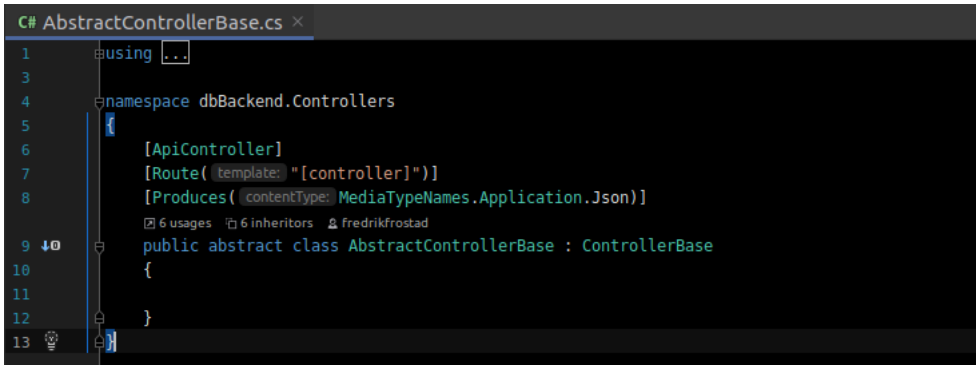
9.5.2.3 Controller klasser

Kontroller klassene definerer endepunktene i API-et som klienten sender HTTP forespørsler mot. Headeren i førespørselen vil først filteres av autentiserings mellomvaren (se 9.5.2.4) som sjekker at brukeren har tilgang til ressursen den spør etter. Avhengig av kallet som gjøres vil kontrolleren validere parametrene som sendes fra klienten, og videre gjøre operasjoner mot databasen via applikasjonens service klasser.



Figur 9.21: Avhengigheter mellom kontroller og service klasser

Dersom en forespørsel feiler sendes korrekt HTTP status tilbake til client med en beskrivelse av feilen. Kontrollerne i applikasjonen er av typen .NET Core Web API controller, som tilbyr annotasjons basert definering av ruting og HTTP-verb, samt direkte serialisering fra Entitet til JSON der dette er ønskelig. Alle endepunkter i applikasjonen returnerer JSON objekter og dette er satt på toppnivå med en abstrakt kontroller klasse som implementeres av alle andre kontrollerklasser (fig 9.22).



Figur 9.22: Abstrakt baseklasse for kontrollere

Figuren under, fig 9.23 viser et typisk endepunkt for en POST forespørsel. Denne metoden har ikke sin egen path, men er istedet mappet til den omsluttende klassens path. I dette tilfellet `.../User` Anotasjonen `[HttpPost]` definerer at det skal være en POST forespørsel. Metoden tar imot en JSON representasjon av en `User` entitet og validerer dennes felter. Dersom modellen ansees å være gyldig opprettes en ny bruker entitet og denne gis en GUID. Deretter sjekker metoden at det ikke allerede eksisterer en bruker med samme registrerte e-post adresse i databasen. Dersom denne valideringen også er gyldig persisteres den nye brukeren i databasen sammen med en autentiseringsentitet, `PasswordAuthenticationData`, som er knyttet til brukeren. Til slutt returneres en JSON representasjon av brukeren sammen med en HTTP-status 200 OK. Dersom valideringen feiler vil metoden returnere HTTP-status 400 BadRequest sammen med en feilbeskrivelse.

```
[HttpPost]
[ProducesResponseType( statusCode: StatusCodes.Status201Created)]
[ProducesResponseType( statusCode: StatusCodes.Status400BadRequest)]
fredrikfrostad *
public async Task<IActionResult> CreateUser(
    [Bind( params include: "customerId, alias")] User user, string password)
{
    User createdUser = null;
    if (ModelState.IsValid)
    {
        user.Id = Guid.NewGuid().ToString();
        if (_authenticationService.FindUserByUsername(user.Email) != null)
        {
            throw new WebappException( message: "User with email: " + user.Email + " already exists");
        }
        createdUser = await _userService.AddItemAsync(user);
    }

    if (createdUser != null)
    {
        if (_authenticationService.CreatePasswordAuthenticationData(user, password))
        {
            return Ok(createdUser);
        }
        else
        {
            await _userService.DeleteItemAsync(createdUser.Id,  partitionKey: createdUser.customerId);
        }
    }
    return BadRequest();
}
```

Figur 9.23: Eksempel på et typisk POST endepunkt (her fra `UserController`)

9.5.2.4 Autentisering og autorisering

Vi har valgt å bruke en forholdsvis enkel modell for autentisering og autorisering, der vi autoriserer en bruker basert på brukernavn og passord, og deretter returnerer en tidsbegrenset token (jwt) som klienten kan legge i header ved HTTP spørringer mot backend API-et for å autorisere bruker. Vi vil i de neste avsnittene se på flyten for henholdsvis autentisering og autorisering.

Autentisering betyr å gå god for, å bevise ekteheten av noe. I vårt system beviser bruker at han eller hun er den han eller hun utgir seg for å være ved å oppgi en gyldig kombinasjon av brukernavn og passord som unikt identifiserer brukeren. Vi skal nå se på flyten for autentisering og komponentene som inngår i denne prosessen.

En registrert bruker identifiseres av en kombinasjon av to entiteter som inngår i domenemodellen: `User` og `PasswordAuthenticationData`. `User` entiteten inneholder all informasjon om en bruker i systemet og har følgende attributter:

- string Alias
- string FirstName
- string Lastname
- string Email
- List<string> Roles

`PasswordAuthenticationData` entiteten inneholder kun informasjon som trengs for å autentisere brukeren i systemet og har følgende attributter:

- string userName
- string PasswordHash
- string PasswordSalt

Vi har i systemet valgt å benytte brukers epost-adresse som brukernavn, og krever derfor at alle brukere har en unik e-postaddresse. Dette sjekkes i `UserService` ved opprettelse av kunde. Ved opprettelse av kunde hashes brukers passord med HMAC-SHA512 [36] og resulterende hash og salt lagres i `PasswordAuthenticationData` entiteten, som knyttes til bruker via brukernavn.

Når en bruker skal autentiseres i systemet skjer følgende:

- Klient sender en HTTP forespørsel til login endepunkt i `AuthenticationController` (fig 9.24). Legg merke til at endepunktet er annotert med `[AllowAnonymous]`, dette gjøres fordi alle HTTP forespørsler fanges opp av autorisasjons-mellomvaren og avvises dersom token ikke er gyldig. På dette tidspunktet har ikke bruker mottatt et token, og vi krever derfor ikke autorisering av dette endepunktet.

```
[AllowAnonymous]
[HttpPost]
[Route( template: "login")]
fredrikfrostad
public IActionResult Login([FromBody] AuthenticationRequestDto authDto)
{
    var user = _authenticationService.Authenticate(authDto.Username, authDto.Password);

    if (user == null) return Unauthorized(new {message = "Username or password is incorrect"});

    var tokenHandler = new JwtSecurityTokenHandler();
    var key :byte[] = Encoding.ASCII.GetBytes(_appSettings.Secret);
    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(new[]
        {
            new Claim( type: ClaimTypes.Name, value: user.Email)
        }),
        Expires = DateTime.UtcNow.AddHours(4),
        SigningCredentials = new SigningCredentials(
            new SymmetricSecurityKey(key), algorithm: SecurityAlgorithms.HmacSha256Signature)
    };
    var token = tokenHandler.CreateToken(tokenDescriptor);
    var tokenString = tokenHandler.WriteToken(token);
    var authenticationResponseDto = _mapper.Map<AuthenticationResponseDto>(user);
    authenticationResponseDto.Token = tokenString;

    return Ok(authenticationResponseDto);
}
```

Figur 9.24: Login endepunkt fra `AuthenticationController`

- Innsendt passord og brukernavn autentiseres i `AuthenticationService` (fig 9.25, fig 9.26)

```
0+1 usages  fredrikfrostad
public User Authenticate(string userName, string password)
{
    if (string.IsNullOrEmpty(userName) || string.IsNullOrEmpty(password))
    {
        return null;
    }

    var user = FindUserByUsername(userName);
    var passwordAuthcData = FindPasswordAuthenticationData(user);

    if (!VerifyPasswordHash(password, passwordAuthcData.PasswordHash, passwordAuthcData.PasswordSalt))
    {
        return null;
    }

    return user;
}
```

Figur 9.25: Autentiseringmetode i `AuthenticationService`

```
1 usage  fredrikfrostad
private static bool VerifyPasswordHash(string password, byte[] storedHash, byte[] storedSalt)
{
    if (password == null) throw new ArgumentNullException( paramName: "password");
    if (string.IsNullOrWhiteSpace(password)) throw new ArgumentException( message: "Value cannot be empty or whitespace only string.", paramName: "password");
    if (storedHash.Length != 64) throw new ArgumentException( message: "Invalid length of password hash (64 bytes expected).", paramName: "passwordHash");
    if (storedSalt.Length != 128) throw new ArgumentException( message: "Invalid length of password salt (128 bytes expected).", paramName: "passwordHash");

    using var hmac = new System.Security.Cryptography.HMACSHA512(storedSalt);
    var computedHash :byte[] = hmac.ComputeHash( buffer: System.Text.Encoding.UTF8.GetBytes(password));
    return !computedHash.Where((t :byte, i :int) => t != storedHash[i]).Any();
}
```

Figur 9.26: Hjelpemetode for verifisering av passord mot lagret hash i `AuthenticationController`

- Dersom kombinasjonen av brukernavn og passord verifiseres av `AuthenticationService` sender `Login` metoden (fig 9.24) i `AuthenticationController` en 200-OK response til klient, sammen med en signert tidsbegrenset token som autoriserer brukeren i API-et sine øvrige endepunkter.

Autorisering Der autentisering handler om hvem brukeren er, går autorisering ut på å avgjøre hva brukeren har lov til å utføre, og hvilke ressurser brukeren har tilgang til i systemet. I vårt system utfører vi autorisering basert på et signert jason web-token som innehas av brukeren etter at denne er autentisert i systemet. Siden backend API-et i dette systemet kun er ment som et proof-of-concept for flyt av telemetridata i den overordnede systemarkitekturen, har vi ikke implementert en mer avansert rollebasert autorisering. Istedet har vi valgt en enkel modell der bruker er autorisert for alle ressurser i systemet, basert på det faktum at bruker innehar et gyldig token. Autorisjasjonen foregår ved at mellomvare i applikasjonens HTTP forespørsel pipeline fanger opp alle HTTP forespørslser som sendes til API-et, og deretter undersøker om headeren i forespørselen inneholder et gyldig token.

I en .NET Core applikasjon definerer vi mellomvare i applikasjonens `StartUp.cs` fil. Vi har valgt en bearer-strategi[37] når vi definerte vår mellomvare for autorisering (fig 9.27)

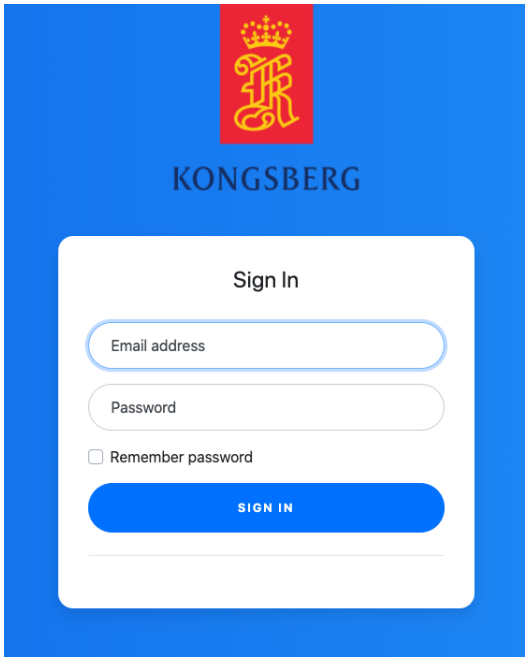
```
// Configure jwt authentication
var appsettings = appSettingsSection.Get<AppSettings>();
var key :byte[] = Encoding.ASCII.GetBytes(appsettings.Secret);

services.AddAuthentication( configureOptions: x :AuthenticationOptions =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(x :JwtBearerOptions =>
{
    x.Events = new JwtBearerEvents
    {
        OnTokenValidated = context =>
        {
            var authenticationService =
                context.HttpContext.RequestServices.GetRequiredService<IAuthenticationService>();
            var userId :string? = context.Principal.Identity.Name;
            var user = authenticationService.FindUserByUsername(userId);

            if (user == null) context.Fail( failureMessage: "Unauthorized");
            return Task.CompletedTask;
        }
    };
    x.RequireHttpsMetadata = false;
    x.SaveToken = true;
    x.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
```

Figur 9.27: Mellomware for autorisering av web-token

Vi kan enkelt bruke denne autorisasjons-mellomvaren i resten av applikasjonen ved å sette `app.UseAuthentication` og `App.useAuthorization` i `ConfigureServices` metoden i `StartUp.cs`. Alle endepunkter i API-et vil da autoriseres med vår valgte autorisasjonsstrategi såfremt vi ikke annoterer endepunktet med `[AllowAnonymous]` (9.24).

