Universitetet i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Datateknologi	Vår semesteret 2021 Åpen
Student/studenter: Henrik Skulevold Ådne Øvrebø Vemund Refnin	Henrik Skulevold …… Ådne Øvrebø …… Vemund Refnin …… signatur (er)
Faglig ansvarlig: Martin Georg Skjæveland Veileder(e): Martin Georg Skjæveland	
Tittel på oppgaven: Visualisering av produksjonsdata – Tytlandsvik Aqua Engelsk tittel: Visualization of production data – Tytlandsvik Aqua	
Studiepoeng: 20	
Emneord: Fiskeoppdrett, API, Webapplikasjon, skytjenester, CI/CD, Databehandling	Sidetall: 186 + vedlegg/annet: 4 Stavanger, 15.05.2021

Sammendrag

Oppgaven tar utgangspunkt i en problemstilling hos Tytlandsvik Aqua vedrørende samling, strukturering og presentering av data generert på deres landbaserte oppdrettsanlegg for smolt. Ønsket var å lage et system for tilgjengeliggjøring av data slik at det kunne gjennomføres gode, presise og raske dataanalyser. Systemet skulle kunne kjøres lokalt på anlegget eller som en skyløsning, samt kunne videreutvikles.

Problemstillingen ble løst ved å utvikle et system bestående av flere tjenester som kommuniserer med hverandre. Systemet består av et API, en tjeneste for innhenting av informasjon og en webapplikasjon. APIet har som formål å behandle, eksponere og lagre dataene knyttet til systemet i en database. Tjenesten for innhenting av informasjon er ansvarlig for å hente data fra anleggets systemer og sende de videre til APIet for prosessering og oppbevaring. Webapplikasjonen konsumerer dataene fra APIet og eksponerer et brukergrensesnitt for å visualisere innhentet data. Systemet inkluderer mulighet for å registrere manuelle vannmålinger, gjort av personell på anlegget, i webapplikasjonen. Samtlige tjenester ble kjørt som enkeltstående beholdere for å kunne kjøres tilnærmet hvor som helst - lokalt på anlegget eller som en skyløsning.

Figur 1: Side for presentering av data fra webapplikasjonen til den ferdige løsningen.

Forord

Denne bacheloroppgaven er avsluttende for studiet Datateknologi ved Universitetet i Stavanger. Det har vært lærerikt å få et innblikk i en annen bransje og hvordan man kan benytte det som er lært på studiet til å utvikle systemer som kan gi verdi til en bedrift.

Ved dette ønsker vi å rette en stor takk til Tytlandsvik Aqua for å ha bistått i utviklingen av dataverktøyet. Vi ønsker også gi en takk til veileder Martin G. Skjæveland for god veiledning samt gode tips innenfor presentering av store datamengder.

Innholdsfortegnelse

Forkortelser

API – Application Platform Interface

CDN – Content Delivery Network

CLI – Command Line Interface

CRUD – Create, Read, Update, Delete

CSS – Cascading Style Sheets

DTU – Data Tier Unit

DFD – Data Flow Diagram

HTML – Hypertext Markup Language

HTTP – Hypertext Transfer Protocol

HTTPS – Hypertext Transfer Protocol Secure

IDE – Integrated Development Environment (Integrert utviklermiljø)

JS – JavaScript

JSON – JavaScript Object Notation

NPM – Node Package Manager

NF – Normal form

ORM – Object Oriented Mapping

OWASP - Open Web Application Security Project

PLS – Programmerbar Logisk Styring (Industridatamaskin)

QA – Quality Assuranse

RAS – Resirculation Aquaculture System

SCADA - Supervisory Control and Data Acquisition

SDK – Software Developer Kit

SEO – Search Enigne Optimalization

SPA – Single Page Application

SQL – Structured Query Language

STRIDE - Spoofing, Temparing, Repudiation, Infomation disclosure, Denial of service and Elevation of Privilege

TCP – Transmission Control Protocol

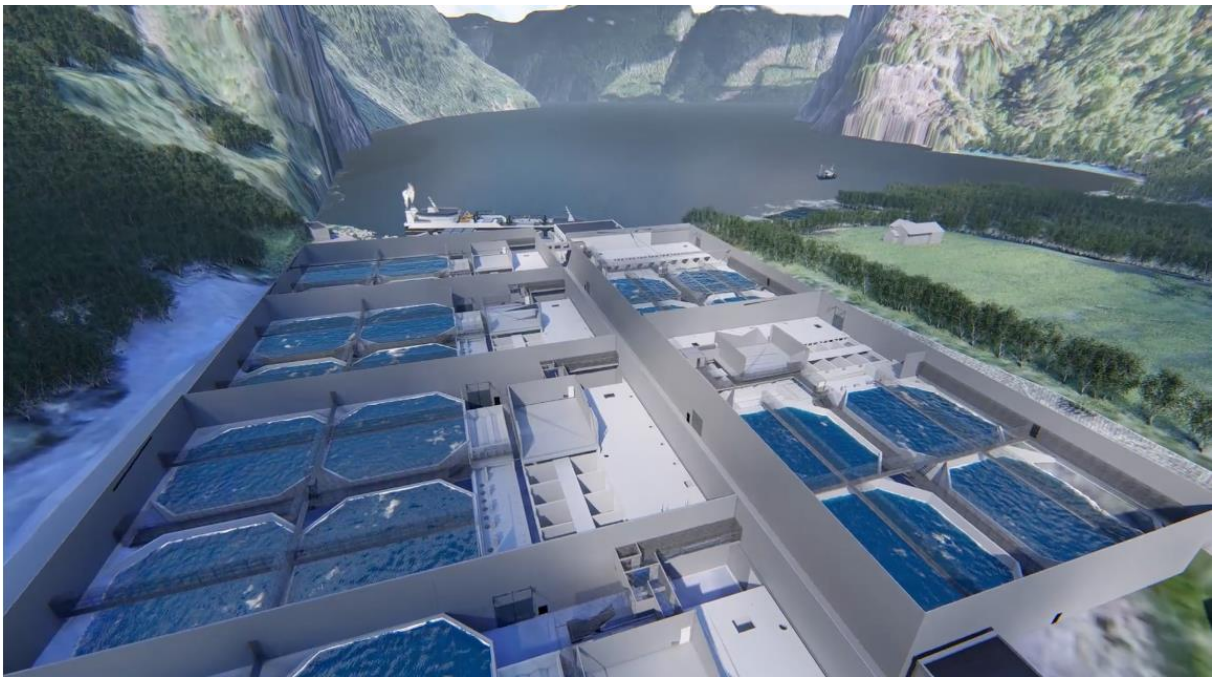
TLS – Transport Layer Security

TS – Typescript

XSS – Cross Site Scripting

1 Innledning

Før bacheloroppgaven ble definert var ønsket at oppgaven skulle dreie seg om å behandle store mengder med data. Det var også ønskelig at dataene skulle ha en nytteverdi for en bedrift. Den konkrete ideen til bacheloroppgaven ble utledet i samarbeid med Tytlandsvik Aqua. Denne bedriften driver et landbasert fiskeoppdrettsanlegg som per dags dato har kapasitet til å produsere 3000 tonn storsmolt i året (Jenssen, 2021). Storsmolt er ung laks som veier fra 200 til 1000 gram. Tytlandsvik Aqua leverer fisken videre til matfiskoppdrett, hovedsakelig i Rogaland. Fisken oppbevares altså bare en periode av livet sitt hos Tytlandsvik Aqua før den sendes videre til havbruksanlegg.



Figur 2: Illustrasjonsbilde fra anlegget (de til høyre er ferdigstilt pr.d.d.).
Hentet fra: https://taqua.no/, Nedlastet 15.04.2021 11:37

Det landbaserte oppdrettsanlegget til Tytlandsvik Aqua er en ny måte å drive oppdrett på. Det er omtrent 200 slike anlegg i Norge. Landbaserte anlegg gjør det mulig å justere og overvåke langt flere parametere enn ved tradisjonelt havbruksanlegg. På havbruksanlegg er det begrenset hvilke parametere man har full kontroll over, og reduseres i stor grad til kun å omhandle mengden og typen fôr. Vannverdier for eksempel, har man minimal kontroll og påvirkningskraft på. Dersom eksempelvis vanntemperaturen synker dramatisk får man ikke gjort noe med det. Havbasert oppdrett er mer eksponert for eksterne trusler som lakselus, enkelte alger og sykdommer.

Mulighetene ved landbasert oppdrett til å overvåke dataene og justere vannverdiene, som for eksempel temperatur, oksygen og ozon, gir mange fordeler. Med muligheten til å justere vannverdiene, oppstår et større behov for å samle inn og analysere data. Gode, presise og raske analyser gir anledning til raskt å kunne justere vannverdiene for å gi optimale vilkår for fisken. Optimale vilkår vil gi mer effektiv vekst, mindre dødelighet og bedre kvalitet på fisken. Det var i sammenheng med dette at Tytlandsvik Aqua fremmet et behov for gode, presise og raske dataanalyser at denne oppgaven kom i stand.

2 Leserveiledning

Resten av oppgaven vil være strukturert som følger:

Kapittel 5 - Krav: Beskriver funksjonelle og ikke-funksjonelle krav til løsningen og hvordan disse kravene kan være til hjelp ved utviklingen av systemet.

Kapittel 6 – Overordnet arkitektur: Forklarer hvordan systemets tjenester i sin helhet henger sammen, hva som er deres ansvarsområde og hvordan dataflyten mellom tjenestene er.

Kapittel 7 – Teori og teknologivalg: Beskriver de ulike teknologiene som er benyttet og hvorfor denne teknologien er anvendt. Tanken bak dette kapittelet er å være et oppslagsverk dersom man ikke har kompetanse om en teknologi eller teori. Kapitler man har bakgrunnskunnskap om er ikke essensielle å lese for å forstå oppgaven, men de kan være relevante for å forstå valget av teknologier og rammeverk samt hvorfor de trengs for å løse problemstillingen med henvisning til krav.

Kapittel 8 – Arbeidsmetodikk: Forklarer hvordan arbeidet med dette prosjektet er strukturert og hvordan ulike arbeidsmetodikker er benyttet.

Kapittel 9 – Arkitektur: Forklarer og begrunner hvordan systemet er satt sammen mer i detalj enn i overordnet arkitektur kapittelet, samt hvordan ulike deler av systemet fungerer og deres hensikt.

Kapittel 10 – Implementasjon: Beskriver teknisk hvordan de ulike delene av systemet er implementert og hvordan systemet ser ut for brukeren.

Kapittel 11 – Sikkerhetsanalyse: Beskriver systemets sikkerhet og hvordan kravene tilknyttet sikkerheten blir imøtekommet.

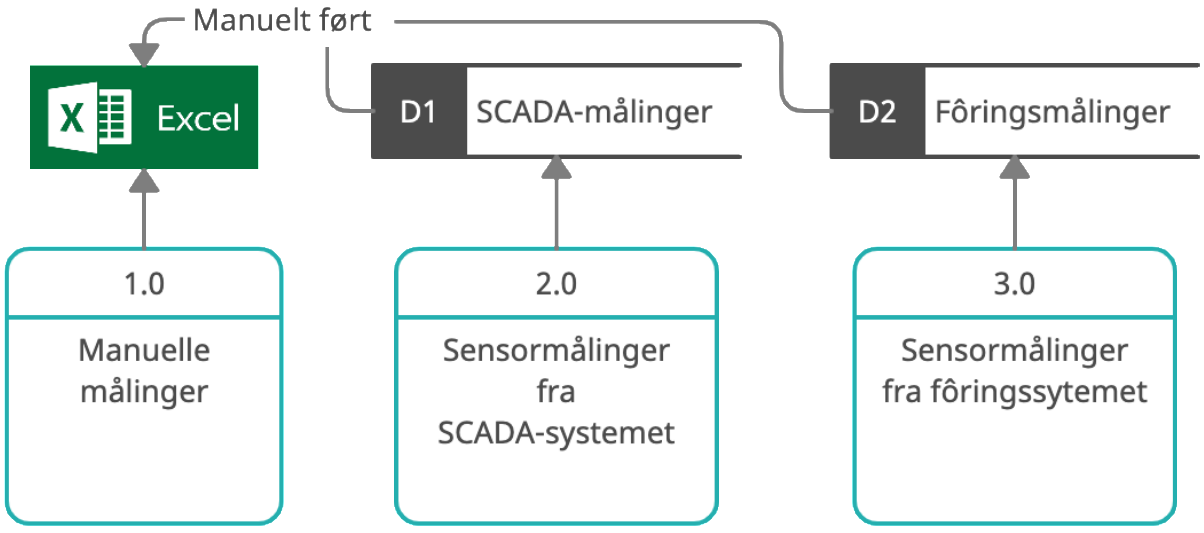
Kapittel 12 – Resultat og brukerevaluering Forklarer hvordan kravene er imøtekommet samt en brukerevaluering til systemet.

Kapittel 13 – Diskusjon: Beskriver hva som skal til for å få systemet helt produksjonsklart, samt videreutviklingsmuligheter og forbedringer.

3 Oppgavebeskrivelse

Per i dag har Tytlandsvik Aqua automatiske systemer som overvåker, samler og lagrer data om vannverdier, fôringsverdier, strømforbruk og mer. Fôringssystemet lagrer verdiene automatisk i en egen SQL database. Et annet system som kalles SCADA, lagrer automatisk alle målinger som blir gjort av vannet i tillegg til andre generelle verdier fra driften av anlegget, eksempelvis strømforbruk. SCADA systemet lagrer dataene i loggfiler. Fôringssystemet og SCADA-systemet er altså uavhengige, og dataene blir lagret på to forskjellige lokasjoner.

En tredje kilde til data er daglige manuelle målinger. Dette er målinger som manuelt blir gjort av personellet i form av vannprøver, samt noe data hentet fra SCADA- og fôringssystemet (Sedberg, 2020). Denne dataen blir videre ført inn i et regneark. Vannprøvene måler verdier anlegget ikke har sensorer til å måle. Grunnen til at de også lagrer noe data fra SCADA- og fôringssystemet manuelt her er for at de lettere kan sammenligne denne dataen med data de får av vannprøver. Regnearket blir brukt til presentasjon av data i rapporter, samt noe enkel analyse der de setter parametere opp mot hverandre. Denne prosessen blir i lengden tungvint, særlig med tanke på å sammenligne innsett mot hverandre. Dataen er altså per i dag ikke lagret systematisert og strukturert og er således vanskelig å tilgjengeliggjøre og kompliserer dataanalysen.



Figur 3: 1.0 Manuelle målinger utført av personell og registrert i et Excel-ark. 2.0 Automatiske sensormålinger fra SCADA-systemet som føres inn i databasen D1. 3.0 Automatiske sensormålinger fra fôringsystemet som føres inn i databasen D2. Det blir manuelt hentet ut én måling fra databasene D1 og D2 daglig og ført inn i Excel.

I dialog med Tytlandsvik Aqua kom det fram et ønske om et dataverktøy som kan strukturere og systematisere lagring av dataen, slik at denne enkelt kan presenteres og innhentes for å gjennomføre analyser. Tytlandsvik Aqua ønsker også at verktøyet skal erstatte Excel-arket der de registrerer de manuelle verdiene. For å skape en god struktur på dataen trengs det en tett dialog med Tytlandsvik Aqua. Primært er Tytlandsvik Aqua sitt ønske at verktøyet skal brukes som et hjelpemiddel for personell med det biologiske ansvaret. Denne bacheloroppgaven går dermed ut på å utvikle et slikt verktøy.

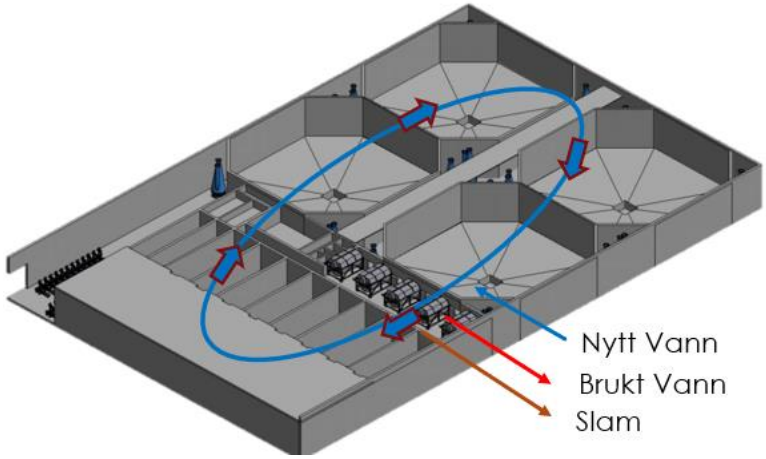
Tytlandsvik Aqua har ønsker om at verktøyet skal kunne videreutvikles og kunne benyttes til andre formål, som for eksempel å finne ut hvordan de kan kutte ned på strømforbruket. Dette innebærer at verktøyet må utvikles med tanke på videreutvikling.

Etter flere runder i dialog med biologisk ansvarlig, har spesifikasjonene for hvordan dataene skal presenteres blitt definert. Denne visualiseringen skal legge til rette for at man enkelt kan sammenligne data fra flere kilder og tidsperioder. Kildene det er snakk om er som nevnt fôringsverdier, SCADA-verdier og manuelle målinger.

Før man starter utviklingen av verktøyet, er det viktig å sette seg inn i hvordan driften hos Tytlandsvik Aqua foregår, dette for å få en god domeneforståelse. Med en god domeneforståelse er man bedre egnet til å imøtekomme og forstå Tytlandsvik Aqua sine ønsker, krav og behov.

4 Domeneforståelse

Begreper fra dette domenet, fiskeoppdrett på landbaserte anlegg, vil bli brukt gjennomgående i oppgaven og danner grunnlaget for utviklingen av dataverktøyet. Tytlandsvik Aqua er en bedrift som driver oppdrett av smolt i RAS-anlegg (Resirculation Aquaculture System). Et RAS-anlegg er et oppdrettsanlegg som er designet for å kunne produsere store mengder fisk med et relativt lavt vannforbruk (Norsk Røyeforum, 2021). Det brukes lite vann fordi anlegget har et internt renseanlegg der vannet går i sirkulasjon fra karene, som inneholder smolten, til renseanlegget. Ved å drive et landbasert anlegg som har et lukket miljø, kan man påvirke de ulike parameterne som videre påvirker smoltens kvaliteter.



Figur 4: Sirkuleringen av vann i et RAS-anlegg. Bilde tilsendt av Eli B. Jenssen.

Smolten på anlegget er ikke klekket der, men kommer fra et annet anlegg via tanklastebiler (Jenssen, 2021). Når smolten kommer til anlegget starter det som kalles for et ***innsett***. Dette er perioden fra smolten overføres fra lastebilen og til den blir tatt ut fra anlegget. Smolten føres fra tanklastebilen inn til vannbassenger som kalles ***kar,*** her lever fisken sin periode hos Tytlandsvik Aqua. Et kar er en stor beholder som kan inneholde opptil 200.000 smolt. Ved anlegget utgjør fire kar en ***hall***. Tytlandsvik Aqua har to ferdigstilte haller og flere er under konstruksjon og i planleggingsfasen. Det er per dags dato 8 definerte kar som overvåkes via en mengde ulike sensorer, der verdiene til sensorene blir lagret i fôring- og SCADA-systemet. Se figur 2 for oversikt over oppdeling av haller og kar.

5 Krav

I møter med Tytlandsvik Aqua er det spesifisert en del ønsker knyttet til dataverktøyet/systemet. Disse ønskene kan formuleres til både funksjonelle og ikke-funksjonelle krav. Det er viktig at kravene er tydelig definerte slik at man kan etterprøve om kravene er innfridd eller ikke. Disse kravene gir føringer for hvordan oppgaven bør løses. Det er også nødvendig å gjøre refleksjoner og vurderinger av hvordan man best mulig kan innfri kravene. Dette gjøres ved å senere se på teoretiske aspekter rund systemet som skal utvikles, i tillegg til å se på hvilke teknologiske løsninger som er mest hensiktsmessige.

5.1 Funksjonelle krav

Funksjonelle krav er krav til ulike funksjoner systemet eller programvaren skal ha (Altexsoft, 2018). Følgende krav er definert:

1. Webapplikasjonen skal visualisere innsamlet data på en måte som gjør at den kan brukes til å sammenligne dataene fra ulike tidsperioder og filtrere på ulike verdier.
2. Systemet skal innhente målinger som blir gjort manuelt på anlegget. Dette skal brukeren enkelt kunne legge inn selv og erstatte nåværende løsning med regneark.
3. Webapplikasjonen skal presentere sanntidsdata som blir hentet fra anlegget i en graf som kontinuerlig oppdateres.
4. Systemet skal automatisk formatere og lagre data fra SCADA- og fôringssystemet i en egen database.
5. Systemet må gjøre det mulig å spore fisken tilbake til hvilke kar de har vært i.
6. Systemet skal eksponere dataene slik at eksterne systemer enkelt kan konsumere dataene.
7. Systemet skal kunne brukes av flere brukere med ulike sikkerhetsklareringer, altså skille på lese-, skrive- og administratorrettigheter.
8. En bruker med administratorrettigheter skal kunne administrere alle brukerne i webapplikasjonen.
9. Webapplikasjonen må være sikret med innlogging og APIet skal kreve autorisering.

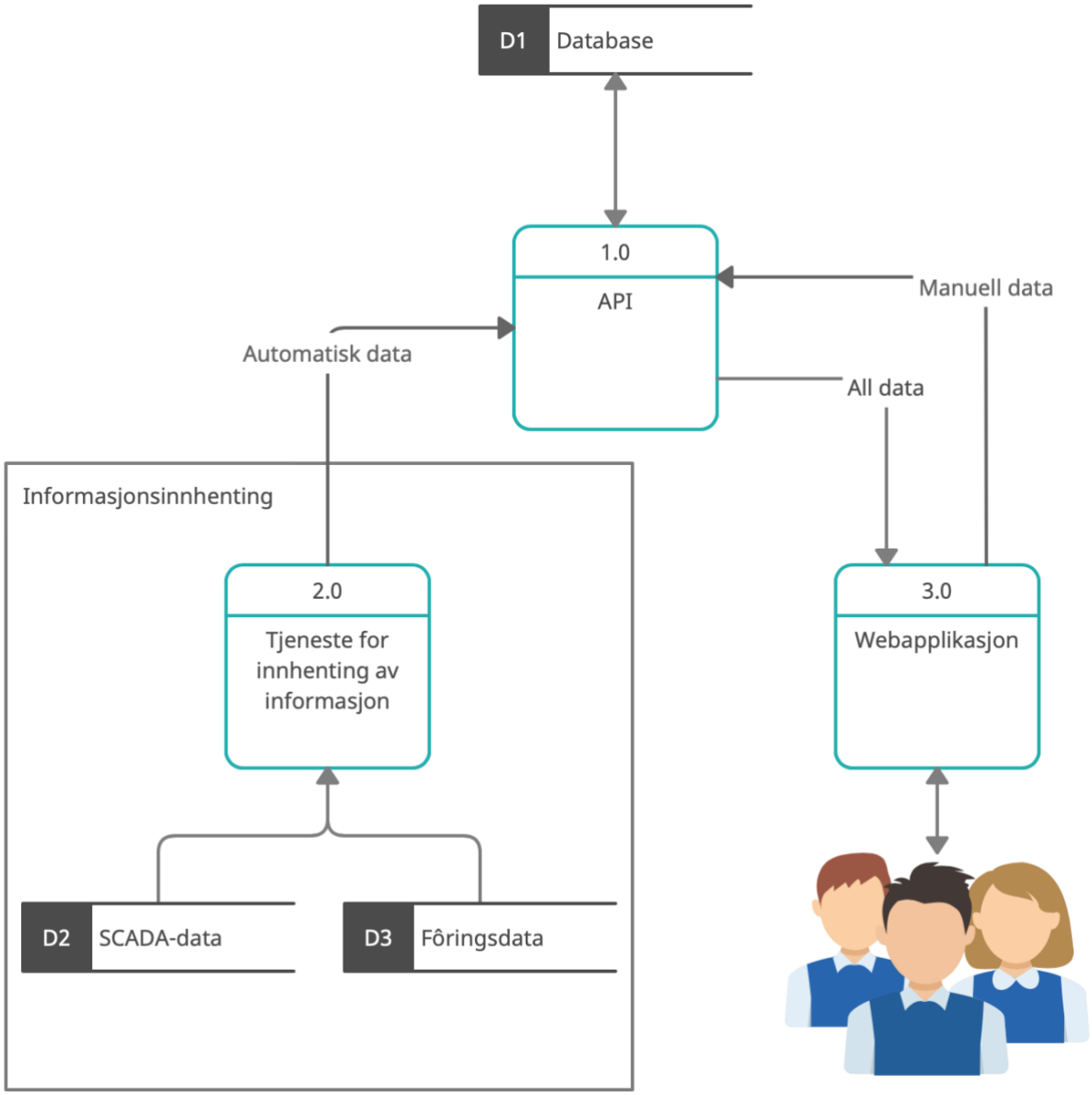
5.2 Ikke-funksjonelle krav

Ikke funksjonelle krav bestemmer atferd eller funksjonalitet systemet eller programvaren har. Frasen «Systemet skal være ...» blir ofte brukt for å definere de ikke-funksjonelle kravene. Man kan også si at ikke-funksjonelle er krav som fokuserer mer på kvalitet enn funksjonalitet (Altexsoft, 2018). De ikke-funksjonelle kravene er definert som følger:

10. Systemet skal kunne skalere for å håndtere økt brukeraktivitet og større datamengder, samt ha minimale driftskostnader.
11. Dataen må lagres på en sikker måte, for å unngå uautorisert tilgang samt tap av data ved systemsvikt.
12. Dataflyten i systemet må være sikker slik at ondsinnede aktører ikke kan se dataflyten mellom de ulike tjenestene.
13. Systemet må legge til rette for videre utvikling.
14. Webapplikasjonen skal være brukervennlig.

6 Overordnet arkitektur

Det er oversiktlig å vite hvordan systemet skal se ut for ha en kontekst på vurderingene og refleksjonene som har blitt gjort. Dette kapittelet gir derfor leseren en oversikt over systemet som er blitt utviklet. I Figuren under kan man se den overordnede arkitekturen, som baserer seg på mikrotjenester. Rektanglene representerer ulike tjenester som inngår i systemet. Pilene indikerer hvilken retning kommunikasjonen går, ikke alle systemer skal kunne kommunisere med hverandre begge veier.



Figur 5: Overordnet arkitektur.

Systemet består følgelig av tre tjenester: API (1.0), system for innhenting av informasjon (2.0) og en Webapplikasjon (3.0). Det er valgt å splitte systemet ut i tre tjenestene for å imøtekomme krav 10, da dette gir mulighet for å skalere ulike deler uavhengig av hverandre.

APIet sitt ansvar er å håndtere flyten mellom tjenesten for innhenting av informasjon, webapplikasjonen og databasen. Det er valgt å skille dette ut til en egen tjeneste slik at dataen i APIet også kan benyttes ved bruk av eksterne tjenester, eksempelvis Power BI. Webapplikasjonen er brukergrensesnittet for brukerens interaksjon med systemet, og skal kun kommunisere med systemets API.

For å hente data fra Tytlandsvik Aqua sine systemer er det valgt å separere dette ut i en egen tjeneste for at det skulle være mulig å kjøre denne ulike steder, da det i starten var en mulighet for at denne måtte kjøres on-premises (lokalt på anlegget) av sikkerhetshensyn. Dette gir samtidig fleksibilitet i skalering av denne tjenesten uavhengig av APIet, samtidig som det enkelt kan dupliseres dersom data skal hentes fra flere eksterne systemer.

Anlegget har allerede flere databaser, men det er valgt å benytte en separat database som APIet konsumerer. Dette er fordi deres databasesystemer har begrenset kapasitet samt dette gir muligheten til en mer fleksibel og strukturert datastruktur. Samtidig kan man benytte seg av mer moderne teknologier, eksempelvis sikkerhetskopiering av data svært ofte og automatisk optimalisering.

7 Teori og teknologivalg

I dette kapittelet presenteres relevant teori og hvilke ulike teknologier som er valgt, samt begrunnelsen bak valget av disse.

7.1 Arbeidsmetodikk

En forutsetning for å imøtekomme kravene er en god overordnet arbeidsmetodikk. Det er særlig en standard som er relevant i moderne utvikling og som vurderes aktuell til utviklingen av dette systemet. Dette består av en smidig utviklingsprosess, Scrum og Kanban-tavle.

7.1.1 Smidig utviklingsprosess

Smidig (eng.: agile) utviklingsprosess er en velkjent utviklingsprosess innen programvareutvikling der hovedpoenget er å raskt respondere på endringsbehov. Dette vil si at det ikke må foreligge en fullverdig kravliste der alt er planlagt på forhånd, men at det kun er overordnede krav. Slik vil det kunne opereres med en mer dynamisk kravliste som endres underveis ut fra hva som fungerer bra og ikke, samt nye eller endrede behov.

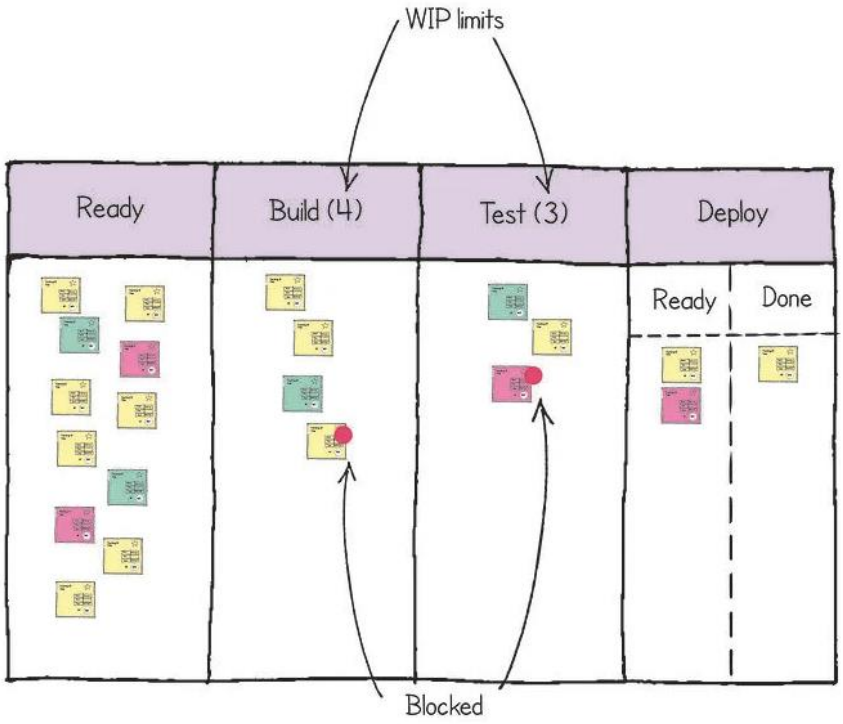
Det ble konkludert med å benytte smidig utviklingsprosess for å sikre et effektivt og fleksibelt samarbeid med Tytlandsvik Aqua. Kravlisten til systemet ble utviklet i en slik prosess.

7.1.2 Scrum

Scrum er et rammeverk innen smidig programvareutvikling som skal gjøre utviklingsprosessen lettere og mer oversiktlig (Nes, 2019). Rammeverket er bestående av tre grunnleggende roller: Produkteier (ofte kunden i en konsulentvirksomhet), utviklingsteamet og Scrum masteren. Metodikken bygges rundt en gjentakene prosess kalt spurter (eng.: sprints), der hvert medlem i et team tar for seg oppgaver som skal bli ferdigstilt i løpet av spurten. Oppgavene blir hentet fra produkt-backloggen, som er en estimert liste over funksjonelle og ikke-funksjonelle krav som det har blitt utledet i samarbeid med produkteier. Scrum masteren sørger for at teamet opprettholder Scrum prinsipper.

7.1.3 Kanban-tavle

Kanban-tavle er et visuelt verktøy som er et godt supplement i en smidig utviklingsprosess (Hva er Kanban?, 2019). Det er en tavle som er delt opp i kolonner, der hver kolonne er et steg i utviklingsprosessen til en spesifikk oppgave. Typisk praksis er at i starten av en spurt velger man ut oppgaver fra backloggen som skal utvikles. Når utvikler er ferdig med oppgaven går den videre til testing, for deretter å bli ferdigstilt og rullet ut i produksjon. For at arbeidslasten på hvert steg ikke skal blir for stor velger man ofte å sette begrensninger på hvor mange oppgaver som kan være i hver kolonne.



Figur 6: Eksempel på Kanban-tavle.

Hentet fra: https://www.prosjektbloggen.no/hva-er-kanban (09.02.21).

7.2 Versjonshåndtering

For å ha kontroll over kildekode og versjonering benyttes versjonshåndtering.

7.2.1 Git

Git er et åpen-kildekode verktøy for å håndtere filendinger (Git, 2021). Man kan ved å bruke Git få full historikk over endringene som har blitt utført i filstrukturen og på de enkelte filene. Man kan også separere endringer ut fra den originale koden for deretter å sette koden sammen igjen. Dette er essensielt for å håndtere kildekode med hyppige endringer der flere jobber sammen.

7.2.2 GitHub

GitHub er en nettbasert tjeneste for å håndtere bruk av Git mellom flere brukere over nettet (GitHub Inc., 2021). Man kan oppbevare kildekoden sin på GitHub som videre gjør at den kan deles med brukere og ikke-brukere av tjenesten. GitHub gir også tilgang til en rekke andre tjenester som forenkler informasjonsflyten mellom utviklere, eksempler på dette er *GitHub issues* og *GitHub projects* som har støtte for Kanban-tavle. GitHub tilbyr svært mange av sine tjenester helt gratis som også kan brukes i forbindelse med kontinuerlig integrasjon og -utrulling.

Det ble valgt å bruke GitHub ettersom vi har god kjennskap og positive opplevelser til denne tjenesten fra andre prosjekter. Eksempler på andre tilsvarende tjenester, som ikke ble benyttet, er GitLab og Azure DevOps.

7.2.3 Semantic Release

Semantic Release er et verktøy for å håndtere versjonsnummeret til programvare basert på Git meldinger (Semantic Release, 2021). Det finnes ulike prefikser som kan brukes på meldingen for at Semantic Release skal endre på versjonering. Semantic Release verktøyet bruker en versjonering som kalles for Semantic Versioning. Konvensjonen på denne versjoneringen er **a.b.c** der **a** er store endringer, ofte det som kalles for ødeleggende endringer (eng.: breaking changes), endringer som bryter med bakover-kompatibiliteten. **b** er ny funksjonalitet mens **c** er små fikser på eksisterende løsning. Semantic Release støtter også mer spesialisert versjonering. Ettersom systemet ikke er veldig stort med svært mange utviklere ser vi ikke noe poeng i å gjøre ting for komplisert.

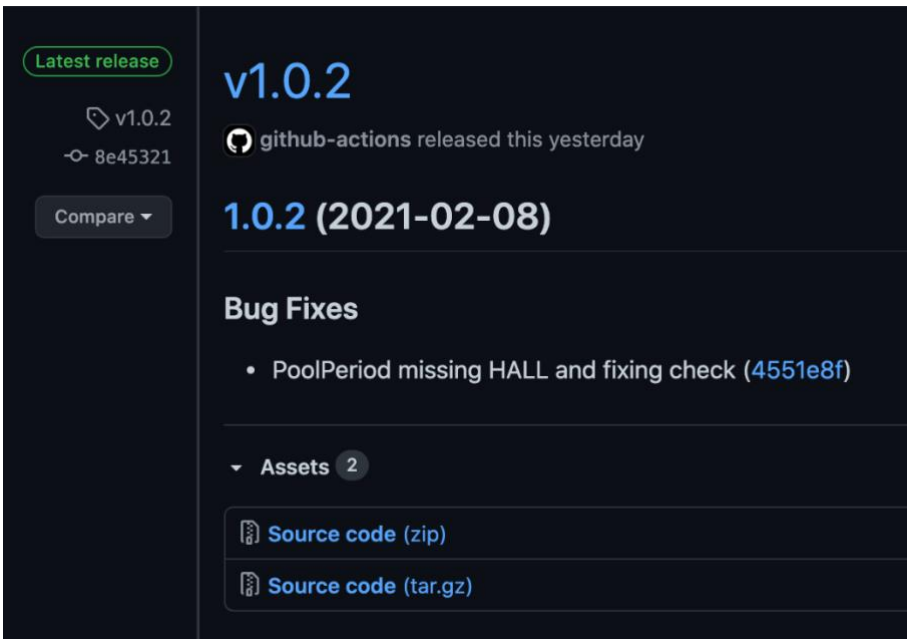
Semantic Release går gjennom Gitloggen og ser på meldingene som finnes der. Dersom man har prefikset med *fix* inkrementeres **c** med én. Prefiks med *feat* inkrementerer **b** med én og nullstiller **c**. Med bruk av *BREAKING CHANGE* inkrementeres **a** med én samtidig som **b** og **c** nullstilles. Det finnes flere prefikser man kan bruke, men det er disse vi har brukt i denne oppgaven. Under kan man se eksempelet som er brukt i Semantic Release dokumentasjonen.

Commit message	Release type
`fix(pencil): stop graphite breaking when too much pressure applied`	Patch Release
`feat(pencil): add 'graphiteWidth' option`	~~Minor~~ Feature Release

`perf(pencil): remove graphiteWidth option`
 	
`BREAKING CHANGE: The graphiteWidth option has been removed.`
`The default graphite width of 10mm is always used for performance reasons.`
	~~Major~~ Breaking Release

Figur 7: Semantic Release prefikser fra dokumentasjon.
Hentet fra: https://github.com/semantic-release/semantic-release (09.02.2021).

Semantic Release lager også en endringslogg man kan studere i GitHub. Nedenfor kan man se et skjermbilde av hvordan endringsloggen blir seende ut i GitHub.



Figur 8: Eksempel på endringslogg fra GitHub

Det er valgt å benytte seg av versjonering for å få en god oversikt over systemets utvikling. Eksempelvis: Dersom systemet plutselig slutter å fungere er det enkelt å gå tilbake til en versjon man vet fungerte, man kan også enkelte å se hvilke endringer som innførte feilen.

7.3 Mikrotjenester

Mikrotjenester er en metode man kan bruke for å strukturere applikasjoner som en samling av mindre tjenester. Dette gjør at man får lav avhengighet mellom tjenestene. Metoden vil også gjøre det enklere å arbeide på de ulike delene av en applikasjon. I tillegg blir det enkelt å skalere opp ved å legge til ekstra mikrotjenester.

Vi har valgt å strukturere applikasjonen som et knippe med mikrotjenester for å enkelt kunne bytte ut deler av tjenesten i ettertid og for at applikasjonen skal være skalerbar, som er et krav i oppgaven (Krav 10).

7.3.1 Beholdere

Mikrotjenester blir ofte delt opp i ulike beholdere (eng.: containers). Beholdere inneholder en standard pakke med programvare som har med seg alle avhengighetene programvaren trenger for å kjøre (Docker Inc., 2021). Ved å bruke beholdere vil man kunne kjøre tjenestene hvor som helst uavhengig av hvordan man har konfigurert systemet som beholderen kjøres i. En beholder har et bilde, et beholder-bilde (eng.: conatiner image), som inneholder alt man trenger for å kjøre programvaren. Dette bildet kan man kjøre opp som en beholder i for eksempel beholder-tjenesten Docker.

7.3.2 Docker

Grunnlaget har nå blitt lagt, man vet hva en applikasjon som mikrotjeneste er samt hva en beholder er. Docker er et selskap og en platform-som-tjeneste (eng.: PaaS, Platform as a Service) for å levere programvare i pakker også kjent som beholdere (Docker Inc., 2021). Alle beholdere kjøres av et operativsystem i én kjerne som gjør at tjenesten bruker mindre ressurser enn andre former for virtualisering. Docker-beholdere kan også kommunisere med hverandre.