9.6 Frontend

Frontend applikasjonen er et minimalt brukbart produkt, hvor vi kan logge inn og visualisere data.



Figur 9.28: Login meny for frontend applikasjonen

9.6.1 Rammeverk og teknologier

Rammeverkene og teknologiene som er brukt på frontenden er delt inn i to deler: en visuell del og en serverside.

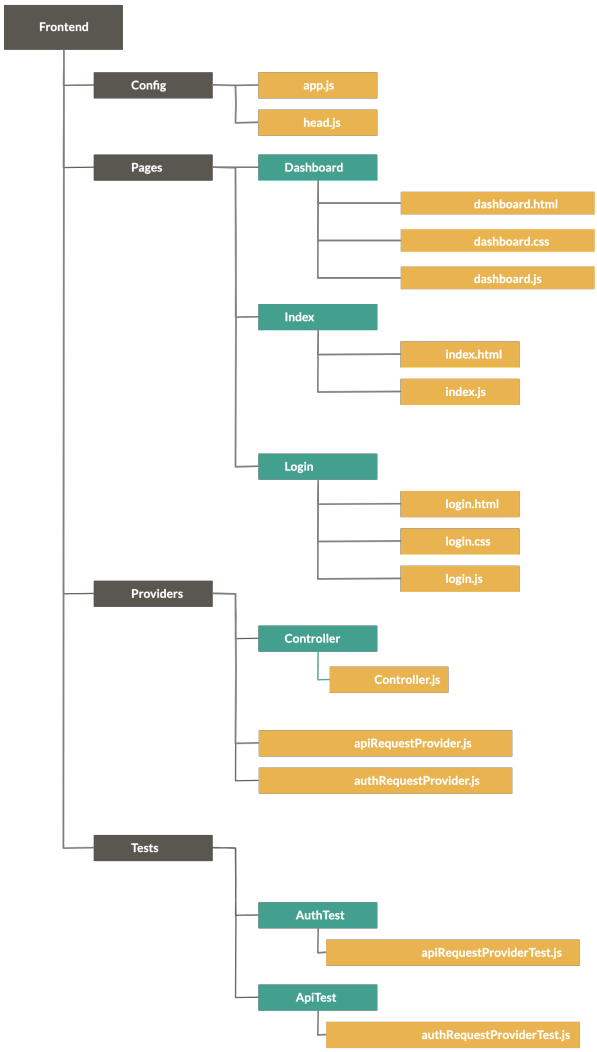
9.6.1.1 Server teknologier

Serversiden av frontend klienten er bygget med Node.js som er en JavaScript basert asynkron kjøretid som brukes til å bygge moderne web applikasjoner, og er lett skalerbar og vedlikeholdbar. For å gjøre HTTP spørringer mot backend API-et brukte vi Axios som er et JavaScript rammeverk for spørringskonfigurering og løfte-basert HTTP klient håndtering. Håndtering av informasjons-flyt gjennom de forskjellige nettsidene brukte vi CORS [38] som lar oss gjøre HTTP spørringer mot andre klienter. Express ble brukt som webserver teknologi for håndtering av all trafikk, routing og servering av nettsidene.

9.6.1.2 Visualiseringsteknologier

Vi bruker Bootstrap for å raskt sette opp et fungerende brukgrensesnitt. Datavisualisering var en sentral del av teknologivalgene vi gjorde på frontend applikasjonen. Ved å velge Time Series Insight sitt åpne kildekode bibliotek brukte vi en del av Microsofts tilbud for visualisering av data.

9.6.2 Arkitektur



Figur 9.29: Frontend filstruktur

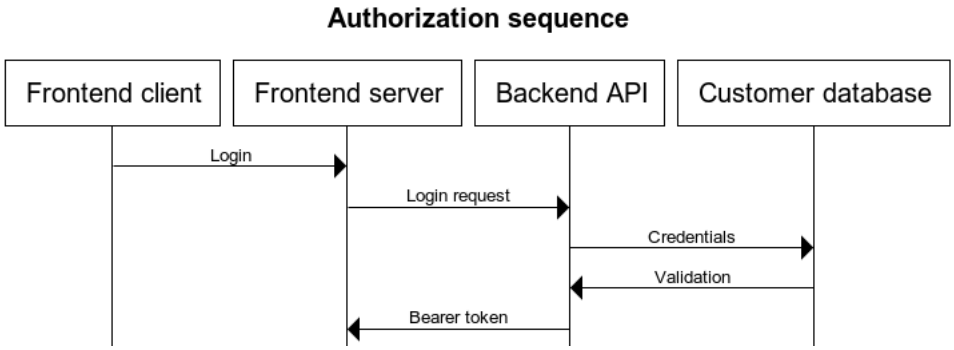
Arkitekturen på prosjektet følger en struktur hvor inndelingen går på felles tjenester i Config mappen og hvor håndtering av routing i app.js, importering og autentisering opp mot Microsofts Time Series Insights i head.js. Pages refererer til nettsider som applikasjonen har hvor arkitetkuren er lagt opp for å enkelt kunne legge til nye sider ved å opprette en ny mappe med de nødvendige filene. Providers håndterer utgående HTTP forespørsler, hvor inndelingen reflekterer hvilken rolle i applikasjonen denne filen har. Controller brukes for å kunne lettere holde kontroll på endepunkter, ved en utvidelse hvor trafikk ruting til spesifikke tjenester vil være en viktig del av den overordnede kompleksiteten i systemet.

9.6.3 Funksjonalitet

Frontend applikasjonen skal håndtere bruker login, autorisasjon, data transformering og visualisering.

9.6.3.1 Login og autorisasjon

Login funksjonaliteten i applikasjonen er laget for å simulere hvordan en sluttbruker vil kunne aksessere data og få den visualisert. Ved å bruke Bearer tokens som blir utstedt fra Backend API-et kan en bruker autentisere seg og kun aksessere data som brukeren er eier av.



Figur 9.30: Sekvens diagram av login sekvensen

Håndtering av autorisasjonstoken blir gjort av serveren på frontend applikasjonen, som sørger for at sekvensen i figur 9.30 ikke må gjennomføres for hver spørring mot backend API-et.

9.6.3.2 Data transformering og visualisering

Siden Time Series Insights client ble brukt for visualisering, måtte vi tilpasse datastrømmen til et format TSI kan håndtere. Dette gjorde at vi måtte bygge en parser som parset all informasjonen som kom fra databasen til det korrekte formatet.

9.6.4 Konfigurasjon og byggsystem

For all håndtering av start, build, testing og avhengigheter brukte vi Node package manager (NPM), som er et pakkehåndterings verktøy for JavaScript kjøretiden Node.js. Vi bruker Docker containere for å utplassere programmet i en app service som automatisk blir oppdatert og konfigurert etter en Git push til develop branchen 3.3.2.

9.7 Grafana

Grafana er et open source analyse og visualiserings program. Med Grafana kan en gjøre query på data, visualisere det, sette opp alarmer. Den tilbyr enkel integrasjon med time-series databaser, noe som gjorde at vi valgte å bruke Grafana i vårt system. Vi bruker Grafana til å gjøre spørringer mot PostgreSQL, som har TimescaleDB tillegget installert. Dette gjør at PostgreSQL databasen blir optimalisert for rask konsumering av data og komplekse spørringer. Vi bruker også Grafana til å se på ytelse på komponenter vi har i Azure igjennom Azure Monitoring. Måten vi har satt opp Grafana på, er igjennom en Docker container som vi har laget. Den kjører på en App Service med kontinuelig utvikling, slik at hvis det skjer en endring på Grafana konfigurasjonen, vil den automatisk kjøre ut den nyeste versjonen.

9.7.1 Konfigurasjon

Konfigurasjonsfilen er skrevet slik at alle i Kongsberg Digital (KDI) kan bruke sin Microsoft bruker til å logge seg inn gjennom Azure Active Directory (Azure AD). På denne måten opprettholder vi cybersikkerhetsretningslinjene til KDI med å bruke Azure AD der det er mulig. I tillegg er App Servicen innstillt på å håndheve HTTPS og TLS 1.2 tilkobling.

For at innstillingene for Grafana Dashboards og brukere skal være vedvarende, så blir alle innstillingen lagret i den samme PostgreSQL instansen som sensor dataen blir lagret, men i en egen database. Innstillingene for det er under [database] seksjonen i konfigurasjonsfilen.

Slik vi har satt opp Grafana, kjører den inne i en Docker container. Det er satt opp en `Dockerfile` som er basert på Grafana sin egen Docker container, men med små modifikasjoner slik at konfigurasjonene vi har gjort vil trå i kraft. Dette er gjort ved at konfigurasjonsfilen blir kopiert inn i containeren under byggeprosessen.

9.8 Azure Time Series Insights

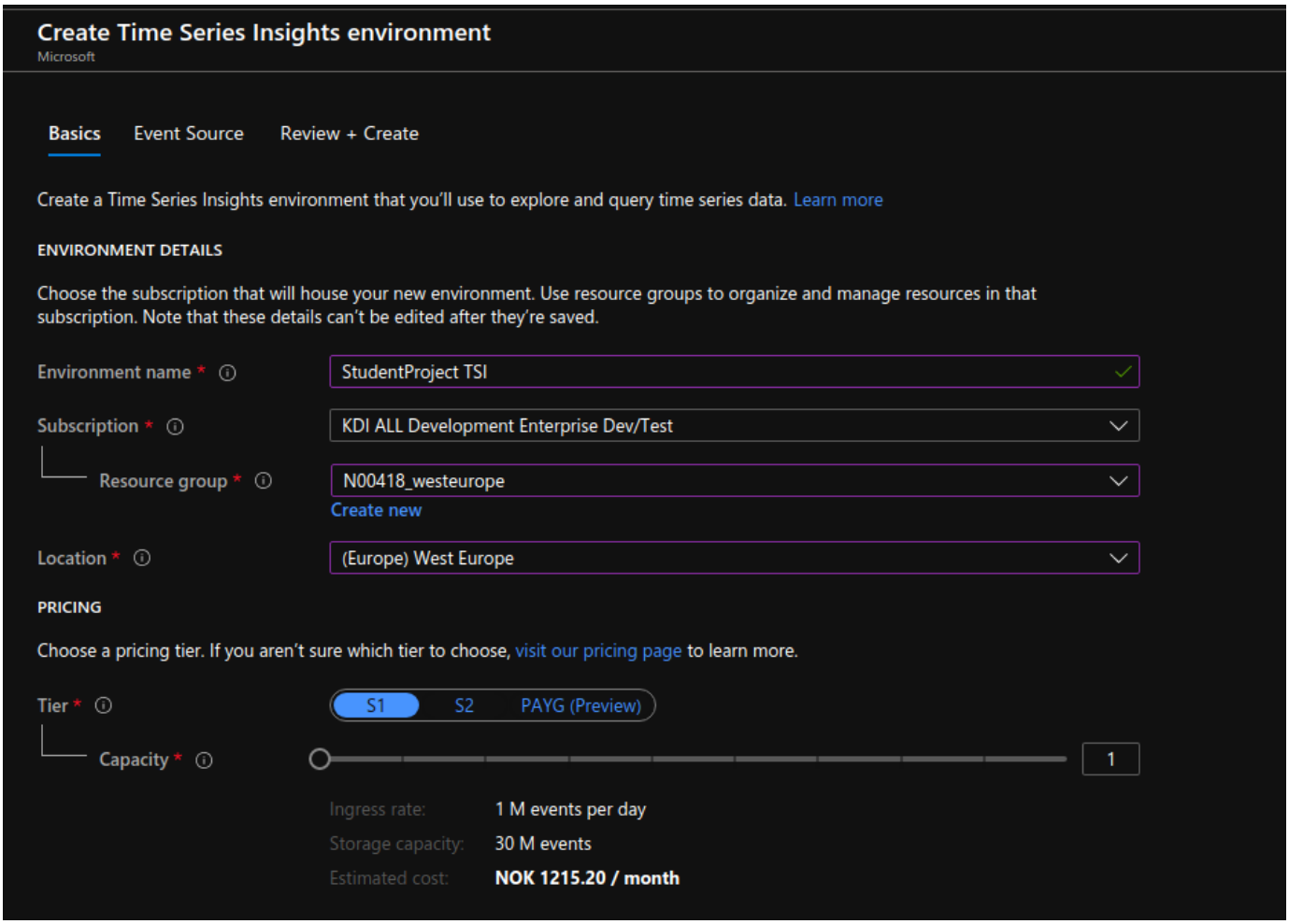
[39] Azure Times Series Insights er en fullt administrert, skybasert PaaS tjeneste fra Microsoft og er en fullstendig løsning for lagring, visualisering og henting av store mengder tidsseriedata. Time Series Insights tilbyr følgende funksjonalitet *rett ut av boksen*:

- Fullstendig integrasjon mot hendelseskilder som IoT Hub og Event Hub.
- Automatisk parsing av json data til en flat rad / kolonnestruktur
- Håndterer automatisk lagring av data. Data lagres på SSD disker og beholdes i opp til 400 dager
- Visualisering av data gjennom Time Series Insights Explorer
- Kan brukes som en backend tjeneste ved exponering av query API-er.