For å lage et Docker-bilde som kan kjøres i en Docker-beholder er man nødt til å definere en såkalt Dockerfil. Dette gjøres på følgende måte: Man oppretter en fil i roten av prosjektet som heter *Dockerfile*. På neste side er et eksempel fra det utviklede systemet på hvordan dette er gjort for den ene mikrotjenesten. Man trenger ikke forstå hva selve kommandoene som bygger og kjører applikasjonen gjør, dette er kun for å vise hvordan en Dockerfil kan se ut.

```
1. # Definerer Docker-bildet som skal anvendes for å bygge koden
2. FROM mcr.microsoft.com/dotnet/sdk:5.0-alpine AS builder
3.
4. # Definerer mappen som skal brukes som arbeidsmappe.
5. WORKDIR /sln
6.
7. # Definerer hva som skal kopieres over til Docker-bildet. Her
   kopieres alt.
8. COPY ./FishyAPI .
9.
10.  # Kommandoen som må kjøres for å bygge prosjektet
11.  RUN dotnet publish -c Release -o /sln/artifacts
12.
13.  # Her definers Docker-bildet som inneholder run-time av
   rammeverket.
14.  FROM mcr.microsoft.com/dotnet/aspnet:5.0
15.
16.  # Ny arbeidsmappe defineres
17.  WORKDIR /app
18.  # Man definerer hvilken kommando som skal kjøres når docker-
   beholderen starter
19.  ENTRYPOINT ["dotnet", "FishyAPI.dll"]
```

.NET docker prosjekt bygg. Hentet fra /Dockerfile.production i API prosjektet.

Når man bygger slike bilder kan man også tagge de med versjonsnummer eller navn som gjør at man enkelt kan knytte bildene til bestemte miljø.

Grunnen til at det er valgt å kjøre tjenestene som Docker-beholdere er at dette gjør tjenestene fleksible til å kunne kjøre nesten hvor som helst (Docker Inc., u.d.). Dersom man velger å bytte skyleverandør kan man enkelt kjøre dem opp som en container der. Man kan også kjøre dem på lokale servere, det eneste man behøver av programvare er Docker. Docker støtter også de mest populære operativsystemene, som Windows, MacOS og mange andre Linux-baserte operativsystemer som f.eks. CentOS som er mye brukt på servere.

7.4 Miljøvariabler

En miljøvariabel er variabler som endrer måten programvare oppfører seg i et miljø. Miljøvariablene er altså en del av miljøet hvor en prosess kjører. Variablene settes på utsiden av selve programvaren. Det er lurt å bruke miljøvariabler der man har differanser i programvaren basert på hvilket miljø det kjører i. Dersom et program som kjører i et test-miljø bruker noen andre nøkler enn et program som kjører i produksjons-miljø er dette en god kandidat til å gjøres om til en miljøvariabel.

7.5 Skytjenester

Skytjenester er et samlebegrep på programvaretjenester som kjører på andre aktørers servere som har blitt tilgjengeliggjort fra eksterne serverparker over internett (Datatilsynet, 2018). Ved å benytte skytjenester åpner man opp for skalering og minimerer driftskostnadene, noe som tilfredsstiller krav 10. Alternativ kunne man valgt å kjøre tjenesten lokalt, men dette ville ikke tilbudt like mye funksjonalitet rett ut av boksen, samt det ville vært mer kostbart å opprettholde for Tytlandsvik Aqua. Dette kapittelet vil ta for seg valg av skytjeneste, samt tjenestene til skytjenesten vi benytter oss av.

7.5.1 Aktører

Det er flere store aktører når det kommer til leveranse av skytjenester. Eksempler på detter er Amazons AWS, Google Cloud og Microsofts Azure. Disse aktørene leverer svært like tjenester i omtrent samme prisklasse.

7.5.2 Azure

Det er valgt å bruke Azure grunnet flere årsaker. Det foreligger blant annet erfaring i gruppen med Azure fra tidligere prosjekter. I tillegg benytter Tytlandsvik Aqua en leverandør av nettjenester som heter Hesbynett, som også har kompetanse innenfor Microsofts tjenester. Microsoft tilbyr svært god dokumentasjon tilhørende sine tjenester i Azure med god support mulighet dersom noe ikke fungerer som forventet. Azure tilbyr svært mange tjenester, i dette prosjektet blir de to følgende benyttet.

Azure AppServices

Azure AppServices er en tjeneste for å kjøre web-applikasjoner på en enkel måte (Microsoft Azure, 2021). Man kan kjøre de fleste nyere rammeverkene eller kjøre applikasjonen som en

Docker-container. Azure AppServices kommer med mange gode innebygde tjenester som brannmur, applikasjonslogging, automatisk utrulling og mye mer. Man kan også konfigurere AppServicen til å skalere etter behov slik at man alltid har nok kraft til å tjene alle forespørsler selv når det er store behov. Prisene varierer fra gratis til flere tusen i måneden.

Azure SQL

Azure SQL Database er en databasetjeneste med mange innebygde intelligente tjenester for økt sikkerhet og ytelse (Microsoft Azure, 2021). Noen av disse tjenestene inkluderer smart-indeksering, kryptering, automatisk sikkerhetskopiering, brannmur og mye mer. Prisene varierer fra gratis til flere titalls tusen i måneden.

7.6 Websockets

Websocket er en kommunikasjonsprotokoll som tillater ende til ende kommunikasjon over en TCP-kobling (Cisco Networking Academy, 2013). Denne teknologien er det man kaller for full-dupleks som vil si at man kan sende kommunikasjon begge veier samtidig. Websocket blir brukt for sanntidskommunikasjon mellom klient og server.

7.7 Kontinuerlig integrasjon/-utrulling (CI/CD)

Kontinuerlig integrasjon og -utrulling er prosessen med å bygge, teste og utgi programvare kontinuerlig. Denne prosessen kan automatiseres ved å benytte ulike verktøy og tjenester. Fordeler ved å en slik automatikk er at miljøene alltid forblir like i hvert bygg, med blant annet like miljøvariabler og versjoner på tredjepartteknologier. I tillegg er det tidsbesparende og sikrer at bygget og utrullingen går gjennom de samme stegene hver gang.

7.7.1 Github Actions

Github Actions er en tjeneste man kan bruker for å gjøre automatiserte oppgaver på kodebasen når man laster opp kode til GitHub. Ved å bruke github actions kan man f.eks. automatisere bygging av prosjekt og utrulling til tjener-tjenester. Man strukturere slike automatiseringer i en .yml fil som blir liggende under /.github/workflows i rot mappen til prosjektet. Github vil da automatisk finne oppsettet.

Eksempel på oppsett:

```
1. # Spesifiserer navn på jobben
2. name: Heisann Verden
3.
4. # Når man dytter ut kode til main branchen vil denne jobben kjøres
5. on:
6.   push:
7.     branches: main
8.
9. jobs:
10.    main:
11.      # Hvilken type operativsystem man ønsker å kjøre på, windows
   eller linux eller som her ubuntu.
12.      # "Latest" beskrivet versjonsnummeret som her sier at man skal
   bruke siste verjson.
13.      runs-on: ubuntu-latest
14.
15.      # "Steps" er et steg i prosessen, man kan ha så mange slike man
   ønsker.
16.      # Her skriver man ut "Heisann verden", man kan kjøre alle typer
   kommandoer her som er støttet av det valgte operativsystemet.
17.     steps:
18.      - name: print Heisann Verden
19.        run: echo "Heisann Verden"
```

Selvskrevet eksempel på en Githup-actions .yml fil (15.02.21).

7.7.2 Github container registry

Github container registry er en tjeneste som kan benyttes for å lagre Docker-bilder (Github Inc., 2021). Docker-bildene blir lagret bak autentisering slik at de er sikret mot uautorisert tilgang og bruk. Man kan også få en oversikt over Docker-bildene i GitHub og knytte dem til et oppbevaringssted som ligger i GitHub.

7.7.3 Azure Webhooks

Azure Webhooks kan brukes til å utløse hendelser i Azure. I vårt tilfelle brukes denne tjenesten til å utløse automatisk utrulling til Azure Appservices fra GitHub Actions.

7.7.4 GitHub Dependabots

Denne tjenesten fra GitHub oppretter forslag til endringer i prosjektet når det kommer nye versjoner av tredjepartspakker som er brukt. På denne måten kan man opprettеholder de nyeste versjonene samtidig som man får lukket sikkerhetshull.

7.8 Frontend

Frontend er webapplikasjonen som vil bli vist i nettleseren hos brukeren. Dette gjøres ved at brukeren sender en forespørsel til en webserver ved å gå inn på en nettside, og webserveren responderer med filer som nettleseren kan lese og presentere en nettside ut ifra.

7.8.1 Grunnleggende webteknologier

For å få en god forståelse over rammeverkene som er brukt i denne oppgaven, vil det i dette delkapittelet bli skrevet kort om de tre grunnleggende språkene innen webutvikling som rammeverkene er bygget opp av: HTML, CSS og Javascript.

HTML

HTML (Hypertext Markup Language) er et markeringsspråk som leses av nettlesere. Språket er bygget opp av tagger som strukturerer selve dokumentet og nettsiden (WHATWG, u.d.).

CSS

CSS (Cascading Style Sheets) er et kodespråk som brukes til å definer utseende på nettsider. Dette språket blir brukt som et supplement til HTML ettersom HTML i seg selv ikke definerer noe mer enn grunnleggende struktur til nettsiden (W3C, 2021). CSS kan skrives rett inn i HTML tagger ved, bruk av attributten *style*. Det kan og skrives i en egen *style* tagg der elementene i dokumentet kan refereres. Eventuelt kan det skrives i et eget dokument, noe som ofte er det mest ryddige valget.

JavaScript

JavaScript er et programmeringsspråk som ofte står for den funksjonelle biten i en nettside, men kan også brukes som et skriptspråk til andre formål enn nettsider (W3C, 2016). Språket har dynamiske datatyper som konverteres under kjøring, noe som kan støte på uventede feil ved kjøretid. JavaScript kunne originalt bare kjøres i nettlesere, men kan nå kjøres gjennom andre kjøretidssystemer som Node.js. Node.js vil bli skrevet mer om i 7.8.2 Node/NPM.

JSON

JSON (JavaScript Object Notation) er en enkel tekstbasert standard for å formatere dokumenter (json, u.d.). Dette er en formatering som vil bli mye brukt gjennomgående i

bacheloren. Videre følger et eksempel som viser hvordan man kan formatere data med ulike datatyper, lister og nøstede objekter.

```
{
  "brukernavn": "example@email.com",
  "erAdmin": false,
  "favorittfrukter": ["eple", "banan", "pære"],
  "favorittbil": {
    "merke": "DeLorean",
    "modell": 1
  }
}
```

Eksempel på JSON struktur.

TypeScript

Typescript er en utvidelse av JavaScript som legger til valgfri sterk typet funksjonalitet (typescriptlang, u.d.). Når Typescript benyttes, må koden kompileres til JavaScript før den kjøres. Denne prosessen er ofte innebygd i rammeverk og er sjeldent et hinder for å velge typescript.

Ved å ha typer på variabler, parametere og funksjoner vil man få en mye mer robust kode i forhold til å kun bruke JavaScript. Det blir altså lettere å utvikle da man i IDEen får tilbakemeldinger på hva som for eksempel returneres av en funksjon i tillegg vil flere av kjøretidsfeilene (eng.: runtime errors) bli fanget opp. Dette er hovedgrunnen til at det i denne oppgaven har valgt å bruke Typescript fremfor bare JavaScript.

7.8.2 Node/NPM

Node.js er et kjøretidssystem for server- og nettverksapplikasjoner som kjører på Chrome v8 motoren (Patel, 2018). Dette kjøretidssystemet gjør det mulig å kjøre JavaScript utenfor en nettleser, noe som gjør at JavaScript nesten brukes i alle slags typer applikasjoner.

NPM (Node Package Manager) er en pakkebehandler for JavaScript (w3schools, u.d.). En pakkebehandler kan kort sakt beskrives som en tjeneste som behandler alle tredjeparts biblioteker som blir brukt i applikasjonen, samt versjonene av disse. Ved bruk av Node.js blir det laget en fil ved navn *package.json* som inneholder navn og versjon av alle pakkene til applikasjonen. Ved bruk av kommandoen *npm install* vil alle pakkene bli lastet ned og installert i en egen mappe ved navn *node_modules*, samt det vil bli generert en *package-lock.json* fil som inneholder metadata om disse pakkene. Denne kommandoen må kjøres før

en Node.js applikasjon kan kjøres, så lenge det er brukt tredjeparts biblioteker. I *package.json* filen er det også en *script* attributt der men kan definere diverse kommandoer for applikasjonen, typisk startkommandoer og bygge kommandoer.

7.8.3 React

React.js er et JavaScript-bibliotek for å bygge brukergrensesnitt (React.js, u.d.). Det vedlikeholdes av Facebook, men er åpenkildekode slik at flere kan bidra med utviklingen. Biblioteket kan brukes som en SPA (Single Page Application), som vil si at nettsiden samhandler med brukeren ved å dynamisk skrive om nåværende side istedenfor at nettleseren laster inn nye sider. Dette gjør at overganger mellom sider og henting av ny data føles raskere og mer ut som er native applikasjon.

React.js kan enkelt brukes ved å legge til CDN (Content Delivery Network) linker fra React i en HTML-side, eller så kan du initialisere en ny React app med en enkel kommando: *npx create-react-app name-of-app.* Det eneste kravet på sistnevnte er at Node.js og NPM er installert.

Kort sagt så fungerer React.js ved at du har en HTML-side med en *div*-tag med en id som React.js bruker til å skrive om HTML siden dynamisk. For at React.js skal få tilgang til *div*-taggen brukes *ReactDOM.render()* metoden. Første parameteren til *ReactDOM.render()* metoden er innholdet som skal vises gjennom taggen det refereres til. Dette innholdet skrives ved hjelp av React-komponenter.

```
1. ReactDOM.render(
2.   <h1>Hello, world!</h1>,
3.   document.getElementById('root')
4. );
```

Enkelt eksempel av ReactDom.render() funksjonen. Kodesnutt hentet fra: https://reactjs.org/docs/hello-world.html (15.02.21).

Det ble valgt å bruke React.js ettersom det er et anerkjent og populært rammeverk, i tillegg så har vi tidligere erfaringer med det. React.js gjør også utviklingen en del mer effektiv fremfor vanlig Javascript, CSS og HTML.

Komponenter

Komponenter i React gjør det mulig å splitte grensesnittet opp i uavhengige og gjenbrukbare biter (React.js, u.d.). Det finnes funksjonsbaserte og klassebaserte komponenter, men i dette prosjektet er det valgt å bruke funksjonsbaserte komponenter ettersom det er den nye standarden. Disse funksjonene returnerer *JSX* kode, som er en syntaks utvidelse til JavaScript som produserer React elementer som kan vises på en HTML-side (React.js, u.d.). Tilstandsvariabler (eng.: state) i funksjonsbaserte komponenter er variabler som trigger en rerender av komponenten når dem endrer seg. Ved rerender menes det at komponenten bygges på nytt, slik at grensesnittet viser ny data til brukeren.

Hooks

I dette prosjektet er det også valgt å lage egendefinerte hooks, samt bruke ferdigdefinerte hooks fra React-biblioteket. *Hooks* er en relativ ny funksjonalitet i React som tillater funksjonsbaserte komponenter å ha samme funksjonalitet som klassebaserte komponenter, som for eksempel tilstandsvariabler (React.js, u.d.).

Next.js

Next.js er et React-rammeverk som muliggjør serverside-rendering (eng.: Server Side Rendering). Dette vil si at nettleseren henter HTML-sider fra serveren i motsetning til at det bare er Javascriptfiler som oppdaterer Webapplikasjonen.

Fordeler med serverside-rendering:

- SEO (Search engine optimization) vennlig.
- Bruker vil se innhold raskere.
- Nettleser gjør mindre prosessering.
- Enklere og raskere utvikling av navigering.

Ulemper med serverside-rendering:

- Kan degradere ytelsen dersom applikasjonen er stor, eller hvis serveren er travel. Dette gjelder spesielt ved sideskifter.
- Kan øke kompleksiteten til applikasjonen.

(TEK-Tools, 2020)

Next.js bruker teknisk sett noe om heter *universal Rendering* (TechStacker, 2020), som vil si at rammeverket bruker det beste av begge verdener, klientside-rendering og serverside-rendering for best mulig ytelse totalt sett. Next.js blir derfor en mye mer komplett og fleksibel løsning fremfor å kun bruke klassisk React.js, noe som veide tung da vi valgte å gå for Next.js.

7.8.4 Sidekart

Et sidekart er oversikt over hvilke sider som finnes i Webapplikasjonen (Osman, 2020). Det er flere grunner til at det kan være lurt å ha med et sidekart. Ved utforming av en trusselmodell, forklart i kapittel 7.10.3 Trusselmodellering, vil et sidekart som beskriver hvilke roller som kreves være behjelpelig. Et sidekart er også fint til andre senarioer hvor man vil ha en rask oversikt over de ulike delene av nettsiden, for eksempel for nye kunder. For god brukervennlighet er det anbefalt at sidekartet har en maksimal dybde på tre. Det vil si at man kommer til alle sidene ved tre klikk.

7.8.5 Tredjepartsbiblioteker til frontend

Det er benyttet noen tredjepartspakker i Webapplikasjonen, deres funksjon blir kort beskrevet her. Pakker som har en stor betydning for systemet, blir gjennomgått i nærmere detalj. Det har vært fokus på at pakkene skal ha lisenser som tilsier at dem kan brukes fritt.

7.8.5.1 Material-UI

- Lisens: MIT
- Dokumentasjon: https://material-ui.com/

Material-UI er et React-komponentbibliotek som baserer seg på Material design, en designstandard som er vedlikeholdt av Google. Komponentene kan brukes som standard React-komponenter, der utseende på dem kan enkelt endres og standardiseres.

Dette prosjektet har valgt å bruke Material-UI ettersom dette er det mest populære brukergrensesnitt rammeverket for React (Moor, u.d.), og det er enkelt å få opp komponenter som ser bra ut. Denne pakken gjør det også enklere for oss å gjøre applikasjon mer brukervennlig, noe som tilfredsstiller krav 14. En ulempe kan være at ettersom biblioteket baserer seg på Material design, så kan det ligne veldig på flere andre applikasjoner. Ettersom

vi har fokusert mer på funksjonalitet fremfor et unikt design så har vi ikke sett på det som en ulempe for dette prosjektet.

7.8.5.2 Apexcharts & react-apexcharts

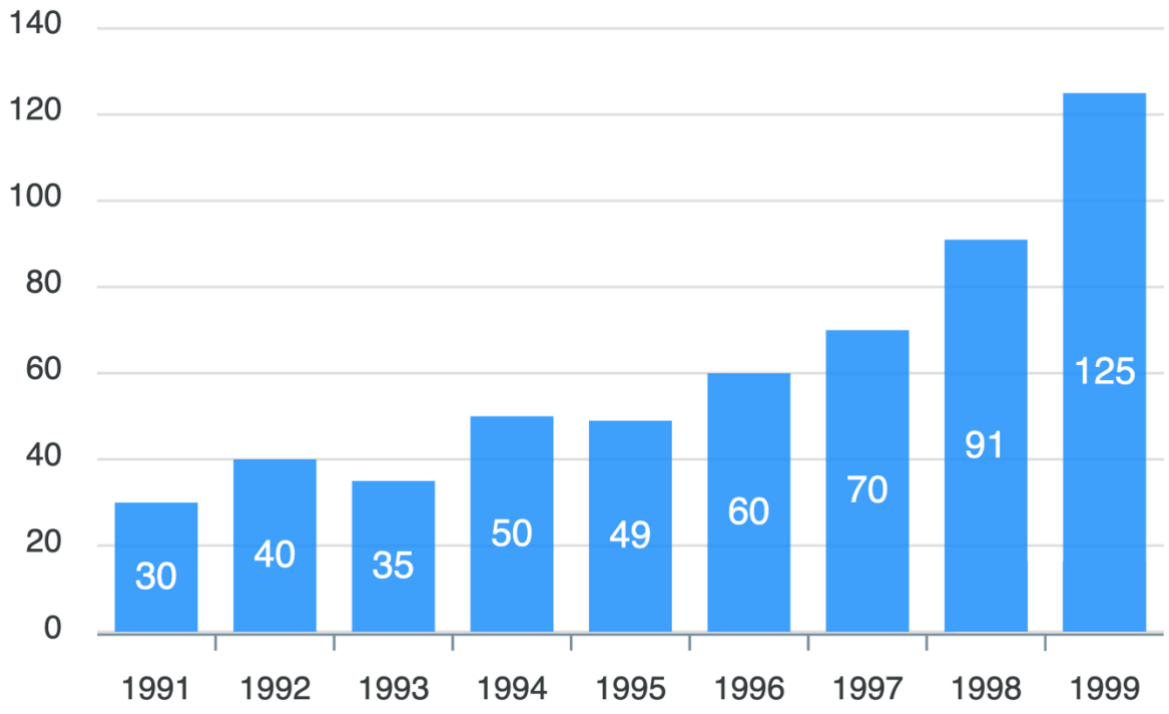
- Lisens: MIT
- Dokumentasjon:
  - Apexcharts: https://apexcharts.com/docs/installation/
  - react-apexcharts: https://www.npmjs.com/package/react-apexcharts

For å tilfredsstille krav 1 om visualisering av data, måtte det tas i bruk et grafbibliotek. Apexcharts er en åpenkildekode-pakke som tilbyr interaktive grafer. Data til grafene blir lagt inn ved attributtet *series* som er en liste med data, og innstillingene til grafene er veldig fleksible. React-apexcharts er en React.js basert innpakning (eng.: wrapper) av biblioteket, og begge må legges til i prosjektet for bruk.

```
1. class App extends Component {
2.   constructor(props) {
3.     super(props);
4.     this.state = {
5.       options: {
6.         chart: {
7.           id: 'apexchart-example'
8.         },
9.         xaxis: {
10.            categories: [1991, 1992, 1993, 1994, 1995, 1996, 1997, 19
   98]
11.          }
12.        },
13.        series: [{
14.          name: 'series-1',
15.          data: [30, 40, 45, 50, 49, 60, 70, 91]
16.        }]
17.      }
18.    }
19.    render() {
20.      return (
21.        <Chart options={this.state.options} series={this.state.series
   } type="bar" width={500} height={320} />
22.      )
23.    }
24.  }
```

Eksempel av en React.js komponent som bruker Apexchart. Kodeutsnitt hentet fra: https://www.npmjs.com/package/react-apexcharts (15.02.21).

Kodeutsnittet gir følgende graf:



Figur 9: Skjermbilde for eksempelkode

Eksempel hentet fra: https://www.npmjs.com/package/react-apexcharts (27.02.21).

Andre pakker tilbyr stort sett samme funksjonalitet, men vi opplevde at denne pakken var enklest å jobbe med ettersom den har god dokumentasjon og er svært fleksibel. I tillegg kan den lett oppgraderes ved å betale for FusionCharts som har enda flere funksjonaliteter. (Fusion Charts Docs, u.d.)

7.8.5.3 Microsoft/signalr

- Lisens: Apache License 2.0
- Dokumentasjon: https://www.npmjs.com/package/@microsoft/signalr

Denne pakken blir brukt som klientside av SignalR tjenesten som er utviklet i APIet. SignalR kan bruke websockets for at appen kan få inn sanntidsdata uten noen samhandling fra brukeren eller automatiske kall til APIet, noe om imøtekommer krav 3. Denne pakken er utviklet av Microsoft (noe som SignalR og .Net også er) og er derfor et naturlig valg for dette bruket.

Nedenfor ser man hvordan man kan koble seg til backend-tjenesten, samt få oppdateringer fra en gitt metode:

```
1. let connection = new signalR.HubConnectionBuilder()
2.     .withUrl("/chat")
3.     .build();
4.
5. connection.on("send", data => {
6.     console.log(data);
7. });
```

Kodesnutt hentet fra: https://www.npmjs.com/package/@microsoft/signalr (15.02.21).

7.8.5.4 Axios

- Lisens: MIT
- Dokumentasjon: https://www.npmjs.com/package/axios

Axios er en pakke før å utføre HTTP (Hypertext Transfer Protocol)-forespørsler. Det ble valgt å bruke Axios fremfor *fetch* metoden, som er støttet direkte i nettlesere, ettersom Axios er enklere å bruke og man slipper å konvertere responsen til et JSON-objekt. Axios har også innebygd CSRF (cross-site request forgery) beskyttelse (verma_anushka, 2020).

7.8.5.5 Easy-peasy

- Lisens: MIT
- Dokumentasjon: https://easy-peasy.now.sh/

Easy-peasy blir brukt for håndtering av global tilstand (eng.: state) i applikasjonen. Pakken er en abstraksjon av Redux. Redux er en annen pakke for tilstands-håndtering som er mer kjent, men er mye kritisert for sin kompleksitet og overhead (Yosef, 2017). Tilstands-håndtering er et stort tema innenfor de fleste frontend-rammeverk, så her er det mye å velge mellom. Det ble valgt å bruke Easy-peasy ettersom det var enkelt og raskt å ta i bruk, samt det er en anerkjent pakke som blir brukt av flere.

Easy-peasy blir brukt ved å lage en *store* med metoden *createStore()*. Denne storen er bygget opp av tre hovedkomponenter: variabler (fungerer som tilstand), *actions* (funksjoner som endrer på tilstanden) og *thunks* (som gir mulighet for asynkrone funksjoner). En action tar inn

state og *payload* som argumenter, mens en *thunk* tar inn *actions* og *paload* som argumenter. En thunk må altså kalle på en action for å endre på tilstanden.

```
1. const store = createStore({
2.    todos: ['Create store', 'Wrap application', 'Use store'],
3.    addTodo: action((state, payload) => {
4.       state.todos.push(payload);
5.    }),
6. });
```

Eksempel på initialiseringen av en store. Kodesnutt hentet fra https://easy-peasy.now.sh/ (16.02.21).

For å kunne få tilgang til storen inne i appen, må man legge komponentene som trenger storen inni en *StoreProvider* komponent. Normalt sett legger vi hovedkomponenten til applikasjonen inni denne komponenten, slik at man har tilgang til storen overalt.

```
1. function App() {
2.   return (
3.     <StoreProvider store={store}>
4.       <TodoList />
5.     </StoreProvider>
6.   );
7. }
```

Eksempel på bruk av StoreProvider komponenten. Kodesnutt hentet fra https://easy-peasy.now.sh/ (16.02.21).

For å bruke en store inne i en komponent, henter man ut state og actions ved funksjonene *useStoreState* og *useStoreActions*. Her spesifiserer man hvilken tilstand eller action/thunk man trenger. Man kan da bruke storen som vanlige variabler/funksjoner inne i komponenten.

```
1. function TodoList() {
2.   const todos = useStoreState((state) => state.todos);
3.   const addTodo = useStoreActions((actions) => actions.addTodo);
4.   return (
5.     <div>
6.       {todos.map((todo, idx) => (
7.         <div key={idx}>{todo}</div>
8.       ))}
9.       <AddTodo onAdd={addTodo} />
10.     </div>
11.   );
12. }
```

Eksempel på hvordan en store kan bli brukt i en komponent. Kodesnutt hentet fra https://easy-peasy.now.sh/ (16.02.21).

7.8.5.6 dotenv

- Lisens: BSD-2-Clause
- Dokumentasjon: https://www.npmjs.com/package/dotenv

Dotenv er en pakke som gjør det mulig å laste inn miljøvariabler fra en *.env*-fil til koden. For bruk må denne linjen legges til tidligst mulig:

```
1.  require('dotenv').config()
```

Når den er lagt til kan man få tak i egendefinerte miljøvariabler ved bruk av *process.env.<navn på variabel>*. Miljøvariabler kan blir brukt for å skjule sensitiv informasjon som for eksempel databaseinnlogging. Det er også er støtte for å ha særegne verdier på variablene ut ifra hvilket miljø systemet kjøres i, eksempelvis som test og produksjon, noe som er hovedgrunnen til at det blir brukt i oppgaven.

7.8.5.7 Env-cmd

- Lisens: MIT
- Dokumentasjon: https://www.npmjs.com/package/env-cmd

Env-cmd er en enkel Node.js-pakke for å kjøre applikasjonen ved gitte miljøvariabel-filer (.env filer). Denne pakken blir brukt til nettopp det, nemlig å kjøre Webapplikasjoner på forskjellige miljøer med egendefinerte miljøvariabler.

7.8.5.8 Framer-motion

- Lisens: MIT
- Dokumentasjon: https://www.npmjs.com/package/framer-motion

Framer-motion er en animasjonspakke som kan gi enkle animasjoner til HTML elementer. Pakken blir brukt for animasjon mellom sider, samt en mer livlig innloggingsform. Å bruke animasjoner er ikke et krav fra kunden, men vi har valgt å ta med enkle animasjoner ettersom det gjør løsningen mer levende og interessant å bruke.

7.8.5.9 Husky (v4)

- Lisens: MIT
- Dokumentasjon https://www.npmjs.com/package/husky

Husky er en pakke som gjør at man kan kjøre kommandoer i forbindelse med *Git hooks*. Hooks er programmer du kan plassere i en *hooks* mappe for å trigge handlinger på en gitt posisjon i *Gits* handlinger (Git, 21).

7.8.5.10 Material Table

- Lisens: MIT
- Dokumentasjon https://material-table.com/#/

Denne pakken baserer seg på Material-UI sin tabellkomponent, men har noen ekstra funksjoner rett ut av boksen. Det å legge til rader, endre på en rad, eller slette rader er eksempler på nyttige ekstrafunksjoner denne pakken har, der man selv bestemmer hva disse handlingene skal gjøre. Pakken fungerer som en enkel komponent, der det finnes mange attributter som kan tilpasses etter behov.

Vi har valgt å bruke denne pakken ettersom mye av dataene vi har i applikasjonen blir godt presentert i en tabell, og pakken støtter *CRUD*-operasjonene (Create, Reade, Update, Delete). Pakken er også veldig intuitivt for brukeren å bruke i brukergrensesnittet.

7.8.5.11 Next-coockie

- Lisens: MIT
- Dokumentasjon https://www.npmjs.com/package/next-cookies

Dette er en enkel pakke som støtter å hente informasjonskapsler både på klient siden og server siden. Pakken trengs ettersom webapplikasjonen skal hente den innloggede brukeren ved et API kall på serversiden til klienten. Det ble prøvd et par andre pakker med samme formål, men denne var den enkleste å jobbe med, mye på grunn av at den trengte mindre konfigurering. Pakken blir brukt ved å importere *useCookie* hooken, som trenger konteksten som parameter. Man kan da enkelt hente en informasjonskapsel, som vist nedenfor.

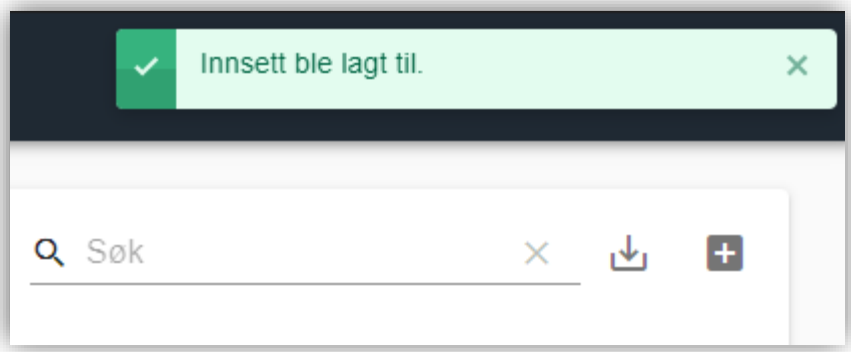
```
1. const cookie = useCookie(ctx);
2. apiToken = cookie.get("APIToken")
```

Eksempel på hvordan Next-coockie kan brukes. Kodesnutt hentet fra kildekoden til applikasjonen (16.02.21).

7.8.5.12 React Toast Notifications

- Lisens: MIT
- Dokumentasjon https://www.npmjs.com/package//react-toast-notifications

Denne pakken blir brukt til å vise varslinger inne i applikasjonen. Disse varslene omhandler som oftest status på om sending av data til APIet gikk igjennom eller om det feilet. Pakken kan enkel brukes ved då innpakke applikasjonen i en *ToastProvider*. Da kan man kalle *addToast* funksjonen inne i komponentene for å gi en varsling til brukeren. Under vises hvordan en varsling kan se ut.



Figur 10: Eksempel av en poppup-varsling i app. Her har et innsett nylig blitt lagt til, og brukeren får beskjed om at forespørselen til APIet fullførte feilfritt.

7.9 Backend

Backend er serversiden tilknyttet datahåndteringen i systemet. Delkapitlet tar for seg teknologien og valgene som er gjort knyttet til database og APIet.

7.9.1 Database

Det finnes flere ulike typer databaser, eksempelvis relasjonsdatabaser, dokumentdatabaser og grafdatabase. Dette delkapitlet tar for seg grunnleggende teori om disse for å danne et godt kunnskapsgrunnlag før valget av databasetype til dette systemet blir gjort. Deretter vil ER-diagram og normalisering bli beskrevet, i tillegg til hvilket databasesystem som ble valgt basert på den valgte databasetypen.

7.9.1.1 Dokumentdatabase

Dokumentdatabase er en såkalt noSQL database som lagrer dokumenter på en nøkkel (AWS, 2021). En slik database kan lagre dataen i dokumenter som er svært dynamisk, slik at man ikke behøver å forhåndsdefinere en modell som sier hvordan dokumentet skal se ut. Objektene kan lagres i kjente filstrukturer som JSON. Denne filstrukturen blir ofte brukt direkte i programmering som kan gjøre det enkelt å anvende dataen direkte. Det har kommet dokumentdatabaser som gjør dette rett i klienten slik som eksempelvis Firebase sin Firestore.

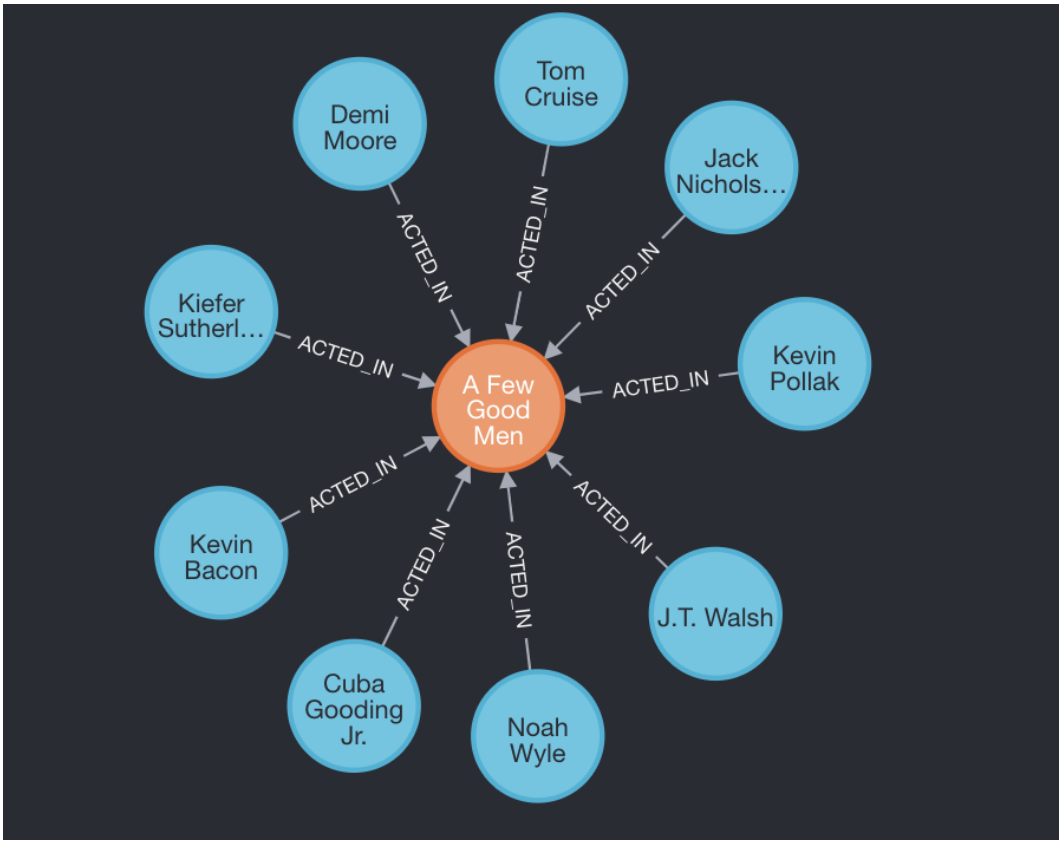
7.9.1.2 Relasjonsdatabase

En relasjonsdatabase anvender relasjonsmodeller. Relasjonsdatabasen er bygget opp av ulike tabeller der tabellene kan har referanser til hverandre i form av nøkler mellom seg (CodeCademy, 2021). Når man anvender en relasjonsdatabase, er man nødt til å definere hvordan dataen skal se ut. Man må altså forklare hvordan dataen skal lagres, hvilken nøkkel den lagres under, hvilken type, man kan også spesifisere regler på de ulike feltene. De fleste relasjonsdatabaser er av typen SQL, som er et spørrespråk man bruker for å opprette, endre og slette databaser, tabeller, rader, indekser og mer.

7.9.1.3 Grafdatabase

En grafdatabaser er bygget opp som en graf, med noder og kanter (neo4j, 2021). Nodene representerer dataen mens kantene representerer relasjonen mellom dataen. I en grafdatabase er det forholdet som prioriteres. Strukturen til en slik database gjør at man enkelt kan visualisere forholdet mellom dataen. Eksempler på en slik grafdatabase er NEO4J. I figuren

under kan man se hvordan strukturen kan se ut. Eksempelet demonstrerer hvordan man kan relatere skuespillere til en film.



Figur 11: Skjermbilde fra Neo4J eksempeldatabase..

7.9.1.4 Valg av databasetype

I dette prosjektet ble det valgt å bruke en relasjonsdatabase ettersom dataen skal struktureres på en bestemt måte som er forhåndsdefinert. Det er heller ingen planer om at strukturen skal ha store endringer. I et slikt tilfellet vil det være best å forhånds definere hele strukturen på dataen for å sikre at alt blir lagret i samme format. I en dokumentdatabase og grafdatabase kan ofte strukturen være mer flytende og ikke så definert i selve databasesystemet. Hvis det var forventet at strukturen skulle endre seg mye, ved for eksempel at Tytlandsvik Aqua la inn nye sensorer hver dag, ville det vært hensiktsmessig å velge en mer dynamisk databasetype som dokumentdatabase eller grafdatabase. Ved at valget av databasetype tar høyde for hvordan systemet skal bli brukt i fremtiden, vil dette imøtekomme kravet om videreutvikling (Krav 13).

7.9.1.5 ER-Diagram

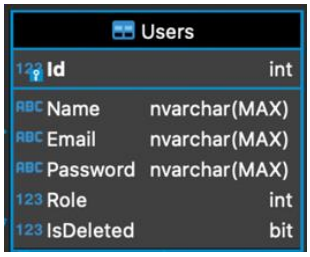
ER-Diagram (Entity Relation Diagram) er en metode man kan anvende for å designe og opprette en grafisk fremstilling av strukturen i databasen. ER-diagram består i hovedsak av entiteter, attributter, forhold og kardinalitet.

Entitet

En entitet er det objektet man ønsker å beskrive. I dette tilfelle kan dette være et kar, bruker eller en hall for eksempel.

Attributter

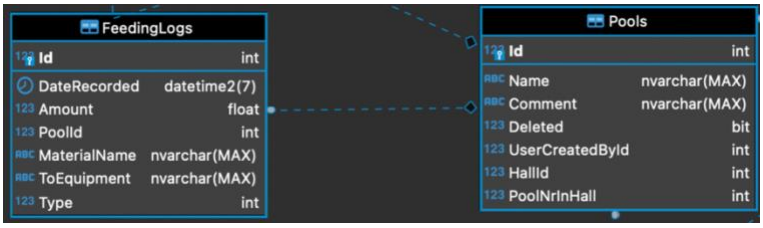
Attributter er egenskaper som tilhører entiteten. For eksempel kan man si at en bruker har et navn. Figuren under viser hvordan en entitet kan se ut, at attributter er egenskapen som entiteten har.