9.8.1 Konfigurasjon av Time Series Insights

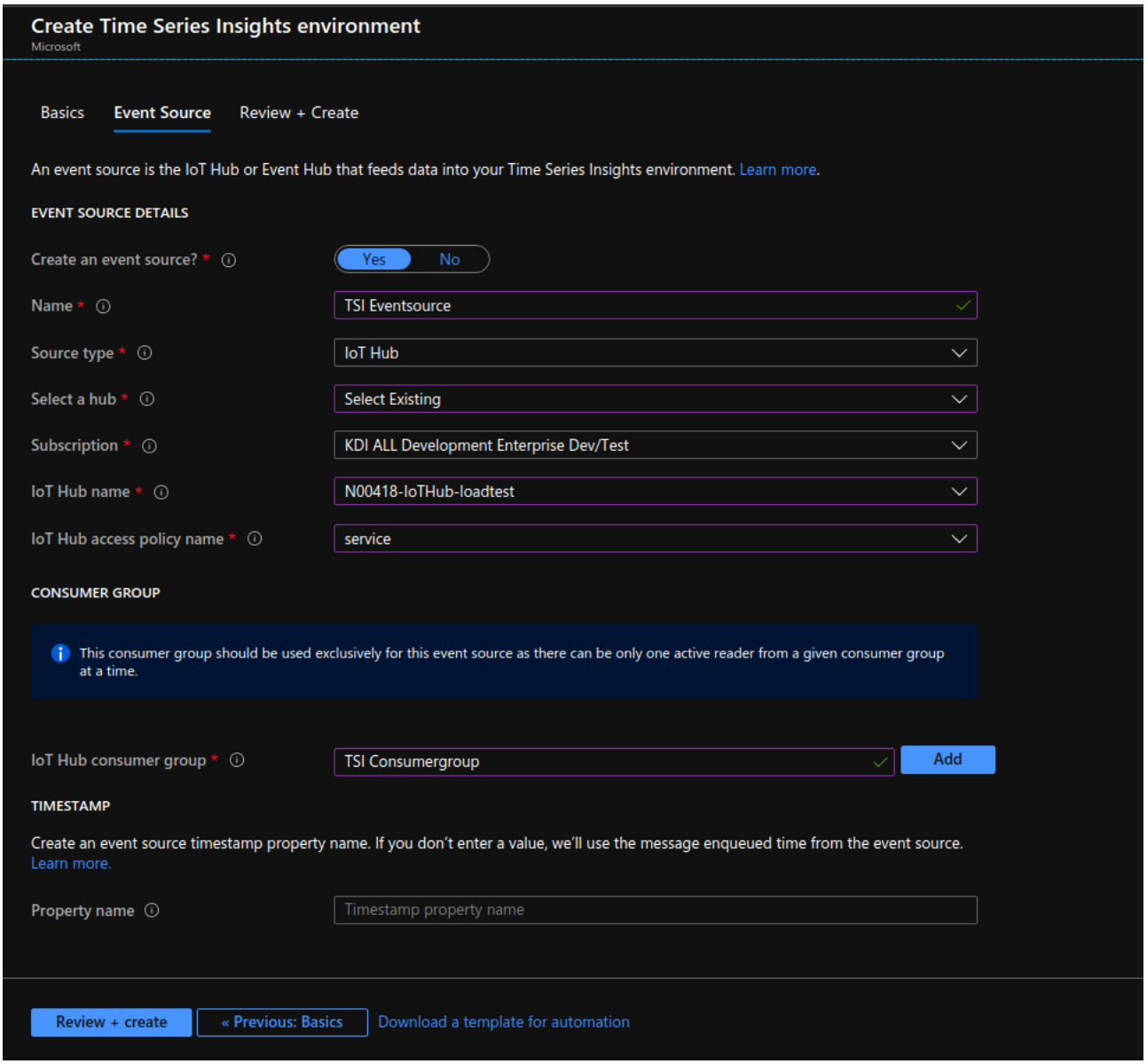
I dette avsnittet vil vi gå igjennom prosessen med å konfigurere Time Series Insights gjennom Azure Portal. Som følge av at time series insights er fullstendig integrert mot IoT Hub, er det svært enkelt å få et fungerende miljø opp å stå.

I første konfigurasjonssteg må vi gi miljøet et navn, knytte det til en konto med tilhørende ressursgruppe og velge hvilket nivå vi ønsker for ytelse/kost (fig 9.31).



Figur 9.31: Konfigurasjon av Time Series Insights steg 1

I neste steg må vi knytte en event source til Time Series Insights instansen. En event source er i vårt system en IoT Hub som sender telemetridata til Time Series Insights miljøet. I samme steg velger vi også en konsument gruppe som kan sees på som en data-tapning fra valgt event source (fig 9.32).

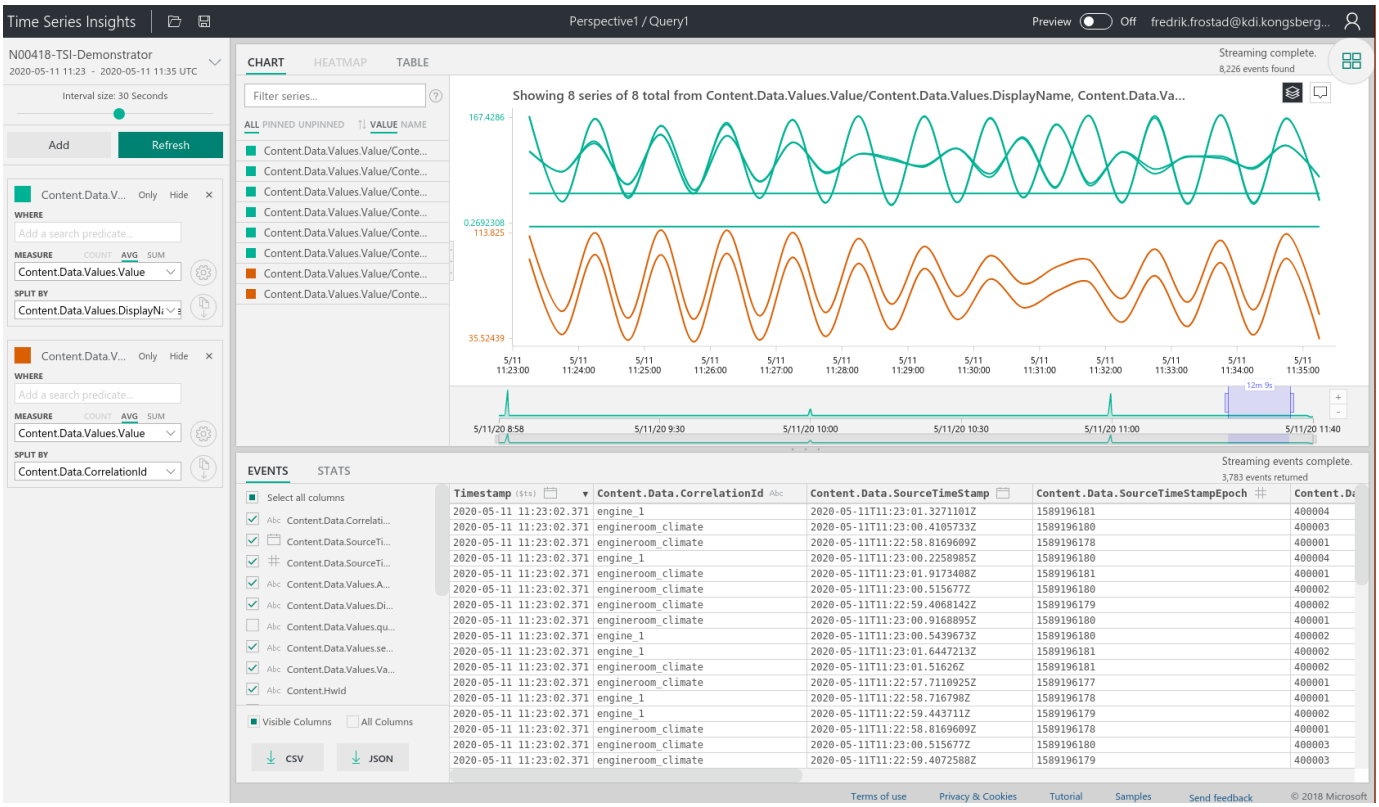


Figur 9.32: Konfigurasjon av Time Series Insights steg 2

I siste steg trenger vi bare å verifisere at informasjonen vi har lagt inn er korrekt, og med det er Time Series Insights Miljøet ferdig konfigurert og klart til bruk. Miljøet vil nå begynne å ta inn eventer fra IoT Hub, og disse kan direkte visualiseres gjennom Time Series Insights explorer, som er en inkludert web-basert frontend til miljøet.

9.8.2 Time Series Insight explorer

Time Series Insights explorer er en visualiserings-tjeneste som er inkludert med Time Series Insights. Denne tjenesten tillater spørring og visualisering på tvers av all telemetridata som er lagret i miljøet. For å opprettholde cybersikkerhetsretningslinjene til Kongsberg Digital, har vi konfigurert tilgangskontroll. Dette er gjort ved at brukeren må ha en Azure Active Directory bruker, i tillegg til å bli gitt eksplisitt tilgang til miljøet.



Figur 9.33: Visning av sensordata i Time Series Insights

Figur 9.33 viser et skjermbilde fra utforskeren. I dette skjermbildet har viser vi simulerte sensordata fra to virtuelle lokasjoner lagt over hverandre. I tillegg til å vise grafer som representerer datene, viser utforskeren også det underliggende datagrunnlaget som en tabell.

9.8.3 Kostnadsmodell

Modellen for kostnad i Time Series Insights er basert på et konsept som Microsoft kaller events. En event er definert som følgende[40]:

- En event er en enkelt dataenhet (sensoravlesning) med et tidsstempel
- Eventer beregnes i 1 KB blokker, altså er 0,8Kb = 1 event og 1.2 KB = 2 eventer
- Maksimal størrelse for en enkelt event er 32 KB

Tjenesten er tilgjengelig i flere forskjellige kostnadsnivåer som differensieres på antall eventer som kan håndteres pr døgn, hvor mange eventer som kan lagres pr Time Series Insights instans og hvor lenge data lagres.