Figur 12: Attributter og entitet eksempel. Skjermbilde fra egen modell.

Forhold

Forhold refererer til forhold mellom de ulike entitetene. Et typisk forhold er et *en til mange* forhold, i vårt tilfelle kan dette være at én hall har mange kar. Figuren under viser hvordan *Pools* (i domenet kar) kan ha mange foringslogger. Men én *FeedingLogs* (i domenet foringslogg) kun kan være knyttet til én *Pool*. Den hvite prikken og de stiplete linjene viser at det ikke må være en relasjon.

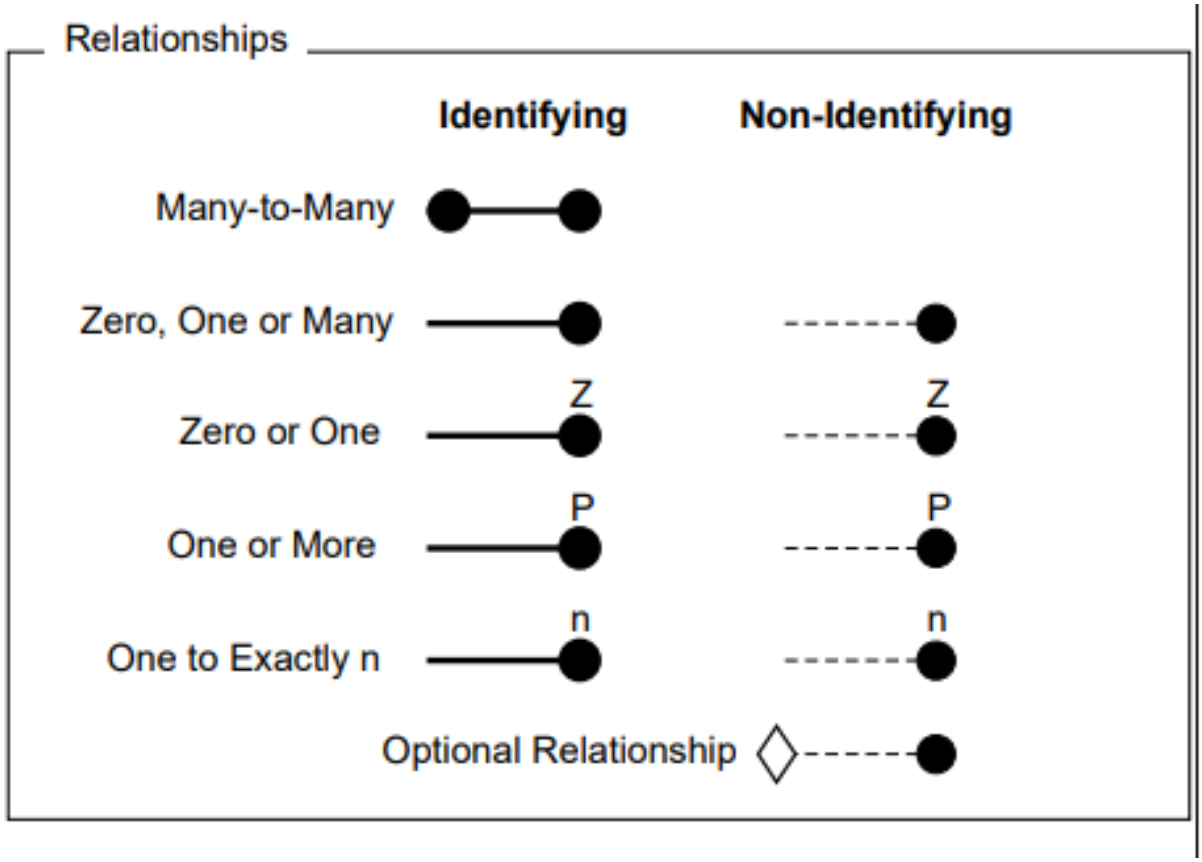


Figur 13: Forhold mellom entiteter. Skjermbilde fra egen modell.

Kardinalitet

Kardinalitet viser til hvor mange forhold en entitet kan ha (Visual Paradigm | ER-Diagram, 2021). For eksempel kan vi i vårt tilfelle si at en hall kan ha mange kar mens et kar kan kun være knyttet til en hall.

Programmet DBeaver er benyttet til å opprette ER-Diagram, forklaring på relasjon representasjon finnes i Figur 14.



Figur 14: DBeaver ER-Diagram relasjons representasjon forklaring.

Bildet er hentet fra: http://www.32geeks.com/classes/resources/IDEF1X_Cheat_Sheet.pdf (23.03.21)

7.9.1.6 Normalisering

Normalisering er en metode man anvender når man skal opprette en database for å sikre integriteten samt redusere lagring av overflødig data (Normalization of Database, u.d.). Vi har valgt å fokusere på å strukturere dataen på en slik måte at den oppfyller normalformene (NF) 1-4. Det finnes flere normalformer, men dette er de aller viktigste for å forhindre store mengder med overflødig data. Ved å normalisere forhindrer man at overflødig data blir lagret, dvs. at man ikke lagrer samme data flere ganger. Man sikrer også at all dataen i en tabell er avhengig av primærnøkkelen.

7.9.1.7 MSSQL/AzureSQL

MSSQL og Azure SQL er den databasen vi har valgt å bruke sammen med Entity Framework. MSSQL står for Microsoft SQL og Azure SQL er Microsofts SQL tjeneste i skyløsningen Azure. Disse databasene er valgt fordi det er valgt å anvende .NET sammen med Entity Framework som også er utviklet av Microsoft. Microsoft har utviklet denne programvaren i lang tid og fortsetter å videreutvikle den, noe som gjør at sikkerhetshull og feil fortsatt blir rettet opp. MSSQL er brukt for lokal utvikling mens AzureSQL er brukt i test- og produksjonsmiljø. Ved å benytte MSSQL og AzureSQL sikrer man også at ACID kravene er oppfylt.

7.9.2 API

Et API er en struktur på hvordan man kan kommunisere med en informasjonskilde. Et API er beskrevet som en megler mellom klienten eller brukeren og en resurs eller en nettservice. API brukes for å håndheve sikkerhet, kontroll og autentisering. Ved å bruke et API blir det lettere å imøtekomme flere av kravene. Krav 6 blir direkte oppfylt ved bruk av API, ettersom den enkelt kan settes opp mot eksterne tjenester. Det gjør det også enklere å få tjenestene til å kommunisere med hverandre.

7.9.2.1 REST

Akronymet REST (også kjent som RESTful API) står for Representational State Transfer (What is a REST API?, u.d.). REST er en arkitekturbegrensning og ikke en protokoll. Når en klient sender et kall til et REST API vil APIet respondere med en representasjon av statusen på det som ble etterspurt. Kommunikasjonen, kall og responser foregår over HTTP og kan bruke flere formater. Det mest vanlige formatet, som også blir brukt i dette systemet, er JSON. Dette er det vanligste fordi det er både lesbart for mennesker og maskiner i tillegg til at det er språkagnostisk. Språkagnostisk vil si at, JSON i dette tilfellet, "ikke vet om andre språk", det har altså ikke noe å si hvilke andre programmeringsspråk som blir brukt. I dette systemet er det valgt å bruke REST ettersom det er god kjennskap til denne teknologien fra før av.

Anatomien til et kall består av endepunkt, metode, hode og kropp:

Endepunkt

Et endepunkt er URLen til man kaller på. Den følger denne strukturen: rot-endepunkt /<*sti*>. Rotendepunktet er utgangspunktet til APIet. <*sti*> bestemmer hvilken resurs som skal kalles på.

Metode

Metoden er typen på kallet man sender til APIet. Siden det blir brukt HTTP kan typene være GET, POST, PUT, PATCH og DELETE. Disse er brukt for å gjøre CRUD operasjoner. Under i tabellen viser bruksområdet til de forskjellige metodene.

Metodenavn	**Kall forklaring**
GET	Dette kallet er brukt for å få ressurser fra serveren. GET kallet utfører en Read-operasjon, det vil si at serveren ser etter og returnerer dataen som ble forespurt. GET er standardverdien for HTTP kall.
POST	POST lager en ny resurs. Serveren oppretter en ny entitet i databasen og gir en respons på om opprettelsen var vellykket eller ikke. POST utfører en Create operasjon.
PUT og PATCH	Begge disse kallene er brukt for å oppdatere en resurs. På samme måte som POST vil serveren gi en tilbakemelding på om oppdateringen er vellykket eller ikke. PUT og PATCH utfører en Update operasjon.
DELETE	DELETE kallet sletter en resurs. Responsen sier om kallet ble vellykket eller ikke. DELETE utfører en Delete-operasjon.

Hodet

Hodet (eng.: head) blir brukt for å dele informasjon mellom klienten og serveren (HTTP headers, u.d.). Det er flere ting som kan være med i hodet, det kan for eksempel ha med

autentiseringsinformasjon eller informasjon på hva kroppen (eng.: body) skal inneholde. En oversikt over hva som er med i et HTTP-hodet finnes på Mozilla sine utviklersider.

Dataen (kroppen)

Kroppen er som regel bare brukt i POST, PATCH eller PUT kall, og gir ytterligere informasjon til serveren. I for eksempel et POST kall vil kroppen si hva resursen som blir laget skal inneholde.

Responsen som APIet returnerer etter et kall består at HTTP status koder og eventuelt feilmeldinger (Liew, 2018). Feilmeldinger blir bare returnert når det er noe feil med kallet. HTTP-statuskodene er derimot alltid med i responsen uansett om kallet var vellykket eller ikke. Statuskodene gjør at man raskt kan se statusen på responsen. Statuskodene er delt opp i fire grupper.

- 200+ betyr at kallet var vellykket
- 300+ betyr at kallet ble redirigert til en annen URL
- 400+ betyr at en feil oppstod hos klienten
- 500+ betyr at en feil oppstod i serveren

7.9.2.2 Caching

Et API uten caching som spør en database om data i hver eneste forespørsel vil i mange tilfeller spørre om den samme dataen flere ganger. Å gjøre en forespørsel igjen og igjen til databasen for å få tak i den samme dataen som forrige gang opptar prosessorkraft fra API-tjenesten samt påføre ekstra belastning på databasen. Det er valgt å bruke caching på diverse endepunkt for å få en raskere respons til klient, samt at det minsker belastningen på databasen, noe som også imøtekommer krav 10.

Caching går ut på at man lagrer en tilbakemelding slik at dataen fra den forrige forespørselen kan sendes tilbake uten å gjøre ytterlige oppslag mot databasen. Man har primært to ulike former for caching, i minne (eng.: in-memory) og distribuert.

I minne

Dette vil si at man lagrer cache i minnet der tjenesten kjøres (Microsoft, 2017). Ettersom dataen blir cachet i minne, vil man ikke kunne dele cache mellom flere instanser. Denne

formen for caching er ideelt dersom man kun skal kjøre på én instans. I minne caching er også relativt enkelt å sette opp og krever ikke noe system for å distribuere cachingen mellom instanser.

Distribuert

Dersom man ønsker å kjøre tjenesten på flere instanser trenger man en database som kan lagre cache-nøkkel med data i en database (Microsoft, 2017). Dette vil gjøre at prosessen går litt tregere, men dataen kan da anvendes av flere instanser.

Valg av form for caching

.NET 5 gjør det særdeles enkelt å implementere i minne og distribuert -caching. Ettersom API-tjenesten til systemet ikke får et veldig stort trykk og kun skal brukes av et fåtall av personer til å begynne med, vil det være tilstrekkelig å gjøre cachingen i minne.

Under følger et kodeeksempel på hvordan man kan anvende caching i et endepunkt.

```
1. /* For å initialisere caching i .NET aplikasjonen ma man legge til
2. linjen under i Startup.cs filen */
3. services.AddMemoryCache();
4.
5. // I API-tjenestens endepunkt kontrollere
6. // Dersom cache-nokkelen finnes er cachet, returner denne dataen
   direkte.
7. if (_cache.TryGetValue(AllHallsCachekey, out List<Hall> cachedHalls))
   {
8.     return Ok(cachedHalls);
9. }
10.  // Er ikke dataen cachet, hent dataen fra databasen og legg den til
   i cache.
11.  var allHalls = _database.Halls.Where((o) => !o.Deleted).ToList();
12.  _cache.Set(AllHallsCachekey, allHalls);
13.  return Ok(allHalls);
14.
15.  /* For a slette et cache objekt, bor gjores dersom dataen endres i
   forhold til cache objektet. */
16.  _cache.Remove(key);
```

Bruk av caching på et endepunkt. Hentet fra ulike steder i kildekoden.

7.9.3 .NET, C#

.NET (uttales "dot net") er et rammeverk utviklet av Microsoft. Rammeverket er kryssplatform (eng.: cross-platform) som vil si at man kan bruke samme kodebase for flere ulike plattformer, eksempelvis mobilapplikasjoner og webapplikasjoner. Vi har valgt å bruke .NET ettersom det er utviklet av Microsoft, noe som sikrer vedlikehold og support i

fremtiden. .NET er skrevet i C# (uttales C-sharp) som er et objektorientert programmeringsspråk utviklet av Microsoft. Språket har hentet mye inspirasjon fra Java og kalles av noen for en imitasjon av Java.

.NET har blitt brukt i all hovedsak til å utvikle APIet. I .NET gjør man dette ved å definere kontrollere til endepunktene. En kontroller er en klasse med metoder til endepunktene, som er funksjoner som løser hva endepunktet skal returnere.

7.9.4 Tredjepartsbiblioteker til backend

Det er benyttet noen tredjebiblioteker i .NET, funksjonen til disse blir kort beskrevet her. Biblioteker som har en stor betydning for prosjektet, blir gått i nærmere detalj i ettertid.

7.9.4.1 AspNetCoreRateLimit

- Lisens: MIT
- Dokumentasjon: https://github.com/stefanprodan/AspNetCoreRateLimit

Denne pakken brukes til å implementere ratebegrensinger på endepunkter i .NET Core API, noe som er forebyggende mot brute force angrep.

7.9.4.2 BCrypt.NET

- Lisens: MIT
- Dokumentasjon: https://github.com/BcryptNet/bcrypt.net

C# funksjon for den kjente hashing-funksjonen BCrypt. Denne blir brukt til å hashe passordet for å lagre det på en sikker måte.

7.9.4.3 NewtonsoftJson

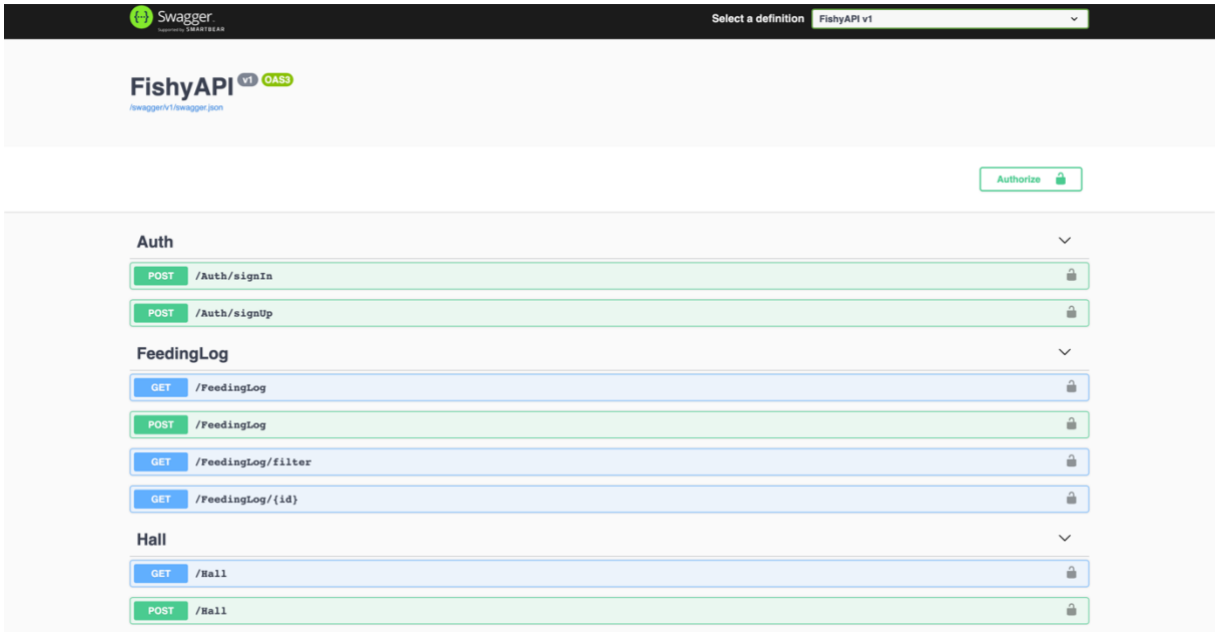
- Lisens: MIT
- Dokumentasjon: https://www.newtonsoft.com/json

Pakke for å serialisere og de-serialisere C# objekter til JSON.

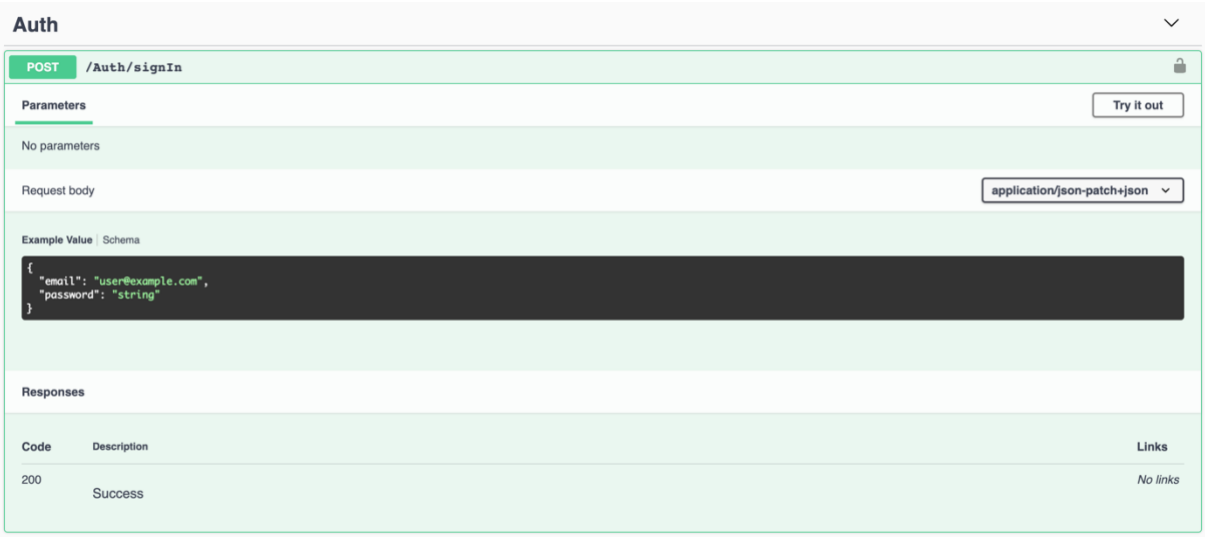
7.9.4.4 Swashbuckle

- Lisens: BSD 3-Clause "New" or "Revised" License
- Dokumentasjon: https://github.com/domaindrivendev/Swashbuckle.WebApi

Swashbuckle tilbyr API-dokumentasjon gjennom verktøyet Swagger. .NET implementasjonen til Swashbuckle leser API kontrollerne og genererer dokumentasjon ut ifra dette. Man får da en side man kan bruke til å teste de ulike API-endepunktene, også med autentisering. Nedenfor kan man se hvordan Swagger lister alle endepunkt, og hvordan det ser ut når man skal teste en forespørsel mot et endepunkt.



Figur 15: Swagger dokumentasjonsside av API tilknyttet prosjektet.



Figur 16: Eksempel på dokumentasjon av endepunkt.

7.9.4.5 SignalR

- Lisens: Apache License 2.0
- Dokumentasjon: https://docs.microsoft.com/en-us/aspnet/signalr/

Denne pakken blir brukt som serverside-tjenesten til websocket-kommunikasjonen i systemet. Som tidligere beskrevet, er SignalR et naturlig valgt ettersom den også er utviklet av Micrisoft og dermed har en god integrasjon mot .NET.