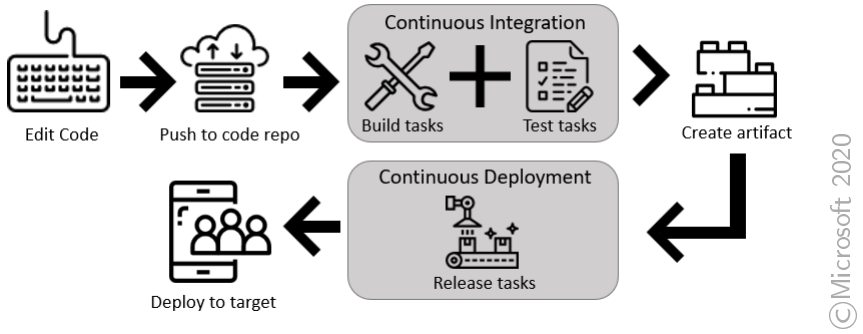
	s1	s2
Lagring pr instans	30GB eller 30 millioner eventer	300GB eller 300 millioner eventer
Daglig intak pr instans	1 GB eller 1 millioner eventer	10 GB eller 10 millioner eventer
Maks lagringslengde	Opp til 13 måneder	Opp til 13 måneder
Antall spørringer	Ubegrenset	Ubegrenset
Maks antall instanser	10	10
Pris pr instans pr måned	$150	$1.350

9.9 Byggsystem

Tidlig i prosessen med planlegging av oppgaven ble det fra oppdragsgivers side ytret ønske om at vi i den grad det lot seg gjøre, skulle benytte oss av DevOps prinsipper i utviklingen av proof-of-concept systemet. Funksjonalitet for å gjøre dette tilbys som en service gjennom Azure DevOps platformen og integrasjonen denne tilbyr mellom Azure Pipelines og Azure Repos. I de neste avsnittene vil vi gjøre rede for hvordan vi har konfigurert pipelines for automatisert bygging, testing og utplassering av kode i vårt system.

9.9.1 Azure Pipelines

Det første spørsmålet som må besvares er følgende; hva er Azure Pipelines? Microsoft definerer det slik: Azure Pipelines er en skytjeneste som kombinerer kontinuerlig integrasjon og kontinuerlig utplassering av kode for å kontinuerlig og konsistent teste, bygge og utplassere kode til enhver måldestinasjon[41].



Figur 9.34: Konseptuelt skjema for bygg og release pipeline.

Azure DevOps tilbyr to metoder for å konfigurere pipelines. Den best dokumenterte metoden er via en grafisk editor kalt *Classic Interface*, men det er også støtte for konfigurasjon via en yaml fil som sjekkes in i Git repoet til prosjektet. Bruken av yaml filer for å definere bygg og release pipelines har en stor fordel i det at man får versjonering av endringer gjort i bygg og release konfigurasjonen. Til tross for at vi anser dette som et betydelig fortrinn, førte tilgjengeligheten på god dokumentasjon og intuitiviteten i brukergrensesnittet til at vi valgte å bruke classic editoren for å konfigurere våre pipelines. Det er også verdt å ta i betraktning at det er mulig å eksportere de ulike oppgavene som defineres via classic editoren som yaml segmenter, slik at en migrering til yaml basert konfigurasjon er mulig på et senere tidspunkt.

Vi har i dette prosjektet definert bygg og release pipelines for følgende komponenter:

- Frontend applikasjon
- Backend applikasjon
- IoT Edge enheter
- Function app for datatransformasjon

Vi vil i de neste avsnittene gå igjennom konfigurasjonen av bygg og release pipelines for en IoT Edge enhet med Modbus og Message Processing modul.

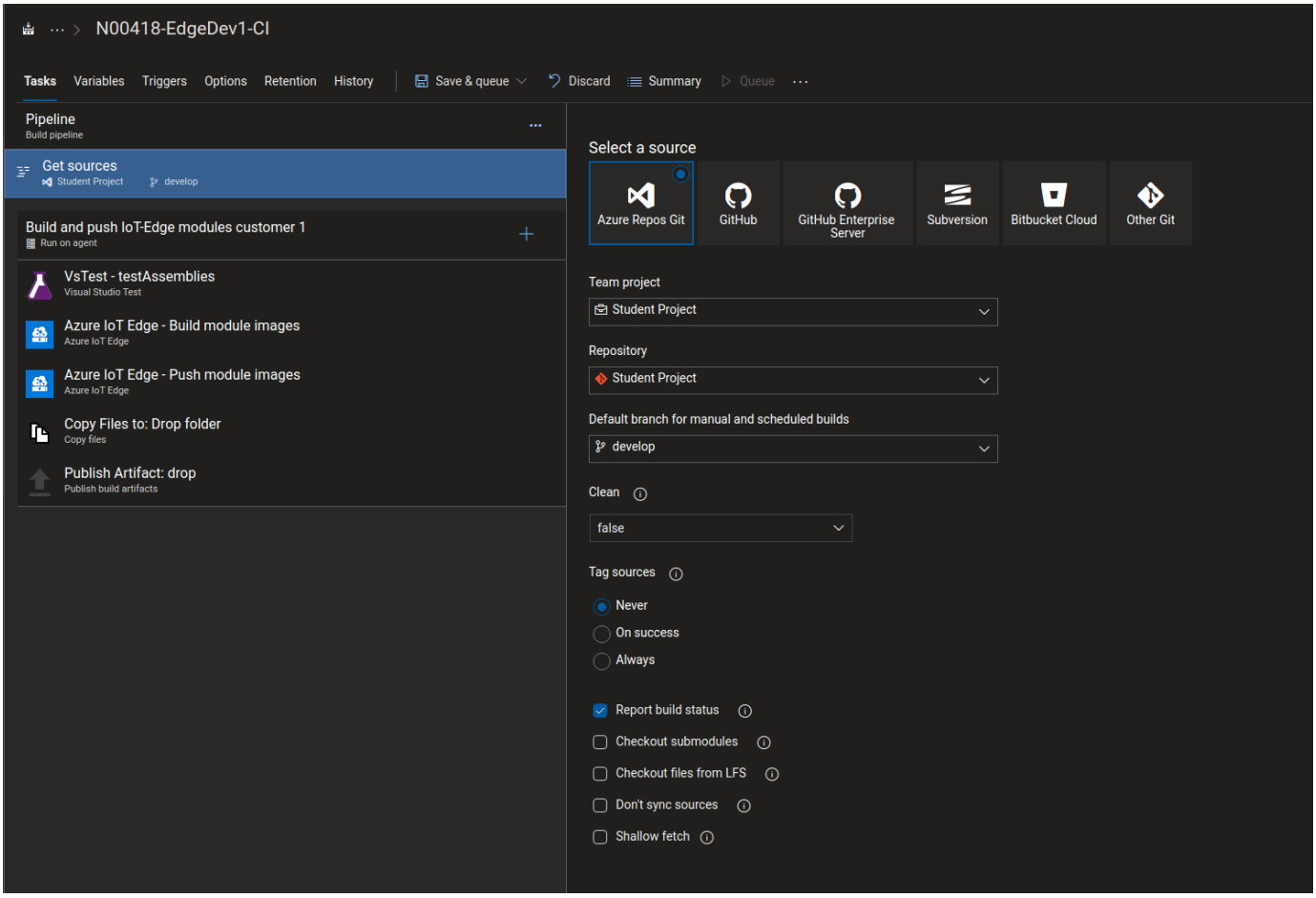
9.9.2 Bygg pipelines - Kontinuerlig integrasjon

Denne pipelinen er ansvarlig for å bygge nye artifakter for en komponent hver gang det pushes en endring til en forhåndsdefinert gren i et repository. Siden vi i dette prosjektet ikke skal produksjonsette noe kode valgte vi develop grenen i prosjektets Git repository som trigger gren for bygg pipelinene. Det er verdt å nevne at GitFlow arbiedsflyten er svært godt egnet for bruk i prosjekter hvor man benytter seg av kontinuerlig integrasjon og utplassering

9.9.2.1 Oppgaver som inngår i en bygg pipeline

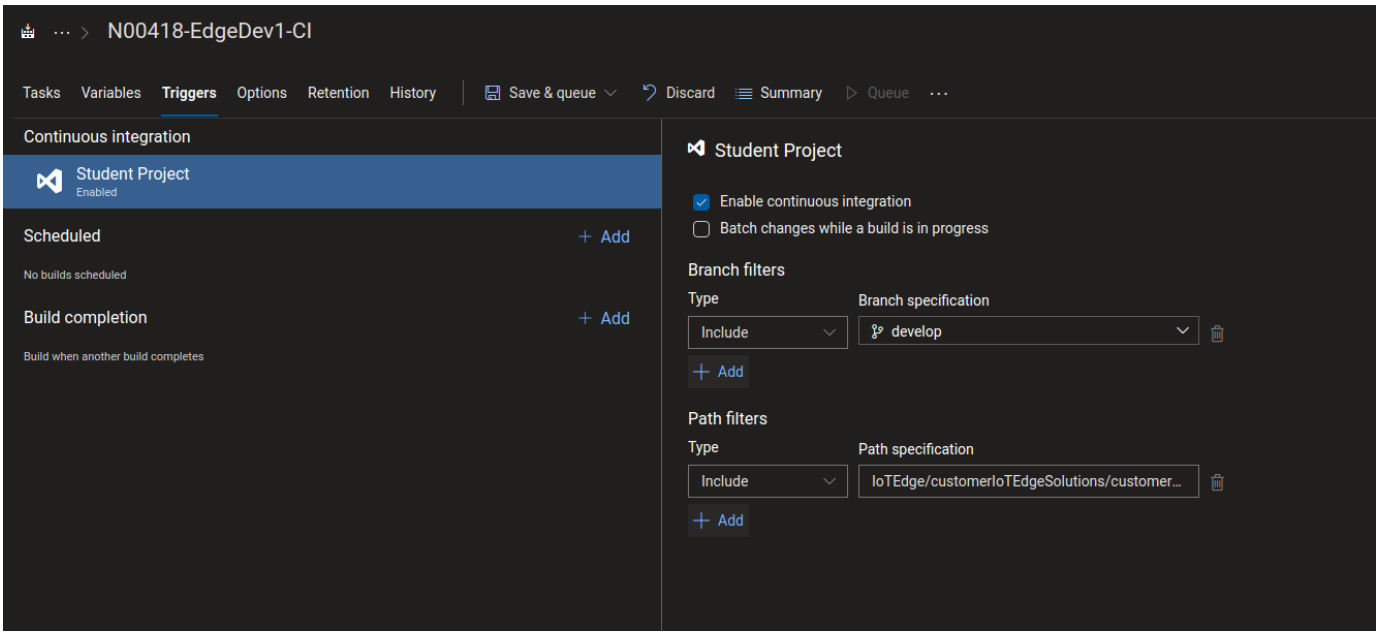
Videre i dette avsnittet vil vi gå gjennom de viktigste momentene ved oppsett av en pipeline for kontinuerlig integrasjon.

Det første trinnet i konfigurasjonen av en pipeline er å velge hva som er kilde for kodebasen som skal bygges (fig 9.35). Som tidligere nevnt bruker vi i dette prosjektet Azure repos som er tett integrert mot Azure pipelines.