7.9.4.6 Hangfire

- Lisens: LGPL v3 License
- Dokumentasjon: https://www.hangfire.io/

Hangfire er et åpent rammeverk i .NET som er en enkel måte å organisere bakgrunnsjobber på. Man får også en enkel brukerflate som kan anvendes til å monitorer når jobbene kjøres og statusen på disse jobbene. Jobbene bruker kronutrykk for å bestemme hvor hyppig de skal kjøres. Et kronutrykk bestemmer med hvilke intervaller jobbene skal kjøres i. (Hangfire, 2021)

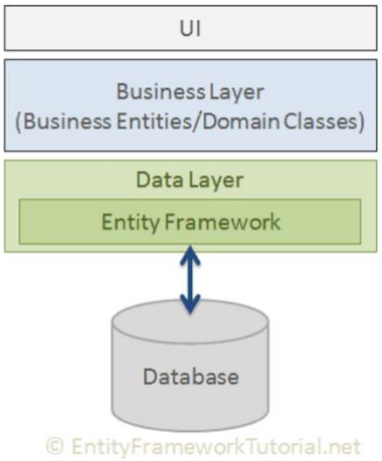
Hangfire blir brukt i tjenesten for innhenting av data. Det er valgt å benytte dette biblioteket på grunn av god støtte i .NET samt mye funksjonalitet som kommer ut av boksen. Eksempelvis webgrensesnitt som gir god oversikt over jobbene som kjøres med tilhørende status evt. feilmeldinger.

7.9.4.7 NodeService

- Lisens: Apache 2.0
- Dokumentasjon: https://github.com/aspnet/JavaScriptServices/tree/master/src/Microsoft.AspNetCore.NodeServices

NodeService er et bibliotek for .NET som tillater å kjøre node.js funksjoner direkte fra .NET applikasjoner. Dette biblioteket ble benyttet ettersom det tidlig ble skrevet node.js script som det var ønskelig å kjøre i .NET for å slippe å bruke tid på omskriving av allerede fungerende kode.

7.9.4.8 Entity framework



Figur 17: Entity Framework

Hentet fra: (What is Entity Framework?, u.d.)

Entity Framework er et ORM (Object Oriented Mapping) rammeverk for .NET applikasjoner (What is Entity Framework?, u.d.). Et ORM rammeverk lager en virtuell objektdatabase som kan brukes i det aktuelle programmeringsspråket. Det vil si at utviklingen blir lettere fordi den virtuelle objektdatabasen støtter typer. Entity Framework lar utvikleren jobbe på ett høyere abstraksjon nivå, som betyr at man jobber med mer domenespesifikke objekter istedenfor å fokusere på de underliggende tabellene og radene til databasen, noe som er hovedgrunnen til at dette ble valgt å benytte.

Migrasjoner

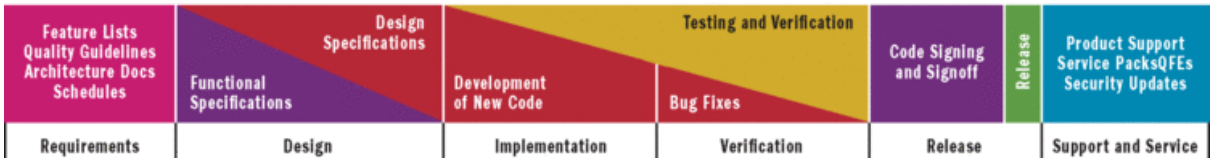
Normalt vil databasen slettes og bygges på nytt når man gjør endringer i datamodellen. Migrasjoner brukes for å løse dette problemet. Ved hjelp av migrasjon kan man endre databaseskjemaer uten å slette all dataen (Anderson, et al., 2020). Dette er spesielt nyttig dersom databasen inneholder mye data.

7.10 Sikkerhet

Når man behandler sensitiv data, er det viktig å ha kontroll på sikkerheten. I dette systemet er det brukerdataen og data om fisken som er sensitiv. Et krav fra Tytlandsvik Aqua er at denne dataen skal bli håndtert på en sikker måte, ref. krav 11 og 12. For å ha full kontroll på sikkerheten til systemet må man se på hvordan alle de ulike delene av systemet håndterer sikkerheten. For at denne oppgaven ikke skal bli for omfattende blir ikke alle sikkerhetsteknologiene forklart i detalj.

7.10.1 Sikkerhet i utviklingsprosessen

Security Development Lifecycle (SDL) er en veiledning som skal gjøre det lettere å adressere sikkerhetsproblemer gjennom utviklingsprosessen (Sullivan, 2019). Figuren under gir en oversikt over prosessene involvert i SDL.



Figur 18: SDL prosessene

Hentet fra (Sullivan, 2019)

SDL er laget av Microsoft og i den nyeste versjonen består den av 80 krav. Veiledningen ble laget for store prosjekter som tar lang tid å utvikle og har kapasitet til å imøtekomme alle de 80 kravene. For mindre prosjekter er det lite hensiktsmessig å prioritere så mye tid på alle disse kravene.

For å ta høyde for prosjekter som tar kortere tid har man kombinert SDL og smidig utviklingsprosess. I en slik kombinasjon er ikke alle kravene i SDL nødvendig å fullføre før man utgir produktet. Hvert krav tjener et unikt formål og ufullførte krav fører til at produktet får sikkerhetssvakheter, noen av de vanligste svakhetene blir presentert i neste kapittel. Man har valgt å dele opp kravene i tre ulike deler.

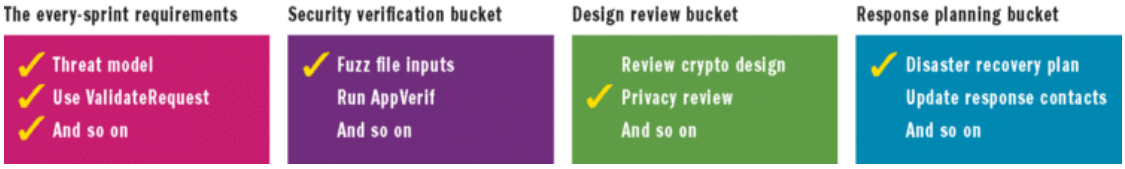
Den ene delen er krav man skal gå gjennom for hver sprint. Dette kaller man førstefaktor (eng.: first factor). Disse kravene sees på som udiskuterbare og må fullføres for hver sprint. Disse blir valgt ut ifra hvor mange og hvor kritisk sikkerhetssårbarheter kravet dekker. I tillegg ser man på hvilke krav som kan automatiseres uavhengig av hvor kritiske sårbarhetene

som blir dekket er. Det er vanlig å ha med trusselmodellering (se kapittel 7.10.3 Trusselmodellering) i disse kravene. Selv om trusselmodellering er tidkrevende er det sett på som svært nødvendig.

"*If we had our hands tied behind our backs (we don't) and could do only one thing to improve software security—threat modeling, better security code reviews, or better security testing—we would do threat modeling every day of the week*" (Howard, Michael; Steve Lipner, 2006).

Disse kravene kaller man for ombordstigning (eng.: onboarding). Dette er krav man bare trenger å gjennomføre i oppstart og trenger ikke gjennomgås igjen senere. Dette kan for eksempel være å sette opp logging av feilmeldinger.

Den siste delen av krav kaller man bøttekrav. Dette er krav som verken passer inn i omborstingings- eller førstefaktor kravene. Disse kravene blir delt opp i tre ulike bøtter. Sikkerhetsverifisering, design gjennomgang og respons planer. Før man gjør en utgivelse skal man gjøre minst en av kravene i hver bøtte. Man må gjøre alle kravene minimum en gang hvert år.



Figur 19: Førstefaktor krav og de tre bøttene med krav.

Hentet fra *(Sullivan, 2019)*

Her er et eksempel på en smidig SDL:

Figur 20: Smidig SDL.

Hentet fra *(Universitetet i Stavanger, 2019)*

7.10.2 OWASP topp 10

OWASP (Open Web Application Security Project) er et online samfunn som lager artikler, dokumentasjon, verktøy og teknologier innen nettsikkerhet (OWASP Foundation, u.d.). OWASP topp 10 er et dokument som beskriver en utviklingsprosess som kan minimere sikkerhetsrisikoen til systemet som utvikles. Dokumentet fokuserer på de ti vanligste sikkerhetsrisikoene som er følgende:

1. **Injeksjon**. Går ut på å sende inn data som uønsket blir kjørt som kode. Injeksjoner er spesielt vanlig i SQL, NoSQL, OS og LDAP.
2. **Ødelagt autentisering**. En usikker autentisering og økt-håndtering tillater angripere å kompromittere passord, token og nøkler.
3. **Sensitiv data eksponering.** Mange APIer beskytter ikke all sensitiv data på en riktig måte som kan resultere i at angripere kan få tilgang på betalings- og identitetsinformasjon mm.
4. **XML eksterne enheter (XXE)**. Angripere kan utnytte svake XML prosesser hvis de kan laste fiendtlig innhold til et XML-dokument.
5. **Ødelagt tilgangskontroll**. Handler om at autoriserte tilganger ikke er ordentlig håndhevet. Det vil si at man for eksempel kan få en liste av alle brukere uten at man burde fått lov.
6. **Feilkonfigurert sikkerhet**. Dette oppstår vanligvis ved at man bruker usikre standard konfigureringer. Dette kan for eksempel føre til at skylagring står åpent og at feilmeldinger inneholder sensitiv data.
7. **Skripting på tvers av nettsider (XSS)**. Kan oppstår når en nettside tillater at ufiltrert fremmed data vises eller kjøres hos brukeren. Angriperen kan laste opp denne dataen i for eksempel postinnlegg som tillater HTML eller JavaScript.
8. **Usikker de-serialisering**. De-serialisering går ut på å oversette eller strukturere serialisert data. Hvis man har en usikker de-serialiseringsmetode og prøver å de-serialisere ondsinnet data, kan det føre til at denne dataen blir kjørt.
9. **Bruke komponenter med kjente svakheter.** Hvis man bruker 3-partskomponenter kan disse ha svakheter. Disse svakhetene kan en angriper utnytte.
10. **Utilstrekkelig logging og overvåking.** Dette kan føre til at angripere uoppdaget kan fortsette med ondsinnede handlinger, som å endre, eksportere, eller ødelegge data.

Dette dokumentet er noe som følges i denne oppgaven for å sørge for at løsningen er så sikker som mulig. Sikkerhetsaspekter ved valg av teknologier, som blir drøftet i de neste delkapitlene, er basert på disse punktene. For å ha en oversikt over hvilke deler av systemet som kan være utsatt for de ulike OWASP risikoene er det nødvendig med en trusselmodell.

7.10.3 Trusselmodellering

Det er mange måter å lage en trusselmodell på (Application Threat Modeling, u.d.). Generelt går trusselmodellering ut på å kartlegge trusler. En trusselmodell har ofte med hva man vil beskytte, potensielle trusler, hvordan håndter de truslene og en måte å validere håndteringen på. I en slik modell kan det være nyttig å benytte seg av et sidekart (se kapittel 7.8.4 Sidekart). Et dataflytdiagram som beskriver hvordan dataen blir sendt mellom de forskjellige komponentene i systemet.

7.10.4 Autentisering

Autentisering er prosessen som omhandler det å etablere eller bekrefte noe som autentisk (Authentication, 2018). I it-sammenheng er innlogging av en bruker en autentisering. Punkt to i OWASP topp 10 peker på denne delen av systemet som tilsier at autentisering er en viktig del av sikkerheten. For å sørge for at brukeren er autentisk er det flere metoder man kan bruke.

For å sjekke om brukeren ikke er en datamaskin, men et faktisk menneske kan man bruke CAPTCHA, som står for Complete Automated Public Turing to tell Computer and Humans Apart (CAPTCHA, 2019).

Flerfaktor autentisering er en annen måte å sjekke om brukeren er autentisk (Multi-factor authentication). Denne autentiseringsmetoden gir tilganger kun dersom man vellykket har gitt flere bevis på autentisitet. Bevisene kan komme fra noe brukeren *vet*, *har* eller *er*. BankID på mobil er et eksempel på dette. Brukeren må vise at de *vet* passordet og at de *har* telefonen knyttet til id-en. Det kan argumenteres for at ip-filtreringen, ved for eksempel bruk av en brannmur, er en form for flerfaktor autentisering.

For å ikke ha en for omfattende oppgave er CAPTCHA eller flerfaktor autentisering ikke brukt, men dette har høy prioritet ved eventuell videreutvikling. En annen grunn til at det er valgt bort er fordi det ikke er forventet store mengder med brukere, under ti brukere, dette blir

derfor enkelt å monitorer i loggene (se kapittel 10.2.10 Logging). Vanlig innlogging med brukernavn og passord er i dette tilfellet derfor sikkert nok.

7.10.5 Passord

Passordet og epost er det eneste som trengs for å bli autorisert i det utviklede systemet. Det er derfor utrolig viktig å håndtere passordene på en sikker måte.

Det er flere måter en angriper kan få tilgang på passordene (Damtoft, 2019). Angriperen kan for eksempel tilegne seg passordet ved bruk av brute force angrep, som betyr at man prøver mange passord helt til man treffer et passord som fungerer. En annen måte er å hacke seg inn på databasen og se passordene hvis de er lagret i klartekst.

Ved å beskytte seg mot de ti vanlige sikkerhetsrisikoene som nevnt i 7.10.2 OWASP topp 10 vil dette løse mange av risikoene forbundet med passord, men det er likevel verdt å merke seg noen ekstra punkter om passordsikkerhet.

Regler for nye passord bør ikke være for strenge, fordi det vil redusere antall mulig passord som gjør det enklere med brute force angrep. For å sikre at passordene ikke er for enkle bør man kreve at de har en viss lengde og evt. at man må ha med symboler eller tall.

Passord skal ikke lagres i klartekst, det skal hashes før det lagres. Forskjellen fra hashing og kryptering er at hashing ikke er reversibelt, og det finnes flere hashealgoritmer som kan brukes. Noen er usikre, som for eksempel MD5 og SHA-1, mens andre er betegnet som sikre. I tillegg til å bruke en sikker hashealgoritme bør man legge inn et salt. Et salt er en tilfeldig mengde med data som legges til det som skal hashes. Ved bruk av saltet hasher vil brute force bli vanskeligere og ta lenger tid. Hvis en angriper får tilgang på lagrede passord vil ikke dette være en like umiddelbar krise ettersom passordene er hashet.

Som Moores lov (Tengesdal, 2018) tilsier vil databehandlingskraften øke og bli billigere. Dette fører til at brute force angrep blir mer og mer effektive. For å beskytte seg mot dette må man bruke fremtidssikre hashealgoritmer som BCrypt og PBKDF2. Disse algoritmene tar høyde for Moores lov og sørger for at hver enkel sjekk av passord vil bli mer krevende etter hvert som databehandlingskraften øker.

Etter å ha undersøkte flere fremtidssikre tredjeparts hashealgoritmer endte vi opp med å velge tredjepartsbiblioteket BCrypt.net-Next (ChrisMcKee, 2020). Grunnen til det er fordi den er veldig populær noe som indikerer at feil vanligvis blir funnet fort og rapportert. Biblioteket er også vedlikeholdt som betyr at feilene som blir funnet blir tatt hånd om.

7.10.6 Autorisering

Autorisering er prosessen som omhandler det å gi eller nekte tilganger til resurser. Det er viktig at alle endepunktene i API-et er konfigurert slik at de bare er tilgjengelig for brukere med riktig autorisering. I OWASP topp 10 tilsvarer denne delen punkt 5, Ødelagt tilgangskontroll. JWT er valgt for å sjekke om brukeren er autorisert, noe som kan leses mer om i 7.10.8 Sikkerhetsteknologier under.

7.10.7 STRIDE

STRIDE er en struktur man kan bruke for å identifisere trusler. Akronymet STRIDE står for Spoofing, Temparing, Repudiation, Infomation disclosure, Denile of service og Elevation of privilege. Dette er trusselkategorier som skal i størst mulig grad dekke alle potensielle trusler for et datasystem.

7.10.8 Sikkerhetsteknologier

Her blir det beskrevet om sikkerhetsteknologier som blir benyttet direkte og indirekte i systemet.

Azure sikkerhetsteknologi

Ved å benytte seg av Azure sine skytjenester får man en del "gratis" tjenester, deriblant sikkerhetstjenester. Azure AppServices tilbyr blant annet TLS, tvungen HTTPS og brannmur. Azure SQL tilbyr kryptering og sikkerhetskopiering. Application Insights er en tjeneste som kan brukes til å overvåke applikasjoner mad tanke på sikkerhet, men også driftssikkerhet.

- TLS er en kryptografisk protokoll som tilbyr endepunkt kryptering noe som gjør at man ikke kan tyde det som blir sendt over nettet. På denne måten forhindrer man tyvlytting av dataene som blir sendt. Ettersom det brukes kryptering vil et ondsinnet mellomledd kun ha mulighet til å lese dataene dersom den dekrypteres.

- Tvungen HTTPS er en funksjon som gjør at man tvinger brukeren til å bruke HTTPS. Dersom en bruker forsøker å gå inn på http, som da ikke er kryptert vil brukeren bli

omdirigert til HTTPS. På denne måten kan man sikre at alle brukere anvender sikre tilkoblinger.

- Brannmur er maskinvare og/eller programvare som beskytter datanettet mot uautorisert bruk. Her kan man eksempelvis spesifisere hvilke IP-adresser som skal ha tilgang til denne tjenesten.

- Dataen som lagres i AzureSQL kan krypteres når den lagres slik at den kun kan brukes av autentiserte brukere. Dette er en sikkerhetsmekanisme som forhindrer tilgang til dataen selv om man kommer inn til databasen i et eventuelt angrep.

- AzureSQL tilbyr som standard sikkerhetskopiering som jevnlig tar sikkerhetskopieringer av databasen. Sammen med dette kommer et verktøy som gjør at man enkelt kan gjenopprette databasen til et gitt tidspunkt. Dette gjør at man ikke risikerer å miste data dersom man blir utsatt for et angrep som sletter eller manipulerer dataen.

JWT

JSON Web Token er en åpen standard som brukes til å overføre informasjon som et JSON objekt på en sikker måte (Broecklmann, 2017). En fordel med JWT er at den er mer kompakt sammenlignet med SAML som brukes til det samme som JWT.

Tokenen er delt opp i tre deler, hode, nyttelast og signatur. For å lage signaturen krypteres hodet, nyttelasten og en nøkkel med algoritmen spesifisert i hodet. Hvis du for eksempel bruker HMACSHA256 vil signaturen bli laget slik:

```
HMACSHA256(
   base64UrlEncode(hodet) + "." +
   base64UrlEncode(nyttelast),
   hemmelig nøkkel
)
```

Signaturen blir brukt for å verifisere at meldingen ikke er endret på veien til en mottaker. Når tokenen er signert med en privat nøkkel kan den også bli brukt for å verifisere at sendere av JWT-en er den de sier de er.

Vi bruker en tredjepartspakke kalt *system.IdentityModel.Tokens.Jwt*, som bruker HS256 spesifisert i hodet. Dette kan bekreftes ved å bruke en JWT-avlusingsverktøy med en JWT generert av tredjepartspakken:

Figur 21: Dekodet JWT, ser at algoritmen i hodet stemmer med det som er definert i tredjepartspakken.
Hentet fra *(Debugger, u.d.)*

OWASP ZAP

ZAP er et flaggskipsprosjekt til OWASP og verktøyet OWASP anbefaler for å sikkerhetsteste webapplikasjoner (Getting Started, u.d.). Dette verktøyet er brukt for å teste den generelle sikkerheten til webapplikasjonen. Ved bruk av dette verktøyet blir det lettere å gjøre systemet som utvikles sikrere.

Figur 22: ZAP mellom nettleser og webapplikasjon.
Hentet fra *(Getting Started, u.d.)*

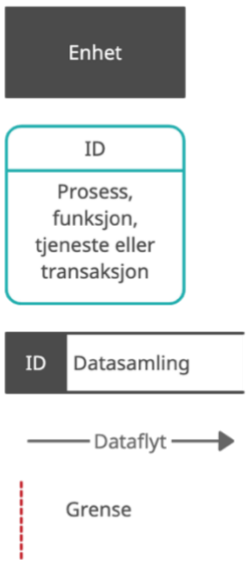
Hovedfunksjonen til ZAP er:

- Være en proxy (mellomledd) som gjør det mulig for ZAP å se HTTP kallene.
- Aktiv og passiv skanning. Passiv ser på HTTP responsene, aktiv kjører flere angrep.
- Edderkopp, som finner skjulte sider og informasjon. Krabber (eng.: crawling) alle linkene og henter strukturen på nettsiden.
- Brute force, bruker OWASP DirBuster.
- Fuzzing, bruker fuzzdb og OWASP JBroFuzz til å finne subtile svakheter. Fuzzing er teknikk som går ut på å automatisk prøve ut tilfeldige ting.
- Rapportgenerator, lager en rapport for brukeren med informasjon om de svakheter som er funnet.

7.10.9 Dataflytdiagram

Et dataflytdiagram er forkortet til DFD som står for Data-flow diagram. DFD gir en grafisk fremstilling av flyten til data i et system. Diagrammet kan fungere som et ryddig verktøy for å kommunisere med kunden, eller som et verktøy til trusselmodellering.
For å lage en DFD må man følge en gitt standard. For å beskrive mer kompliserte dataflyter finner man litt forskjellige løsninger, men standarden har fem grunnleggende komponenter som skal følges.

1. Firkant representerer en enhet. For eksempel en bruker.
2. En avrundet firkant representerer en funksjon, prosess eller transaksjon.
3. To parallelle streker representerer en datasamling.
4. En pil representerer dataflyten.
5. En stiplet rød linje representerer en grense. For eksempel en brannmur.



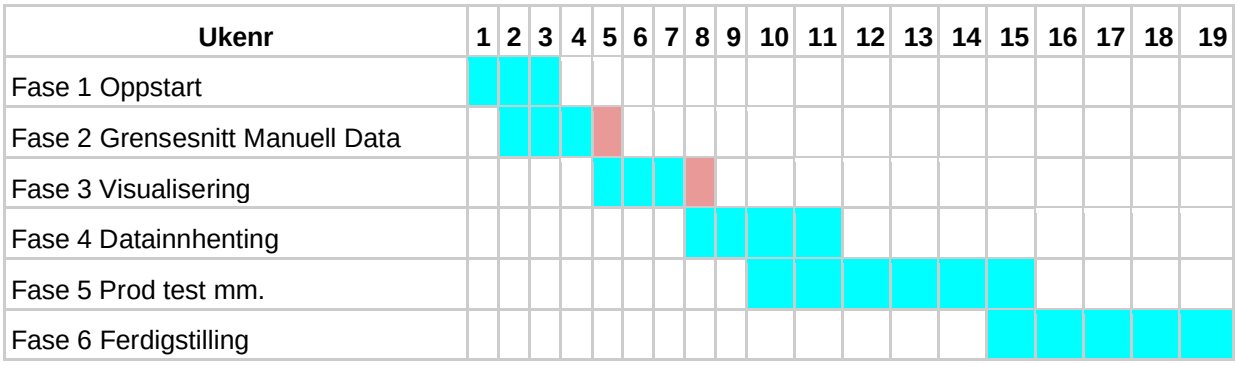
Figur 23: De fem grunnleggende komponentene i DFD

8 Arbeidsmetodikk

Dette kapittelet tar for seg hvordan vi som gruppe har valgt å organisere oss, samt ulike prosesser som blir anvendt for å sikre fremgang og verifisering av progresjonen. Ved å på forhånd definere de ulike arbeidsmetodikkene og prosessene som skal anvendes blir det også klarere hvem som skal gjøre hva og hvordan kvaliteten på arbeidet skal sikres.

8.1 Fremdriftsplan

Ved oppstart av prosjektet ble det definert ulike faser av prosjektet for å ha noen små milepæler å jobbe mot, samt at alle skal vite hva man skal fokusere på til enhver tid. Det ble kommet frem til at fasene ville overlappe hverandre slik at to personer ikke risikerer å vente på at en fase skal fullføres. Fremdriftsplanen ble delt inn i seks faser:



- **Fase 1: Oppstart**
  Den første fasen valgte vi å kalle for oppstart. Formålet med denne fasen var å gjøre klart samt legge til rette for videre utvikling. Et av hovedmålene var for eksempel at arkitekturen skulle verifiseres av Tytlandsvik Aqua. Fasen ble delt inn i flere mindre oppgaver for å sette faste betingelser på hva som må være gjort før vi beveger oss over i fase nummer 2.

- **Fase 2: Grensesnitt Manuell Data**
  I fase to skulle grensesnittet for å legge inn manuell data samt API-endepunkter for dette ferdigstilles.

- **Fase 3: Visualisering**
  I denne fasen var målet å ferdigstille frontend for å kunne presentere dataen mot hverandre på en fornuftig måte.

- **Fase 4: Datauthenting**
  Målet med denne fasen var hente ut data fra eksisterende systemer slik at det kunne anvendes opp mot det nyutviklede systemet.

- **Fase 5: Testing, produksjonssetting og endringer**
  Målet med denne fasen var å teste implementasjonen i alle ender, rulle ut programvaren til produksjon samt gjøre endringer som måtte forekomme etter grundig testing.

- **Fase 6: Ferdigstilling**
  I denne fasen var det fokus på resultat, ferdigstilling av rapport og kartlegge muligheter for videreutvikling.

8.1.1 Smidig endring av faser

Grunnet Covid-19 pandemien som fikk en oppblussing i Norge på nyåret i 2021, var det ikke mulig å komme til anlegget for å hente ut data de første ukene. Den planen, som ble lagt i den første uken, var å starte med å hente ut dataen slik at systemet kunne bygges opp rundt dataen. Ettersom dette ikke var mulig grunnet situasjonen ble det bestemt å flytte denne fasen til en av de siste fasene.

8.2 Arbeidsflyt

For å sikre at ny funksjonalitet blir dokumentert og gjennomgått på en måte som minimerer muligheter for feil, samt øker den kollektive kjennskapen til alle ender av systemet, er det viktig å innføre en god arbeidsflyt.

8.2.1 Scrum og Kanban

For å organisere arbeidsoppgavene ble det benyttet en arbeidsmetodikk kombinert av Scrum og Kanban. Scrum ble anvendt ved å satte opp mange små oppgaver i en backlog og avtalte hva som skulle gjennomføres fra uke til uke. På begynnelsen av hver uke ble oppgavene gjennomgått for å sjekke opp hva som faktisk hadde blitt fullført. Det ble også opprettet kanaler på Slack som var koblet til de ulike GitHub oppbevaringsstedene for at alle enkelt kunne se fremgangen som ble gjort.

En Kanban-tavle ble anvendt til å holde styr på oppgaver og dens status. Tavlen ble delt opp i 5 ulike faser:

- **Backlog**
  Her legges oppgaver som må gjøres, men som ikke har blitt prioritert enda, eller ikke hører til den fasen prosjektet er i.

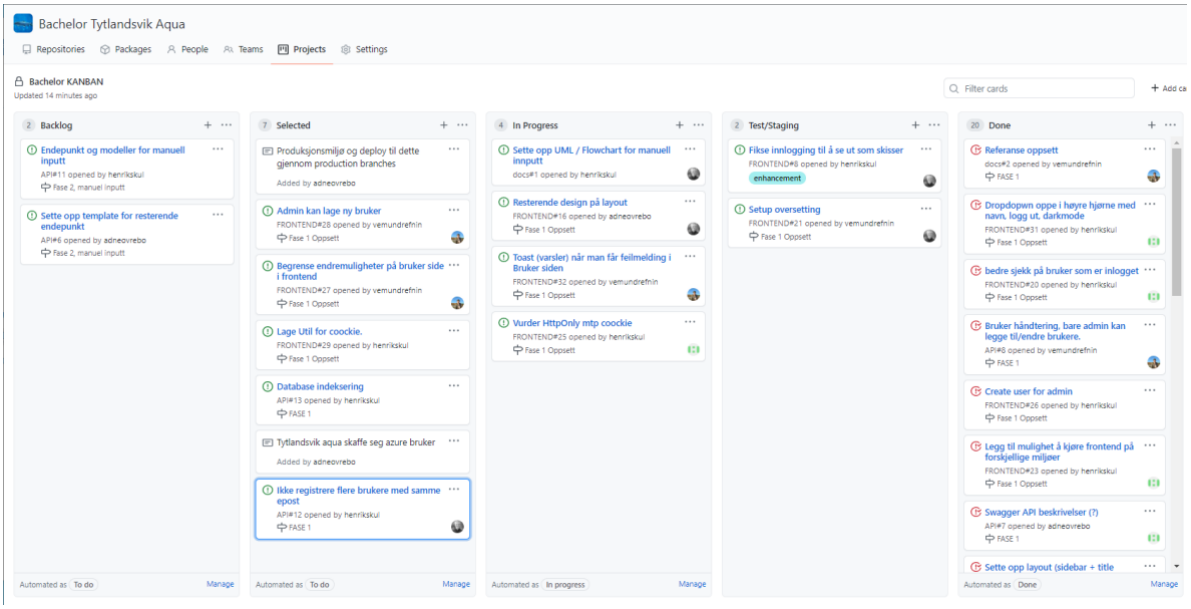
- **Selected**
  Oppgaver som har blitt bestemt at skal utføres så snart som mulig.

- **In Progess**
  Oppgaver som for øyeblikket blir jobbet på.

- **Test/Staging**
  Oppgaver som er fullførte og har blitt rullet ut til test og staging miljø. Oppgavene kan ikke flyttes herfra før prosjektmedlemmene har verifisert at oppgaven er utført riktig. Dersom man finner feil eller må gjøre endringer her flyttes oppgaven tilbake til In Progress kolonnen på tavlen.

- **Done**
  Oppgaver som har blitt kontrollert og er ferdig utført.

Det ble benyttet en Kanban-tavle i GitHub. Ved å benytte en tavle her kan oppgaver knyttes til de ulike oppbevaringsstedene. Dette gjør det enkelt å ha kontroll over hvor en oppgave skal utføres, for eksempel i frontend.



Figur 24: Skjermbilde fra Kanban-tavle under prosjektet.

8.2.2 Kodegjennomgang

For å sikre kvaliteten på koden og at oppgaver ble implementert på riktig måte, gjorde vi det påbudt med kodegjennomgang fra minst én person før koden kunne legges inn i prosjektet, helst en som hadde jobbet i samme området av systemet tidligere. Dette ble håndhevet ved at man ikke kunne pushe direkte til master branchen. Man måtte heller lage en pull request, som krevde minst en gjennomgang, for å få koden inn i master branchen.

9 Arkitektur

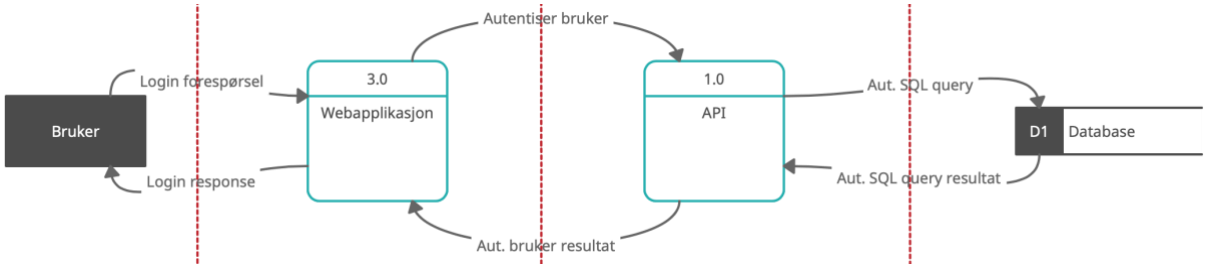
Dette kapitelet tar for seg arkitekturen til systemet. Som nevnt tidligere er det valgt å benytte en arkitektur med flere mikrotjenester. En slik arkitektur baserer seg på flere mindre individuelle tjenester som snakker sammen. Tjenestene bygges til Dockerbilder og kjøres som individuelle beholdere som kan kommunisere med hverandre. På denne måten kan de ulike tjenestene skalere uavhengig, samtidig som man reduserer avhengigheten mellom de ulike tjenestene til et så lavt nivå som mulig.

Systemet er som tidligere nevnt delt opp i 4 ulike beholdere, frontend, API, tjeneste for informasjonsinnhenting og Database. I produksjon og testmiljø er det valgt å ikke pakke databasen inn i en beholder ettersom AzureSQL tilbyr svært mye god tilleggsfunksjonalitet man mister ved å kjøre eksempelvis MSSQL i en beholder. MSSQL blir kjørt i en beholder for lokal utvikling.

9.1 Dataflyt

Diagrammet i 6 Overordnet arkitektur (Figur 5) viser dataflyten til hele systemet. I kontekst av dataflytdiagrammer kaller man dette for nivå null. Høyere nivåer av dataflytdiagram går mer i detalj. Systemet er strukturert rundt mikrotjenester og flyten av data beveger seg derfor gjennom flere av disse tjenestene. Tjenestene kjøres på hver sin AppService i Azure. Som en følge av dette er det derfor en del grenser, røde stiplede linjer, i diagrammene.

Under følger en oversikt over hele innloggingsprosessen gitt ved en DFD på nivå 1, altså litt mer detaljert:

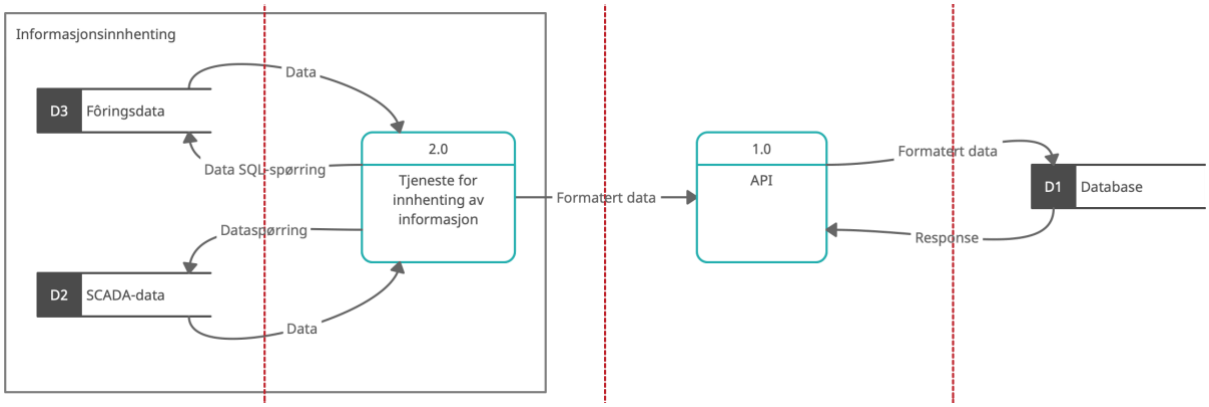


Figur 25: Dataflyt for innlogging

En bruker sender en innloggingsforespørsel til webapplikasjon (3.0) som befinner seg på en annen tjeneste, derav den stiplede røde linjen. Webapplikasjon (3.0) prøver å autorisere brukeren ved å sende en forespørsel til APIet. APIet spør etter brukerdataen i databasen(D1)

og gjør selve autentiseringen. Resultatet av autentiseringen sendes tilbake samme veien som forespørselen kom. Samme forespørselsflyt blir også benyttet hvis en bruker skal hente ut eller laste opp data.

Innhentingen av dataen fra Tytlandsvik Aquas systemer ser derimot noe annerledes ut:



Figur 26: Dataflyt for innhentingstjenester

Både fôringsdataen (D3) og SCADA-dataen(D3) befinner seg på Tytlandvik Aquas nettverk. Innhentingstjenesten (2.0) gjør regelmessige kall på fôringsdataen og SCADA-dataen. Denne dataen blir formatert og sendt videre til APIet (1.0). APIet lagrer denne dataen i databasen (D1).

9.2 Miljøer

For å kunne teste programvaren slik at man ikke påvirker dataen som brukes i produksjon er det valgt å opprette flere miljøer. Dette gjøres for å sikre klare skiller mellom systemer man utvikler og tester på med systemer som blir benyttet av kunden.

Det er valgt å benytte fire ulike miljøer. Lokal, test, QA (eng.: quality assuranse, no: kvalitets sikring)/Staging og produksjon. Disse systemene brukes ved ulike senarioer og skal ikke kunne påvirke hverandre under noen omstendigheter.

- **Lokal**

  Det lokale miljøet skal benyttes under utviklingen av systemet når det kjøres på en egen PC. Dette miljøet benytter kun lokale instanser av de ulike tjenestene. Grunnen til at det er satt opp et lokalt miljø er fordi der kan man teste endringer som man kanskje ikke ønsker å bruke. Hvis man hadde testet slike endringer direkte i andre miljøer enn det lokale miljøet kan det være vanskelig og tidkrevende å reversere.

- **Test**

  Testmiljøet kjører i Azure som ulike AppServices-instanser og skal brukes til å teste hele det eksisterende systemet med nye endringer, samt teste ut alle mikrotjenestene mot hverandre. Dette miljøet skal ikke under noen omstendigheter benyttes som et produksjonsmiljø da testmiljøet kan krasje og måtte resettes, dataen i dette miljøet kan altså gå tapt.

- **QA/Staging**

  Dette miljøet brukes for å teste systemet før det går ut i produksjon og skal som hovedregel i dette prosjektet være en nærmest identisk kopi av produksjonsmiljøet. Det skal altså bruke produksjonsdata. Dette miljøet skal verifisere at migrering av kode fra test mot produksjon skjer på riktig måte uten feil.

- **Produksjon**

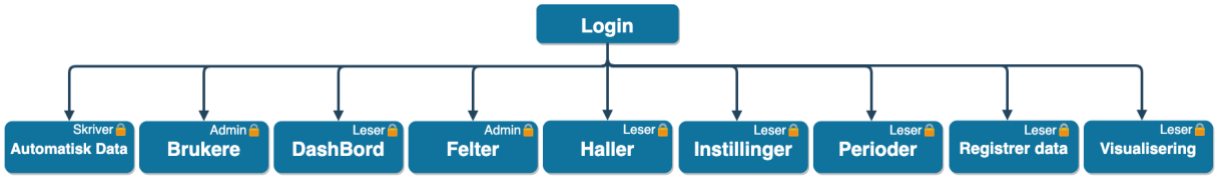
  Produksjonsmiljøet benyttes av kunden i daglig bruk og skal kun motta oppdateringer som har blitt verifisert i QA/Staging-miljøet. Dette miljøet skal også inneholde de strengeste sikkerhetsmekanismene med tanke på uautorisert tilgang og tap av data.

9.3 Frontend

I dette kapittelet vil det bli beskrevet om generell flyt og struktur av frontenden til systemet.

9.3.1 Sidekart

I denne oppgaven er sidekartet to steg dypt, som er innenfor anbefalingene fra brukervennlighetsekspertene (Osman, 2020). På alle sidene utenom **Logg inn** må brukeren være autorisert som rollen *leser* eller høyere, dersom man ikke er det vil man bli omdirigert til **Logg inn** siden. Rollene som er spesifisert er et minimumskrav, der brukere med roller som har høyere gradering også vil ha tilgang. Det er enkelte funksjoner i noen av sidene som er gradert høyere enn tilgangen på å se siden. En leser kan for eksempel se registrert data, men ikke registrere ny data. Se kapittel 9.4.1 Roller for mer informasjon om roller.