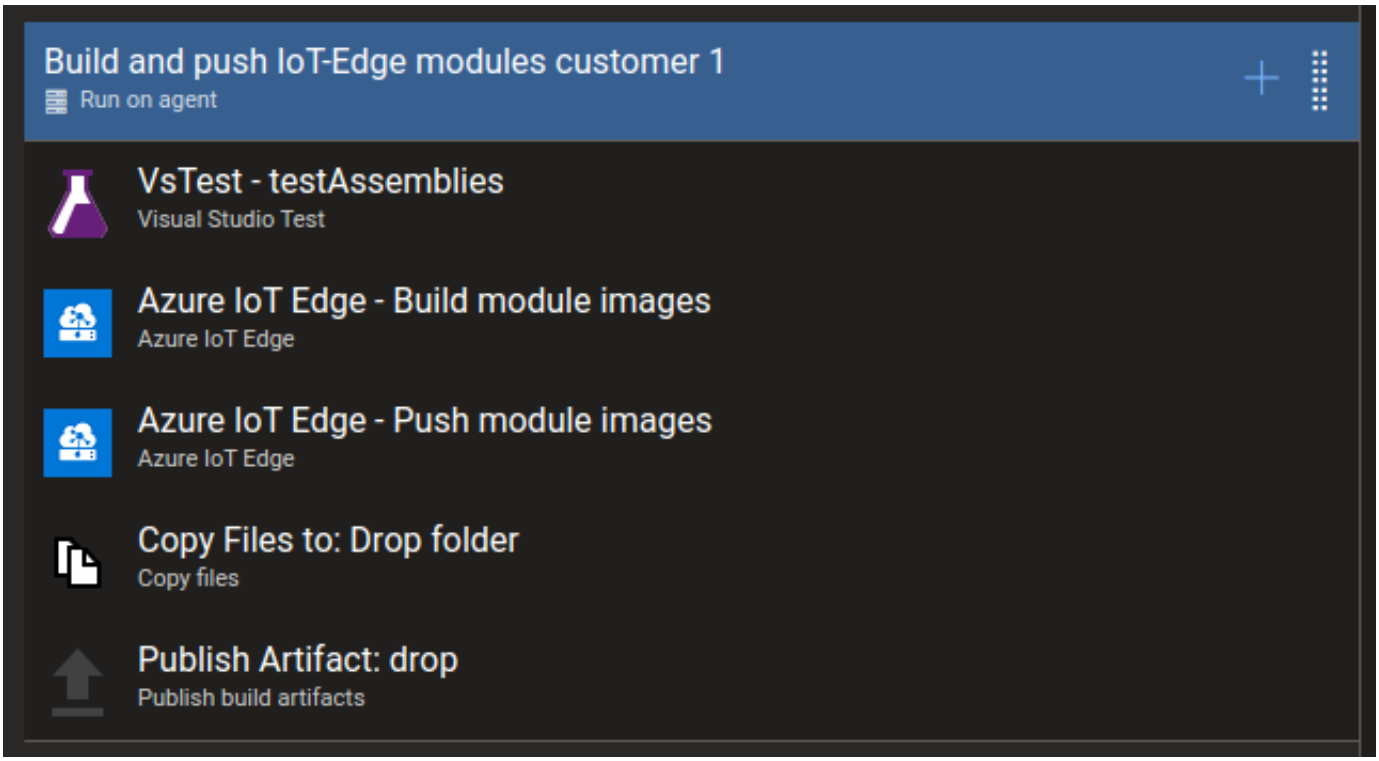
Figur 9.35: Konfigurasjon av kilde for kodebase i Azure pipeline

Videre ønsker vi å unngå at det igangsettes et bygg hver eneste gang noen fletter inn kode i develop grenen. Vi ønsker bare at denne pipelinen bygger når det gjøres endringer i kodebasen som tillhører IoT Edge enheten som denne pipelinen bygger. Dette oppnår vi ved å definere en trigger som kun starter et bygg dersom det gjøres endringer i den mappen i repoet som inneholder kode relevant for IoT Edge enheten som skal bygges (fig 9.36). Vi konfigurerer dette ved å definere ett filsti filter under fanen triggers i editoren. I dette tilfellet sier vi at filteret skal lytte på filstien `IoTEdge/customerIoTEdgeSolutions/customer1_IoTEdgeSolution/*` i repoet. Dette filsti-filteret lytter på endringer i en vilken som helst fil i målmappen. Det er også mulig å sette trigger filtere på individuelle filer dersom dette er ønskelig.



Figur 9.36: Konfigurasjon av trigger for igangsetting av byggprosess

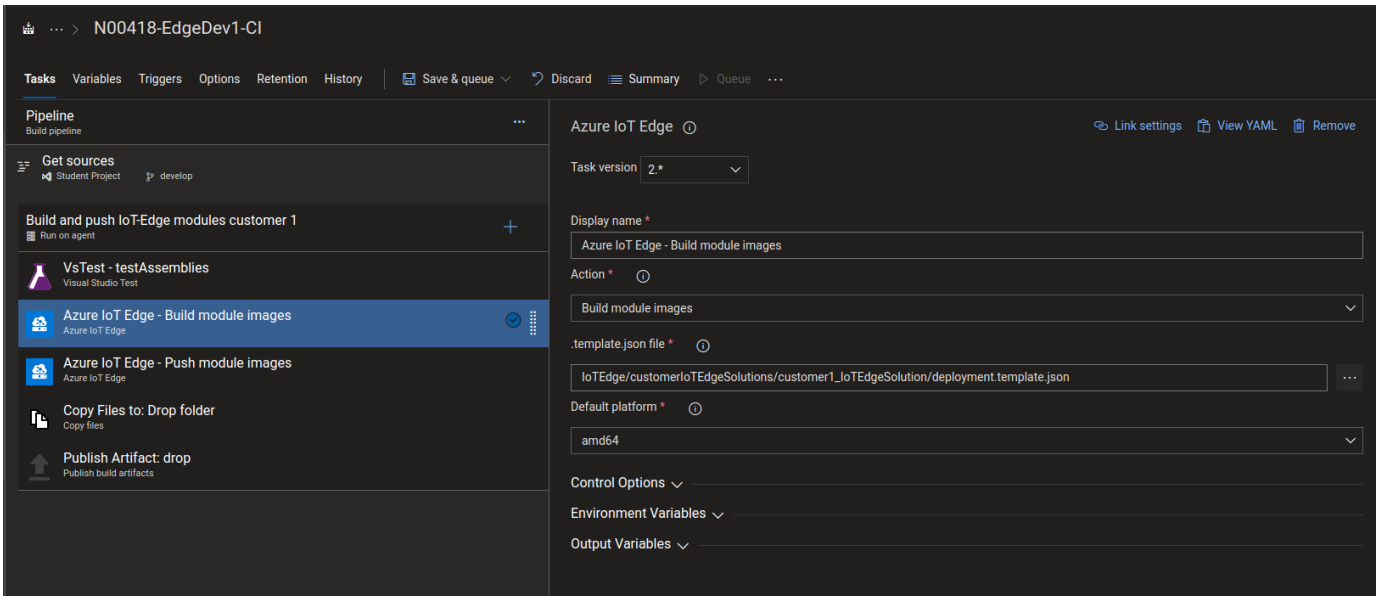
Når en trigger blir aktivert som følge av at det flettes ny kode inn i repositoryets valgte gren, settes det i gang en jobb som bygger en ny artifakt i pipelinen. I eksempelpipelinen vi ser på i dette avsnittet består denne jobben av 5 separate oppgaver (tasks) (fig 9.37), som utføres sekvensielt. Videre vil vi gå igjennom hver av disse oppgavene.



Figur 9.37: Her ser vi jobben `Build and push IoT Edge modules customer 1` med tilhørende oppgaver

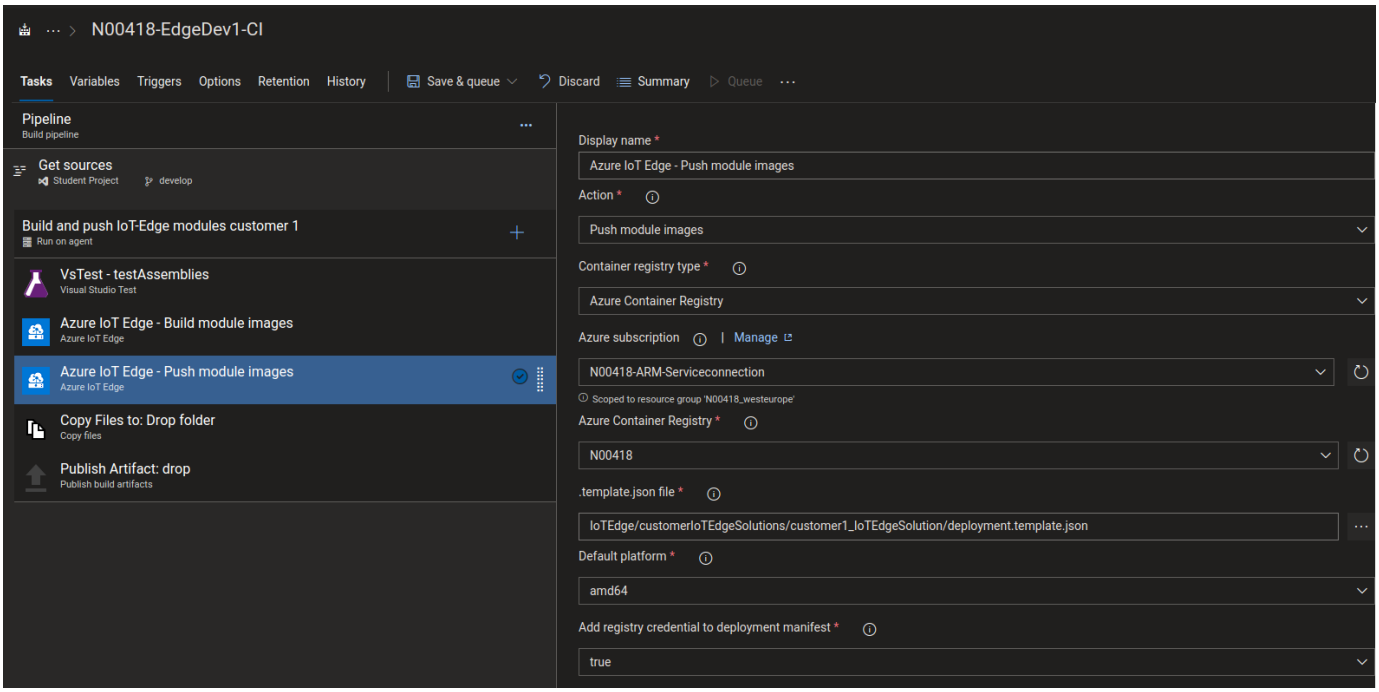
VsTest - testAssemblies er en oppgave som automatisk kjører alle tester som er konfigurert i prosjektet. Dersom noen av testene feiler, vil byggprosessen stoppe og få status failed. Denne oppgaven bruker Visual Studios testrunner for å kjøre testene, og er kompatibel med alle testrammeverk som har en Visual Studio test adapter. Dette er tilfelle for vårt valget C# enhetstestrammeverk nUnit.

Azure IoT Edge - Build module images er en oppgave som bygger Docker images med modulene som skal kjøre på IoT Edge enheten pipelinen bygger artifakter til. I denne oppgaven konfigurerer vi hvilken konfigurasjonsfil (deployment template) som ligger til grunn for bygget, og hvilken platform (cpu arkitektur) avbildningen bygges for (fig 9.38).



Figur 9.38: Oppgave som bygger Docker images for valgte IoT Edge moduler

Azure IoT Edge - push module images tar Docker avbildningene som ble bygget av forrige oppgave og pusher disse til prosjektets private avbildnings repository (fig 9.39). Dette tilgjengeliggjør avbildningene for alle påfølgende oppgaver som avhenger av disse.



Figur 9.39: Oppgave som pusher Docker images til prosjektets avbildnings repository

Copy Files to: Drop folder er en enkel oppgave som kopierer alle filer fra byggprosessen til en drop mappe som vil publiseres av den påfølgende oppgaven.

Publish Artifact: drop er den siste oppgaven i vår eksempel pipeline. Den publiserer alle artifakter som er blitt generert i byggprosessen til et Azure stagingdirectory (en ikke-varig lagringslokasjon i Azure) slik at disse artifaktene er tilgjengelige for påfølgende Release pipeline.

9.9.3 Release pipelines - Kontinuerlig utplassering

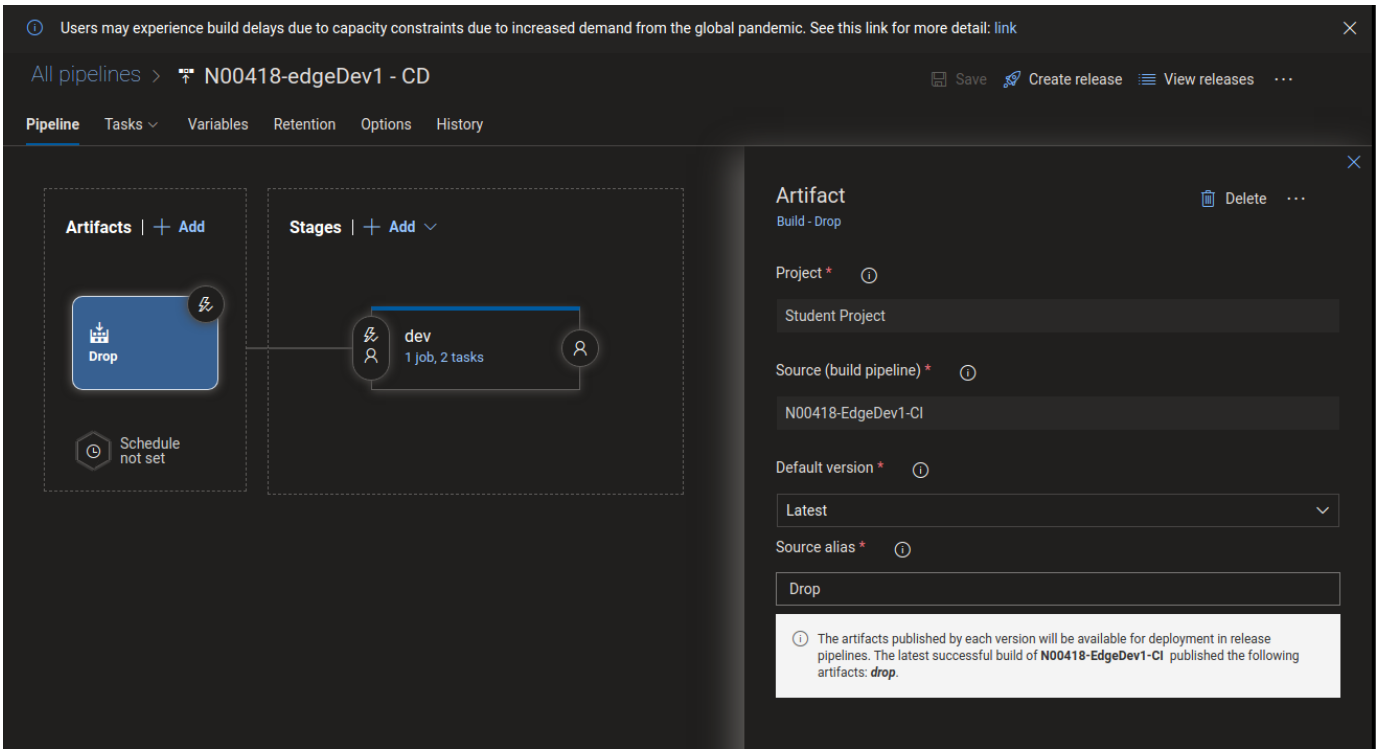
En release pipeline spesifiserer en sammensetining av artifakter, og/eller Docker images som er resultatet av en vellykket kjøring av en tilhørende bygg pipeline. Vi vil i dette avsnittet fortsette å se på prosessen for kontinurlig integrasjon / kontinuerlig utplassering av endringer i kodebasen for en IoT Edge enhet.

Når byggeprosessen beskrevet i forrige delkapittel kontinuerlig integrasjon har fullført med status vellykket vil en tilhørende release pipeline starte opp og sørge for at de bygde artifaktene og avbildningene automatisk utplasseres i et produksjons, eller i vårt tilfelle, utviklingsmiljø. Vi vil på samme måte som i forrige delkapittel gå igjennom konfigurasjonen av de forskjellige stegene som inngår i release pipelinen og hvilke oppgaver disse utfører.

9.9.3.1 Oppgaver som inngår i en release pipeline

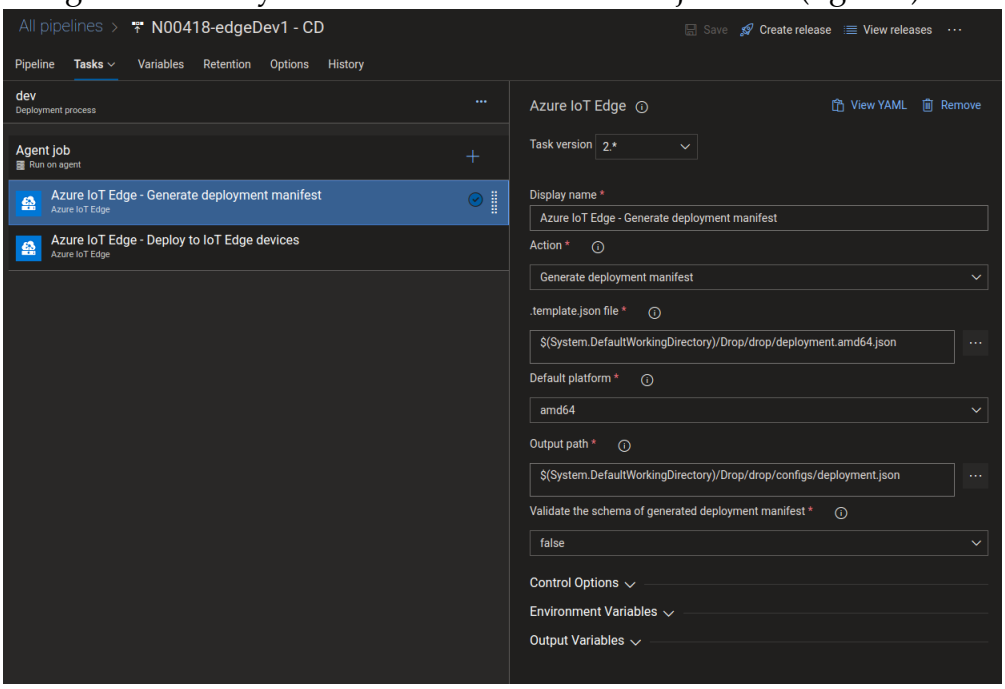
En release pipeline består av en eller flere artifakter og en eller flere stages. En artifakt er ferdigbyggede komponenter fra bygg pipelinen, og en stage inneholder oppgaver som må utføres for at de bygde artifaktene kan utplasseres.

Artifakter. Vi kan definere flere lokasjoner hvor vi ønsker at pipelinen skal hente artifakter. I eksempelpipelinen vi ser på her er det bare en lokasjon som er interessant, nemlig drop mappen beskrevet i forrige delkapittel. Det er verdt å merke seg at det er i denne delen av konfigurasjonen vi knytter release pipelinen opp mot bygg pipelinen.

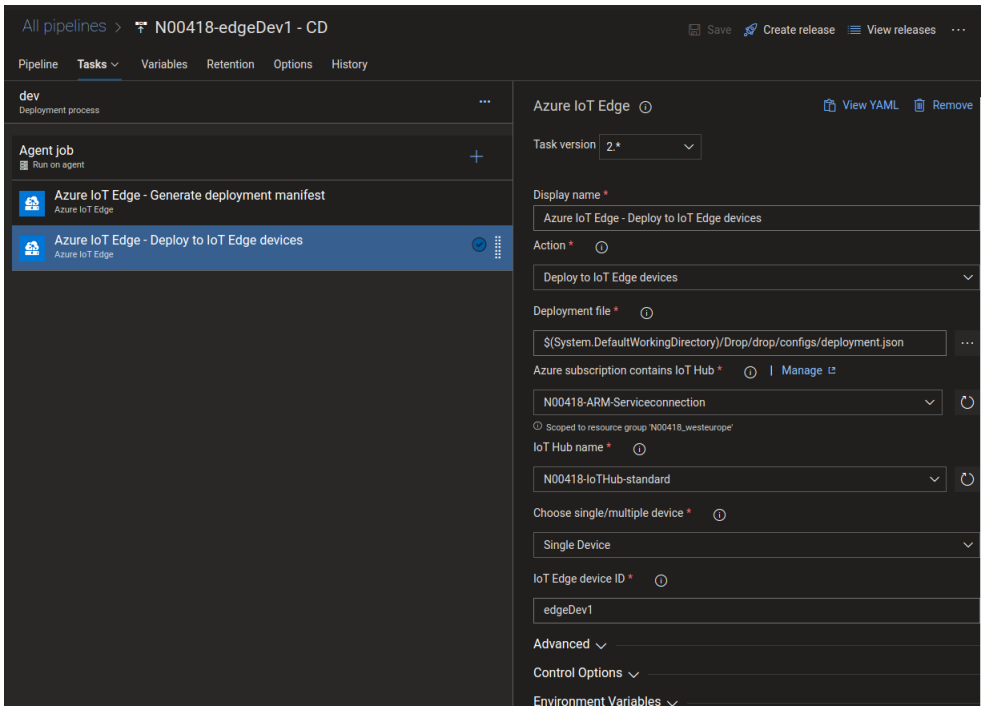


Figur 9.40: Konfigurasjon av lokasjon for å hente artifakter for utplassering

Stages. Vi har i denne pipelinen definert en stage som inneholder to oppgaver. Den første av disse (fig: 9.41), genererer et deployment manifest som brukes av neste oppgave for å konfigurere tilhørende IoT Edge enhet tilknyttet den tilkoblede IoT Hub tjenesten (fig 9.42).



Figur 9.41: Release-oppgave som genererer et deployment manifest for IoT Edge



Figur 9.42: Utplasserings-oppgave som pusher konfigurasjon til Iot Edge

Kapittel 10

Teknologier, verktøy og rammeverk

Gjennom arbeidet med prosjektet har vi benyttet en rekke verktøy og teknologier. Vi vil i dette kapitelet presentere de mest sentrale teknologiene som er benyttet i systemet, samt verktøyene vi har brukt.

10.1 Programmeringsspråk

C# [42] er et objekt-orientert, typesikkert programmeringsspråk med røtter i C. C# har mange likheter med C, C++ og Java. C# tilbyr innebygget minnehåndtering og avvikshåndtering, samt et enhetlig typesystem. Kongsberg Digital benytter språket internt i sine utviklingsprosjekter, og ønsker derfor at C# skal benyttes i systemet.

JavaScript ES6 er et objekt-orientert, høynivå dynamisk programmeringsspråk. JavaScript er en av kjerneteknologiene bak det moderne internet.[43, p. 1] Et overveldende flertall av moderne nettsider benytter JavaScript for klientside funksjonalitet og alle moderne webbrowsere støtter språket. Vi har i dette prosjektet brukt JavaScript for å implementere frontend applikasjonen for visualisering av data.

Python [44] er et dynamisk objekt-orientert, høynivå programmerings og scripte-språk bygget for hurtig utvikling. Python er et språk i hurtig vekst, og språket benyttes i undervisning på flere norske universiteter. Vi har i denne oppgaven brukt python til scripting av blant annet Modbus-simulator og generering av JSON strukturer for konfigurasjon av IoT Edge.

10.2 Rammeverk

.NET Core [45]er en open-source utviklingsplatform for Windows, macOS og Linux. .NET Core tilbyr rammeverk for blant cloud, IoT, webutvikling, klientside GUI med mer. I vårt system har vi benyttet .NET Core for å skrive en backend tjeneste som eksponerer data fra Cosmos DB gjennom REST API-er.

nUnit [46] er et enhetstesting rammeverk for .NET. NUnit er et open-source prosjekt, og er et av de mest populære rammeverkene for enhetstesting av .NET. Grunnet Nunits store og aktive brukerbase eksisterer det store mengder av dokumentasjon og informasjon rundt bruken av dette rammeverket. I vårt system har vi benyttet oss av nUnit for å skrive enhetstester i backend-applikasjonen.

Moq [47]er et populært mocking bibliotek for .NET som er utviklet for å benytte seg av .NET Linq og lambda uttrykk. Dette underletter bruken av biblioteket i enhetstesting. Vi har benyttet Moq sammen med NUnit for enhetstesting av backend applikasjon.

Node.js [49] er en asynkron event-drevet JavasScript kjøretid, utviklet for å konstruere skalerbare nettverksapplikasjoner. Node.js er en videreutvikling av JavaScript fra ren klientside kode til et serverside scripte språk. Node.js sitt pakkehåndteringssystem, NPM er et av verdens største økosystemer[49] for åpen kildekode.

DotCover [48] er et .NET unit testframework og kodedekningsverktøy som integreres med Visual Studio og JetBrains Rider.

express

Express [50]er et backend rammeverk for utvikling av web applikasjoner basert på Node.js. Express har blitt omtalt som den defakto standaren når det kommer til mobil og API utvikling med Node.js.

TSI-Client [51] Time series insight JavaScript SDK (TSI-Client) er et JavaScript biblotek basert på Microsoft azure Time series insight som inneholder visualisering og analyse komponenter. Disse komponentene kan enten brukes mot Time Series Insights API-et eller egen data som transformeres til et spesielt flatt format.

Mocha [52] er et test rammeverk for JavaScript som kjører både på Node.js og i browser. Vi har benyttet Mocha til enhetstesting av vår front-end applikasjon.

Chai [53] er et JavaScript bibliotek brukt til å lage mer komplekse påstander for testfunksjoner i en applikasjon. Dette biblioteket er en ekstra utvidelse brukt sammen med test rammeverk.

Istanbul [54] er et testdekningsbibliotek brukt for å sjekke testdekning i en applikasjon. Biblioteket gjør linje for linje analyse for at man til enhver tid skal kunne se hvilke linjer som ikke er testet.

Bootstrap [55] er et åpen kildekode rammeverk for å bygge responsive nettsider. Det tar i bruk JavaScript og CSS-baserte grensesnittkomponenter for å kunne raskt og enkelt kunne bygge en nettside.

10.3 Teknologiplatformer

Docker [56] er et opensource verktøy utviklet for å gjøre det enkelt å kjøre applikasjoner i containere. En container tillater en utvikler å pakke en applikasjon med alle avhengigheter og nødvendige biblioteker i en enkelt avbildning. Dette kan kjøres på en hvilken som helst datamaskin som har Docker rammeverket installert. I vårt system kjører begge frontend og alle IoT Edge moduler i Docker containere.

LXC (Linux Containers) [57] er en OS-nivå virtualiseringsteknikk som tillater kjøring av flere isolerte Linux containere på en host masking i en enkelt Linux kjerne. Vi har i vårt system benyttet LXC containere for å tillate kjøring av et stort antall simulerte IoT Edge instanser på en enkelt Azure virtuell maskin.

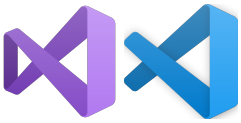
Azure DevOps [58] er platform som samler flere underliggende tjenester som underletter smidig utvikling og DevOps[59] metodikk. Under arbeidet med denne oppgaven har vi brukt Azure DevOps som prosjektstyringsverktøy, til versjonskontroll og for å automatisere bygg, test og utrulling av software. Azure DevOps tjenestene vi har benyttet oss av er:

Azure Boards er et prossess-styrings verktøy som tilbyr kanban brett, sprint tracking, backlogger og rapporteringsverktøy for sprint-retrospektiver. Vi har i dette prosjektet utelukkende benyttet oss av Azure Boards for prossess-styring og planlegging av arbeid.

Azure Repos tilbyr Git repository hosting med full integrasjon mot alle andre tjenester under Azure Devops paraplyen. Vi har hostet prosjektets Git repository her, og dratt stor nytte av integrasjonen som tilbys mot Azure Boards og Azure Pipelines.

Azure Pipelines tilbyr bygg, automatisk testing og utrulling applikasjoner i skyen. Vi har i dette prosjektet ønsket å arbeide etter Continuous Integration / Continuous Delivery prinsipper, og har benyttet oss av Azure Pipelines for å oppnå dette. Vi har konfigurert pipelines for alle komponenter som automatisk tester, bygger og ruller ut ny versjon ved push av ny kode til et prekonfigurert repository.

10.4 Utviklingsmiljø - IDE



10.4.1 Visual Studio Enterprise / Visual Studio Code

er integrerte utviklingsmiljø (IDE) fra Microsoft. Med disse har det gjort det mulig for oss å følge offisielle guider fra Microsoft og få ferdige maler med kode for forskjellige plattformer i Azure. De har et statisk kodeanalyseverktøy innebygd som heter IntelliSense som har gjort det enklere for oss å skrive kode. Disse utviklingsmiljøene har en rekke utvidelser som kommuniserer med Azure. Dette har gjort det mulig for oss å utplassere kode direkte fra utviklingsmiljøet.

10.4.2 Jetbrains Webstorm / Rider

er integrerte utviklingsmiljøer basert på JetBrains IntelliJ platform. Med disse utviklingsmiljøene har vi kunnet skrive, teste og kjøre de egentutviklede komponentene i systemet på en effektiv måte. Disse utviklingsmiljøene tilbyr også svært solide verktøy for feilsøking, blant annet debuggere som tillater å steppe gjennom koden, samt funksjonalitet for å evaluere uttrykk i koden på et gitt punkt i eksekusjonssekvensen.

10.4.3 Postman

er et HTTP test verktøy for å gjøre forespørsler mot nettsider og rest API-er for dataeksponering og testing. Ved å kunne bygge opp HTTP forespørsler på en enkel måte gjør Postman det enklere å teste eksponerte API tjenester og serverautentisering ved hjelp av deres innebygde tilpasningsdyktig header oppsett.

Tillegg A

Ordliste

AMQP (Advanced Message Queuing Protocol) er en åpen standard for kommunikasjon mellom foretak eller tilkoblede enheter. AMQP har mange av de samme bruksområdene som MQTT men tilbyr utvidet funksjonalitet og dermed også mere overhead pr. melding.

API Application programming interface. Et grensesnitt som eksponerer deler av et programvaresystem.

ARM-Templates For å implementere infrastruktur som kode i et Azure system benyttes ARM-templates. En template er en JSON fil som definerer og konfigurerer infrastrukturen i et system. Disse templatene bruker et deklarativt syntax som tillater bruker å definere hvilke komponenter har eller hun ønsker å benytte, uten å måtte skrive ut en detaljert kjede av kommandoer for å opprette disse.

ACR (Azure Container Registry) er en tjeneste som tillater lagring og henting av Docker images. ACR tilbyr fullstendig integrasjon med andre Azure tjenester som Azure Pipelines, Azure IoT Edge og Azure IoT Hub.

Azure Cost Analysis er en tjeneste som tilbyr kostnadsdata for alle Azure komponenter som inngår i en ressursgruppe. Kostnader kan brytes ned basert på ressurs og filtreres på tid.

Azure Monitoring er en tjeneste som tilbyr omfattende ytelses og helsedata for Azure komponenter.

Azure Portal er et nettbasert enhetlig grafisk grensesnitt som brukes for å konfigurere og kontrollere azure tjenester og abbonementer. Man kan bygge, kontrollere, konfigurere og monitorere alt ifra enkle web-apper til komplekse sky-arkitekturer.

CRUD-operasjoner Create, Read, Update og Delete. Samlebetegnelse for de fire grunnleggende funksjonene for persistent lagring av data.

Continuous Deployment (kontinuerlig utplassering) er en praksis hvor man har som fokus å inkrementelt og hyppig produksjonssette endringer i programvare. Dette prinsippet henger tett sammen med kontinuerlig integrasjon. Kontinuerlig utplassering avhenger av at man har en automtisert prosess for å verifisere å utplassere endringer i kodebasen til produksjonskode.

Continuous Integration [60] (kontinuerlig integrasjon), er en praksis innen programvareutvikling hvor utviklere enkelt og hyppig, gjerne på daglig basis, kan integrere arbeidet han eller hun har gjort, i den overordnede kodebasen. For at dette skal fungere i praksis og for å sørge for at kodebasen består av feilfri, høykvalitets kode må hver integrasjon verifiseres av et automatisert bygg og testsystem.

DevOps [30] [61] er en praksis hvor man etterstreber å redusere tiden det tar fra en utvikler gjør en endring i et system, til denne endringen havner i produksjonskode. Navnet DevOps kommer av en sammenslåing av termene *software development* (Dev) og *it operations* (Ops). Begrepet devops brukes ofte i tandem med prinsipper for continuous integration og continuous deployment.

GUID / UUID Globally unique identifier / Universally unique identifier eller globalt unik identifikator . En autogenerert id som for alle formål kan ansees som unik dersom den er generert etter gjeldende standarder.

IDE Integrated development environment eller integrert utviklingsmiljø. Et verktøy som underletter utvikling av programvare ved å tilby statisk kodeanalyse, autokomplettering, kompilering, søkeverktøy og versjonskontroll.

IoT Internet of things eller tingenes internett. Et nettverk av sammenkoblede enheter hver med en global unik id (GUID) som kan overføre data over et nettverk.

JSON JavaScript object notation, et tekstbasert menneske-leselig format for dataoverføring.

JWT [37](Json Web Token) er en åpen standard som tilbyr en kompakt og selvforsynt metode for sikker overføring av informasjon mellom to parter i form av et JSON objekt. Informasjonen i dette JSON objektet kan stoles på, fordi objektet har en digital signatur som kan verifiseres av mottaker. Et token kan dermed sees på som en nøkkel som gir tilgang til en ressurs. Det betyr at man må passe på at tokenet ikke havner i feil hender.

Kanban er et agilt rammeverk som har sine røtter i Toyotas arbeid med optimalisering av sine prosesser sent på 1940 tallet. Kanban fokuserer på å kontrollere arbeidsflyt. Kanban baserer seg i stor grad på bruken av et kanban brett for visulisering av arbeidsoppgavers flyt og fremgang. Kanban kan oppsumeres i følgende dogmer:

- Visualiser arbeidsflyt
- Begrens antall pågående oppgaver
- Håndter oppgaveflyt gjennom hele prosessen

- Gjør alle retninglinjer tydelige og eksplisitte
- Daglige standupmøter for kontinuerlig evaluering og forbedring av prosessen
- Samarbeidsbasert utvikling, forbedring og kunnskapsoverføring

Modbus/TCP Modbus/TCP er en kommunikasjonsprotokoll publisert i 1979 av Scheider Electric, tidligere Modicon for bruk i Programerbare Logisk Styring (PLS). Modbus har over årenes løp blitt den foretrukne kommunikasjonsprotokollen innen industriell automasjon og styring, som følge av at det er en åpent publisert, lisensfri protokoll.[62] Modbus/TCP er en variant av Modbus protokollen som tillater kommunikasjon i et inter eller intranett ved bruk av TCP/IP protokoller. Det vanligste use caset er sammenkobling av PLSer i ett ethernet nettverk.[63, p. 3]

MQTT [64][65] (MQ Telemetry Transport) er en tilkoblingsprotokoll for maskin-til-maskin kommunikasjon utviklet for bruk i IoT-enheter. Protokollen er designet for å være en ekstremt lettvekts utgi/abboner type meldingstransport. Protokollen er godt egnet for bruksområder der båndbredde er en begrenset ressurs. Eksempler på dette er bruk i satelittlinker, oppringningstilkoblinger i helsevesenet og i små integrerte enheter, brukt til for eksempel hjemmeautomasjon.

Objektrelasjonell database bygger på prinsippene til relasjonsdatabase, men har ekstra funksjonalitet som innebygd som er basert på objektorientert programmering. Den har støtte for lagring av objekter, metoder innenfor disse og opererer med pekere inne i databasen[66].

Platform-as-a-service Platform-as-a-service (PaaS) er en type skytjeneste som tilbyr en plattform for personer for å utvikle og kjøre applikasjoner uten å tenke på den underliggende arkitekturen og maskinvaren.

Relasjonsdatabase er en databasemodell som består av tabeller som har relasjoner til hverandre ved hjelp av nøkler. Hver rad i hver tabell har sin unike nøkkel som kan brukes til å identifisere raden når en annen tabell har en relasjon til nøyaktig den raden.

Scrum er et agilt rammeverk for samarbeid i team, hovedsaklig brukt innen programvareutvikling. Scrum er et rammeverk basert på timeboksing. Det vil si at man deler arbeidet inn i tydelig definerte tidsintervaller, kalt sprinter i scrum, og at det settes et mål for slutten av sprinten. Scrum rammeverket kan kort oppsummeres i følgende punkter:

- Levering eller demo av funksjonalitet etter hver sprint
- Ledelsen setter sammen et team, og teamet har full autonomi ellers i prosessen
- Evaluer og tilpass arbeidet hver eneste dag gjennom daglige, korte standup møter
- Hvert team har en scrum-master som fjerner hindringer og forstyrrelser som hindrer utviklerne i å gjøre jobben sin effektivt.
- Den eneste personen som skal påvirke teamets prioriteringer, behov og krav er produkteieren.

Tidsseriedata er registrering av en variabel med faste mellomrom over tid. Eksempler på tidsseriedata er avlesning av temperaturen i lufta med fem minutters mellomrom over en uke.

Tillegg B

Kode-eksempler

B.1 deployIoTEdge-symmetricKey-range.sh

Bash script som setter igang utplassering av IoT Edge enheter brukt i lasttesting av systemet. Scriptet verifiserer at nødvendige parametere sendes inn og utfører sekvensiell inrullering av et spesifisert antall IoT Edge enheter.

```
#!/bin/bash
help () {
        echo "Usage:"
        echo "[from] [to]       the range of devices to be deployed"
        echo "-p                enrollment Primary Key"
        exit 0

}
if [[ $# == 0 ]]; then
        help
fi

while test $# -gt 0; do
        case "$1" in
        -h|--help)
                help
                ;;
        -p)
                shift
                KEY=$1
                shift
                ;;
        *)
                reg='^[0-9]+$'
                if ! [[ $1 =~ $reg && $2 =~ $reg ]]; then
                        help
                fi
                if [[ $1 -gt $2 ]]; then
                        help
                fi
                from=$1
                to=$2
                shift
                shift
```

```
                ;;
        esac
done


if [[ $from == ""  || $to == "" || $KEY == "" ]]; then
        help
fi

for i in $(eval echo {$from..$to}); do
        echo "./deployIoTEdge-symmetricKey.sh -n iot$i -p $KEY -s $scopeId"
done
```

Listing B.1: Script 1 for oppstart av belastningstest

B.2 deployIoTEdge-symmetricKey.sh

Bash script som starter en LXC container, genererer symmetriske nøkler basert på en primærnøkkel Device Provisioning Service og maskinnavnet til IoT Edge instansen som skal startes opp. Deretter modifiserer scriptet konfigurasjonsfilen til IoT Edge instansen slik at denne kan autentiseres mot Device Provisioning Service.

```
 #!/bin/bash
help () {
        echo "Usage:"
        echo "-n         name of device"
        echo "-p         enrollment Primary Key"
        echo "-s         scope id from the provisioning service"
        exit 0

}
if [[ $# == 0 ]]; then
        help
fi

while test $# -gt 0; do
        case "$1" in
        -h|--help)
                help
                ;;
        -n)
                shift
                edgeName=$1
                shift
                ;;
        -p)
                shift
                KEY=$1
                shift
                ;;
        -s)
                                shift
                                scopeId=$1
```

```
                                shift
                ;;
        *)
                break
                ;;
        esac
done

if [[ $edgeName == ""  || $KEY == "" || $scopeId == "" ]]; then
        help
fi

lxc launch iotedge $edgeName -c security.nesting=true -c security.privileged=
    true

keybytes=$(echo $KEY | base64 --decode | xxd -p -u -c 1000)
KEY=`echo -n $edgeName | openssl sha256 -mac HMAC -macopt hexkey:$keybytes -
    binary | base64`

lxc exec $edgeName -- sed -i "31,33 s/^/#/" /etc/iotedge/config.yaml
lxc exec $edgeName -- sed -i "45,52 s/^# //" /etc/iotedge/config.yaml
lxc exec $edgeName -- sed -i "48 s/{scope_id}/$scopeId/" /etc/iotedge/config.
    yaml
lxc exec $edgeName -- sed -i "51 s/{registration_id}/$edgeName/" /etc/iotedge/
    config.yaml
lxc exec $edgeName -- sed -i "52 s~{symmetric_key}~$KEY~" /etc/iotedge/config.
    yaml
lxc exec $edgeName -- sed -i "s/hostname: \".*\"/hostname: \"$edgeName\"/g" /
    etc/iotedge/config.yaml
lxc exec $edgeName -- bash -c "systemctl restart iotedge"
echo "Done with $edgeName"
```

Listing B.2: Script 2 for oppstart av belastningstest

B.3 delete-deploymentRange.sh

Bash script som stopper og sletter alle LXC-containere brukt i en test-kjøring.

```
#!/bin/bash
help () {
        echo "Usage:"
        echo "[from] [to]       the range of devices to be deployed"
        exit 0

}
if [[ $# == 0 ]]; then
        help
fi

while test $# -gt 0; do
        case "$1" in
        -h|--help)
                help
                ;;
```

```
        *)
                reg='^[0-9]+$'
                if ! [[ $1 =~ $reg && $2 =~ $reg ]]; then
                        help
                fi
                if [[ $1 -gt $2 ]]; then
                        help
                fi
                from=$1
                to=$2
                shift
                shift
                ;;
        esac
done


if [[ $from == ""  || $to == "" ]]; then
        help
fi
edges=""
for i in $(eval echo {$from..$to}); do
        edges+="iot$i "
done
lxc stop $edges
lxc delete $edges
```

Listing B.3: Script stansing og opprydning av belastningstest

Bibliografi

[1] Wikipedia. Kongsberg digital — wikipedia,. https://no.wikipedia.org/w/index.php?title=Kongsberg_Digital&oldid=20115987, 2020. [besøkt 31-mars-2020].

[2] Kongsberg Digital AS. Kognifai - unlock the value of <your data. https://www.kongsberg.com/digital/kognifaiecosystem/, 2020. [besøkt 31-mars-2020].

[3] Jay Lee, Hung-An Kao, Shanhu Yang, et al. Service innovation and smart analytics for industry 4.0 and big data environment. *Procedia Cirp*, 16(1):3–8, 2014.

[4] Robail Yasrab. Platform-as-a-service (paas): The next hype of cloud computing. *arXiv preprint arXiv:1804.10811*, 2018.

[5] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[6] Henrik Kniberg and Mattias Skarin. *Kanban and Scrum-making the most of both*. Lulu. com, 2010.

[7] Wikimedia Commons. File:simple-kanban-board-.jpg — wikimedia commons, the free media repository, 2019. [Online; accessed 28-April-2020].

[8] Dag Sjøberg and Yngve Lidsjørn. *Selected chapters from Software Engineering*. Pearson Education Limited, 2016.

[9] Datanyze. Github market share. https://www.datanyze.com/market-share/source-code-management--315/github-market-share. [besøkt 28-april-2020].

[10] Vincent Driessen. A successful git branching model. https://nvie.com/posts/a-successful-git-branching-model/. [besøkt 29-april-2020].

[11] Atlassian. Gitflow workflow. https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. [besøkt 29-april-2020].

[12] Slack Technologies Inc. What is slack? https://slack.com/intl/en-no/help/articles/115004071768-What-is-Slack-. [besøkt 4-mai-2020].

[13] Wikipedia contributors. Slack (software) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Slack_(software)&oldid=954637014, 2020. [Online; accessed 4-May-2020].

[14] Microsoft. What is azure iot edge? https://docs.microsoft.com/en-us/azure/iot-edge/about-iot-edge. [besøkt 2-mai-2020].

[15] Microsoft. What is azure iot hub? https://docs.microsoft.com/en-us/azure/iot-hub/about-iot-hub. [besøkt 2-mai-2020].

[16] Microsoft. Provisioning devices with azure iot hub device provisioning service. https://docs.microsoft.com/en-us/azure/iot-dps/about-iot-dps. [besøkt 2-mai-2020].

[17] Microsoft. Welcome to azure cosmos db. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction. [besøkt 02-mai-2020].

[18] Github contributors. Timescaledb. https://github.com/timescale/timescaledb. [besøkt 4-mai-2020].

[19] Microsoft. App service overview. https://docs.microsoft.com/en-us/azure/app-service/overview. [besøkt 4-mai-2020].

[20] Microsoft. What is azure stream analytics? https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-introduction. [besøkt 4-mai-2020].

[21] Microsoft. Welcome to azure blob storage. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction. [besøkt 02-mai-2020].

[22] Microsoft. What is azure time series insights? https://docs.microsoft.com/en-us/azure/time-series-insights/time-series-insights-overview. [besøkt 4-mai-2020].

[23] Microsoft. Sla for azure database for postgresql. https://azure.microsoft.com/en-us/support/legal/sla/postgresql/v1_1/.

[24] Marino Posadas and Tadit Dash. *Dependency Injection in. NET Core 2.0: Make use of constructors, parameters, setters, and interface injection to write reusable and loosely-coupled code*. Packt Publishing Ltd, 2017.

[25] ESLint. Eslint static code analysis. https://eslint.org/.

[26] Richard Bellairs. What is static analysis. https://www.perforce.com/blog/sca/what-static-analysis.

[27] Todd Fredrich. Http status codes. https://www.restapitutorial.com/httpstatuscodes.html.

[28] Wikipedia contributors. Representational state transfer — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Representational_state_transfer, 2020. [Online; accessed 4-May-2020].

[29] Kelly Gremban, Alberto Vega, Beth Harvey, Theanno Petersen, Shahan Robin, and Nick Schonning. Understand the azure iot edge runtime and its architecture. https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-runtime.

[30] Liming Zhu, Len Bass, and George Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016.

[31] Microsoft. Simulatedtemperaturesensor. https://github.com/Azure/iotedge/tree/master/edge-modules/SimulatedTemperatureSensor.

[32] Microsoft. Azure functions overview. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview. [besøkt 29-april-2020].

[33] Microsoft. Azure functions scale and hosting. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale#how-the-consumption-and-premium-plans-work. [besøkt 6-mai-2020].

[34] Inc. MongoDB. What is a document database? https://www.mongodb.com/document-databases. [besøkt 02-mai-2020].

[35] Steve Smith and Scott Addie. Dependency injection in asp.net core. https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1. [besøkt 4-mai-2020].

[36] Wikipedia contributors. Hmac — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=HMAC&oldid=955119550, 2020. [besøkt 9-mai-2020].

[37] AuthO. Introduction to json web tokens. https://jwt.io/introduction/.

[38] Mozilla. Cross-origin resource sharing (cors). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[39] Microsoft. What is azure time series insights. https://docs.microsoft.com/en-us/azure/time-series-insights/time-series-insights-overview.

[40] Microsoft. Azure time series insights pricing. https://azure.microsoft.com/en-us/pricing/details/time-series-insights/.

[41] Microsoft. What is azure pipelines? https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops.

[42] Microsoft. C# language specification - introduction. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction, 2017. [besøkt 27-april-2020].

[43] David Flanagan. *JavaScript: the definitive guide*. "O'Reilly Media, Inc.", 2006.

[44] Python software foundation. What is python - executive summary. https://www.python.org/doc/essays/blurb/. [besøkt 27-april-2020].

[45] Microsoft. .net core overview. https://docs.microsoft.com/en-us/dotnet/core/about. [besøkt 27-april-2020].

[46] What is nunit. https://nunit.org/. [besøkt 27-april-2020].

[47] Moq readme. https://github.com/moq/moq4. [besøkt 27-april-2020].

[48] Jetbrains. dotcover. https://www.jetbrains.com/dotcover/.

[49] OpenJS Foundation. About node.js. https://nodejs.org/en/about/. [besøkt 27-april-2020].

[50] Wikipedia contributors. Express.js — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Express.js&oldid=944443884, 2020. [Online; accessed 28-April-2020].

[51] Microsoft. Tsiclient: The azure time series insights javascript sdk. https://github.com/Microsoft/tsiclient.

[52] What is mocha. https://mochajs.org/#more-information. [besøkt 28-april-2020].

[53] Chai. Chai assertions library. https://www.chaijs.com/.

[54] istanbul. istanbul coverage libarary. https://www.istanbul.js.org/.

[55] Bootstrap. Bootstrap. https://getbootstrap.com/.

[56] What is docker. https://opensource.com/resources/what-docker. [besøkt 28-april-2020].

[57] Wikipedia contributors. Lxc — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=LXC&oldid=950952254, 2020. [Online; accessed 28-April-2020].

[58] Devops overview. https://azure.microsoft.com/en-in/services/devops/#DevOps. [besøkt 28-april-2020].

[59] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[60] Martin Fowler and Matthew Foemmel. Continuous integration, 2006.

[61] Wikipedia contributors. Devops — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=DevOps&oldid=952709409, 2020. [besøkt 9-mai-2020].

[62] IDA Modbus. Modbus application protocol specification v1. 1a. *North Grafton, Massachusetts (www. modbus. org/specs. php)*, 2004.

[63] Andy Swales et al. Open modbus/tcp specification. *Schneider Electric*, 29, 1999.

[64] Mqtt.org. Mqtt.org. http://mqtt.org/.

[65] Andy Stanford-Clark and Hong Linh Truong. Mqtt for sensor networks (mqtt-sn) protocol specification. *International business machines (IBM) Corporation version*, 1:2, 2013.

[66] Wikipedia contributors. Objektrelasjonell database. https://no.wikipedia.org/wiki/Objektrelasjonell_database.