Figur 27: Enkelt sidekart av klientsiden til systemet

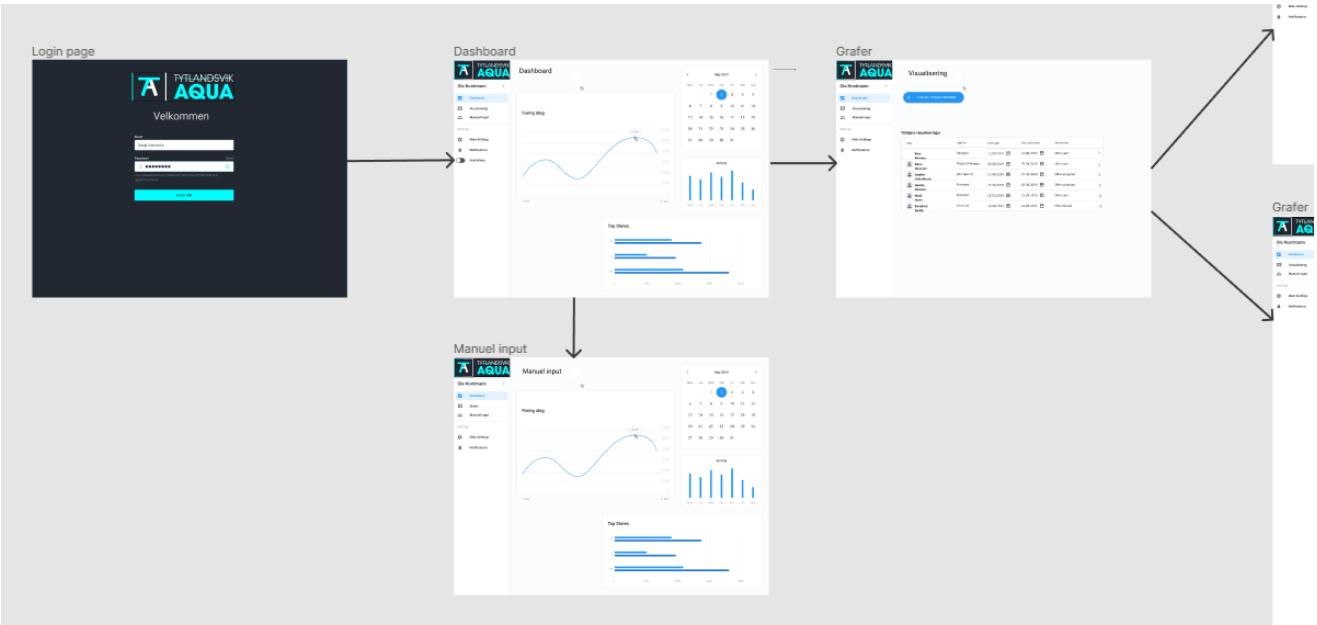
I figuren ovenfor kan man se oversikten over alle sidene til applikasjonen i et sidekart. Formålet med de forskjellige sidene er:

- **Logg inn:** Innloggingside med epost og passord.
- **Automatisk data:** Mulighet til å laste opp data ved å sende inn loggfiler fra SCADA-systemet.
- **Brukere:** Side med oversikt over alle brukere, kan endre passord, brukernavn og epost. Kun administratorer har tilgang til denne siden.
- **Dashbord:** Denne siden viser viktigste informasjon med sanntidsdata.
- **Felter:** Mulighet til å spesifisere feltene dataen filtreres på med navn, farger ol.
- **Haller:** Side med oversikt over hallene, same mulighet til å slette og endre Hallen.
- **Innstillinger:** Enkel innstillingsside med mulighet til å endre på innlogget brukers navn og passord.
- **Perioder:** Side med oversikt over innsett. Den heter periode fordi det i fremtiden skal være mulig å lage perioder som er uavhengig innsett.

- **Registrer data:** Siden hvor man kan manuelt registrere data.
- **Visualisering:** Her kan man velge hvilken data som skal visualiseres og hvordan den dataen skal visualiseres.

9.3.2 Skisser

Tidlig i prosessen på utviklingen av dette systemet ble det valgt å lage noen enkle skisser for webapplikasjonen ved bruk av plattformen Figma, som vises i figur 28 under. Utseende på disse sidene har blitt endret på ved selve implementeringen, men selve brukerflyten i skissene av applikasjonen har blitt beholdt.



Figur 28: Første skisser av klientsiden som viser enkel sidestruktur

Skissene ble laget med komponenter fra ferdiglagde maler, for å raskt få en god oversikt over hvordan vi skulle strukturere webapplikasjonen. Brukeren starter på innloggingssiden, og kommer deretter inn til dashbordet på hovedsidene. Inne på hovedsidene ville vi ha en enkel navigasjonsbar på venstre side. De to pilene på høyre side i Figur 28 viser til to sider som senere ble utelatt, til fordel for å få en enklere brukeropplevelse. Ellers i dette stadiet ble det bestemt å bruke modaler der bruker legger til ny data. I nåværende applikasjon blir modaler kun brukt som bekreftelsesmodaler, samt når buker legger inn ny manuell data ettersom det må fylles ut mange felter. Resten av dataene blir lagt til ved bruk av Material Table.

9.4 API

Dette kapittelet vil i hovedsak ta for seg hvordan APIet er strukturert og hvem som skal ha tilgang til hva.

9.4.1 Roller

For å begrense tilganger til de ulike brukerne kan man benytte seg av et MLS (Multi Level Security) system. Dette er gjort ved å benytte seg av ulike roller. For å kunne definere de ulike rollene til dette systemet må man se på hvilken funksjonalitet man ønsker å begrense til de ulike brukerne. Den enkleste måten å gjennomføre dette på er å lage en kravliste rundt hva brukere skal ha mulighet til å gjøre:

- Noen brukere skal bare kunne se dataen.
- Noen brukere skal kunne registrere ny data og endre på eksisterende data.
- Noen brukere må ha tilgang til å administrere brukertilganger og andre nøkkelinnstillinger.
- Noen brukere skal være et system som automatisk med jevne mellomrom legge inn data.

Ut ifra de fire kravene som er spesifisert over kan man utlede fire roller der hver rolle skal oppfylle ett eller flere krav. Navn i parentes er det som er brukt i koden.

- **Leser (Reader)**: En bruker som kun kan se data, ikke all data, men data tilknyttet de ulike datakildene.
- **Skriver (Writer)**: En bruker som kan se data, endre data og legge til data, dette gjelder kun data tilknyttet datakildene.
- **Administrator (Admin)**: En bruker som kan se data, endra data, legge til data og administrere brukere og nøkkelinnstillinger.
- **Data System (Scheduler):** Et system som kun skal kunne legge til data fra datakildene. Tjenesten for innhenting av informasjon kan være et eksempel på dette systemet. Denne rollen må kun ha tilgang til å legge til data for å forhindre skader dersom et system blir kompromittert.

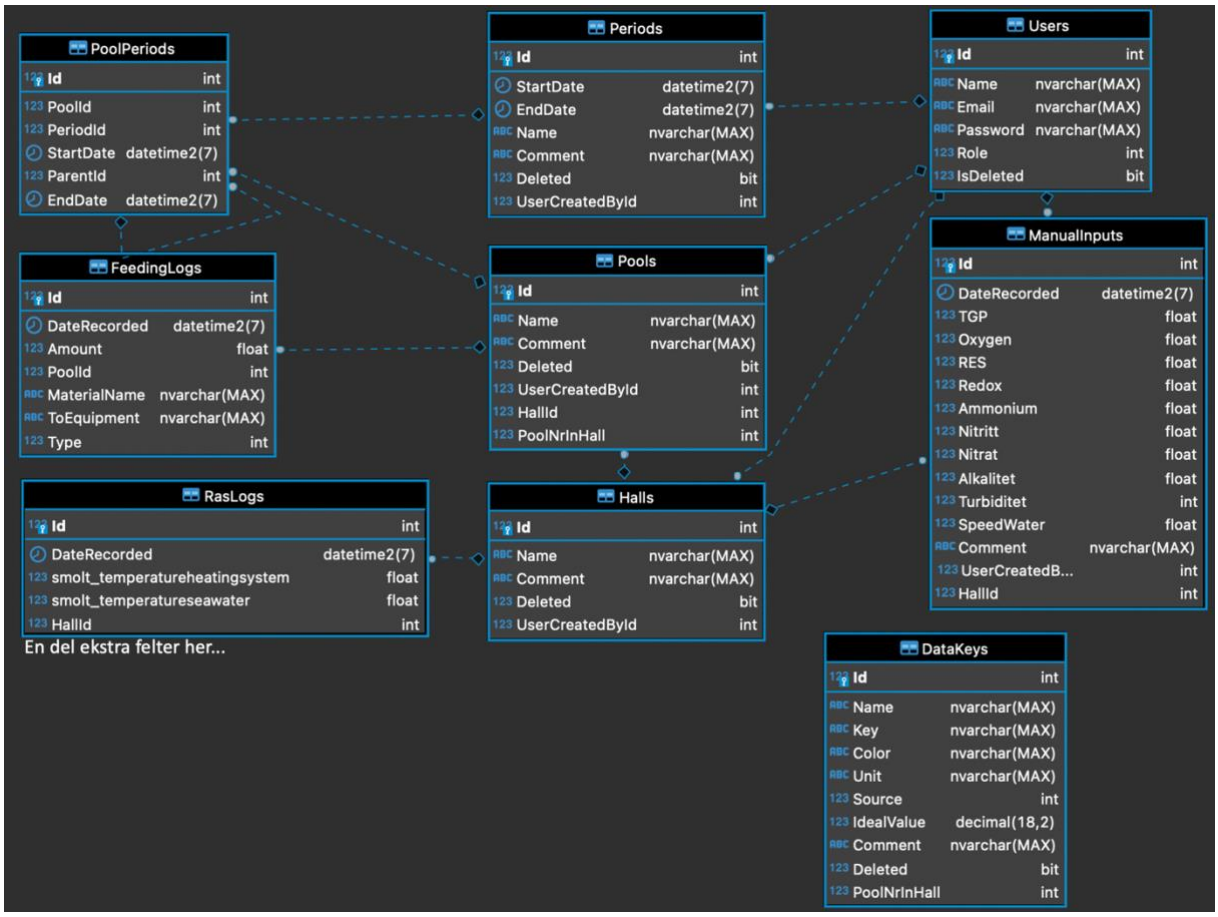
Hvordan disse rollene faktisk skal håndteres og håndheves blir gått nærmere innpå i implementasjonen av API-et.

9.4.2 Endepunkter

For å strukturere APIet på en god måte er det viktig å gå gjennom hvilke endepunkter det er behov for og funksjonaliteten til disse. Se vedlegget 16.2 Systemets Endepunkter for en fullstendig liste.

9.5 Database ER-Diagram

Under utvikling av databasen ble det gjennomgått mange iterasjoner med ER-Diagram. Prinsippene rundt normalisering, som nevnt i teoridelen, er benyttet med størst fokus på å ikke lagre overflødig data, samt lage modeller som stemmer overens med det domenemodellene skal representere. Domenemodeller er objekter fra domenet som for eksempel kar, hall o.l. Siste utarbeidet ER-Diagram som stemmer overens struktur som er implementert kan sees under.



Figur 29: Siste iterasjon av ER-diagram

Strukturen til de fleste tabellene er enkel å utlede ut ifra ER-diagrammet og har ingen struktur som er særdeles komplisert og kan enkelt utledes fra domenet utenom tabellen *PoolPeriods*. Denne tabellen ble utledet ettersom det var et krav (Krav 5) fra kunden at man skulle kunne spore hvilke kar fisken har vært i løpet av et innsett. Dette er fordi de i fremtiden, når de får flere haller ferdigstilt, ønsker å splitte fisken i et kar til flere kar og eventuelt flytte på fisken fra et kar til et annet. I denne prosessen er det viktig for kunden å kunne få riktig data basert på hvilke vannverdier og foringsverdier det har vært i karene til den tiden fisken har vært der. Man skal altså kunne finne ut av hvorfor fisken i et gitt kar har gått enten bra eller dårlig ved å se tilbake på den historiske dataen.

Det trengs her altså en mange-til-mange relasjon mellom *Periods* og *Pool* for å løse dette problemet, der *PoolPeriods* vil fungere som krysstabell (eng.: Junction table). Når en *PoolPeriod* avsluttes og da fisken enten er ferdig hos Tytlandsvik Aqua eller skal til et nytt kar settes *EndDate* til når perioden ble avsluttet. Dersom fisken blir satt inn i et nytt kar blir det opprettet en ny *PoolPeriod* som setter *ParentId* til den perioden som nettopp ble avsluttet. Ved å referere til forelder kan man alltid finne veien fisken har tatt og dermed har man grunnlaget for å kunne se hvor fisken har vært til enhver tid i et bestemt innsett.

9.6 Tjeneste for innhenting av informasjon

Dette er en tjeneste som kunne vært lagt under API-tjenesten, men det er valgt å skille dette ut som en egen mikrotjeneste for å holde de ulike delene av systemet adskilt for hverandre. Dette ble også valgt fordi det i starten var usikkert om det var ønskelig å kjøre systemet for innhentingen av dataen lokalt på anlegget eller om alt skulle kjøres i skyen. Denne løsningen gir også en løs kobling til APIet, som gjør at tjenesten kan benyttes mot andre kilder dersom dette er nødvendig uten å påvirke APIet. Tjenesten skal kun fungere som en klient som kun sender ut data fra en innlogget bruker som har en egen tilegnet rolle, **Data system**. Som vist i oversikten over hele arkitekturen i Figur 5 skal denne tjenesten kun sende data og ikke motta noe informasjon annet enn JWT-token for autorisering.

Tjenesten skal ha to funksjoner som kjøres ved jevne mellomrom: en funksjon for å sende informasjon fra Fôringssystemet til APIet og en funksjon for å sende logg filer fra SCADA-systemet til APIet. Det er viktig at denne tjenesten kun får tilgang til å gjøre disse oppgavene.

10 Implementasjon

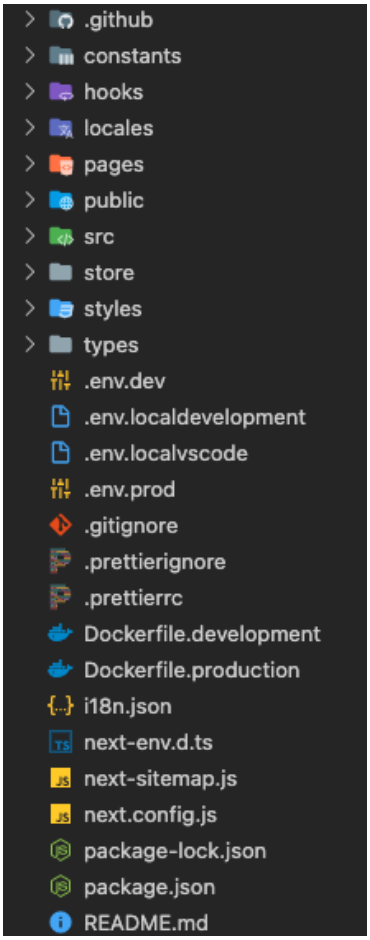
Til nå har det vært mye teori og begrunnelser for teknologiene og rammeverkene som er valgt, samt arkitekturen til systemet. Dette har vært nødvendig for å kunne forstå selve implementasjonen av løsningen, som vi vil gå nærmere inn på i dette kapittelet. Det er valgt å først gjennomgå implementasjonen av webapplikasjonen til systemet for å gi en bedre forståelse av hvordan dataen presenteres, før vi viser hvordan dataen bli håndtert i APIet og tjenesten for innhenting av informasjon. Merk at det er nødvendigvis ikke i denne rekkefølgen systemet er implementert, tjenestene er implementert i parallell.

10.1 Frontend

Klientsiden av systemet er som kjent skrevet i Next.js. Dette underkapittelet vil gå mer detaljert inn på implementasjonen av webapplikasjonen, der det først vil bli beskrevet litt overordnet om filstrukturen.

10.1.1 Filstruktur

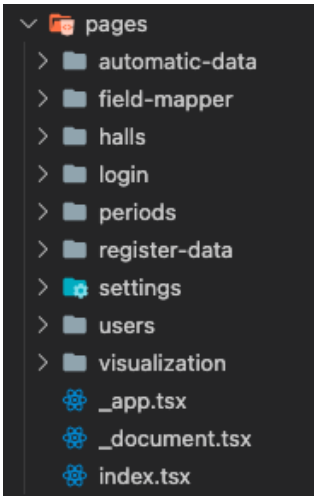
Figuren til høyre viser strukturen til webapplikasjonen. *pages*, *public*, *styles*, *node_modules*, *.gitignore*, *styles*, *package.json*, *package-lock.json* og *README.md* er autogenererte mapper og filer som kommer ved et nylig opprettet next.js prosjekt, mens resten er filer og mapper vi har opprettet i ettertid.



Figur 30: Filstruktur til Webapplikasjon

10.1.2 Sider

Pages mappen inneholder alle sidene til applikasjonen, der next.js automatisk genererer endepunktene ut ifra mappestrukturen her. For eksempel så vil filen ved lokasjon *pages/halls/index.ts* ha URLen *<domenet>/halls*.



Figur 31: filstruktur under pages mappen

Hovedkomponenten til applikasjonen er *_app.tsx*. Når brukeren besøker de forskjellige endepunktene, vil denne komponenten alltid ligge som ytterste lag til sidekomponenten som skal vises. Disse sidekomponentene skrives som helt vanlige React.js-komponenter.

```
1.   return (
2.     <StoreProvider store={store}>
3.       <ToastProvider components={{ ToastContainer }}>
4.         <ThemeProvider>
5.           <AuthComponent user={currentUser}>
6.             <Layout>
7.               <Component {...pageProps} />
8.             </Layout>
9.           </AuthComponent>
10.         </ThemeProvider>
11.       </ToastProvider>
12.     </StoreProvider>
13.   );
```

Returverdi til _app.tsx komponenten..

Ovenfor ser man alle innpakningskomponentene som blir brukt i *_app.tsx*. *Storeprovider* er en provider komponent som blir brukt for å håndtere global tilstand, noe som gås nærmere innpå i kapittel 10.1.4 Global tilstand nedenfor. *ToastProvider* er fra *react-toast-notifications* pakken, som gjør det mulig å enkelt vise varslinger i applikasjonen. *ThemeProvider* er en egendefinert komponent som i hovedsak tillater brukeren å veksle mellom mørkt og lyst modus i applikasjonen ved hjelp av Material-UI, samt sette opp egendefinert fargepalett og lignende. Mer om styling i kapittel 10.1.3 Styling. *AuthProvider* har kontroll på å holde nåværende autoriserte bruker oppdatert. *Layout* inneholder navigasjons- og appbaren til sidene.

10.1.3 Styling

Ettersom vi har valgt å bruke Material-UI konsistent i applikasjonen, har det vært lite fokus på egendefinert styling. Når det har vært behov for å tilpasse Material-UI-komponentene har vi som regel brukt Material-UI sin *makeStyles* funksjon, som tillater CSS lignende syntaks som JavaScript objekter. Ellers har det blitt brukt pålinje-styling (eng.: inline-styling) når det skal endres på småting som er avhengig av logikk.

10.1.4 Global tilstand

Som tidligere nevnt har vi valgt å bruke easy-peasy for å håndtere global tilstand. De fleste sidene har lokal tilstand som bare trengs på de sidene. I de tilfellende er det ikke blitt brukt easy-peasy for å holde koden oversiktlig. Vi har valgt å bruke global tilstand på data som blir brukt flere steder i applikasjonen, som nåværende innloggede bruker, perioder/innsett, kar og ekspandert meny. Dette er data som ville hatt en mye mer kompleks logikk og dårligere ytelse dersom det ikke ble brukt en global tilstandspakke som easy-peasy. I all hovedsak består global-tilstands logikken i applikasjonen av to filer: *store.js* og *userStore.ts*.

```
1. const storeModel: StoreModel = {
2.     drawerOpen: false,
3.     toggleDrawer: action((state, payload) => {
4.         state.drawerOpen = payload;
5.     }),
6.     userStore: userStore,
7.     periods: [],
8.     setPeriods: action((state, payload) => {
9.        state.periods = payload;
10.     }),
11.     pools: [],
12.     setPools: action((state, payload) => {
13.       state.pools = payload
14.     })
15. };
16.
17. const store = createStore(storeModel);
18. export default store;
```

Hoved tilstanden i store.ts filen.

Kodeutdraget ovenfor viser *storeModel* objektet som fungerer som hoved-storen. Ettersom *drawerOpen* (ekspandert meny), *periods* (innsett) og *pools* (kar) har ganske enkel logikk, er det valgt å ha håndteringen av disse dataene rett i hoved-storen selv om det strengt tatt skulle vært i en egen tilstand. Nåværende innloggede bruker har mer kompleks logikk, og det er derfor valgt å lage en egen tilstand på dette kalt *userStore*.

```
const userStore: UserStoreModel = {
    currentUser: null,
    isAuthed: false,
    isLoading: true,
    setCurrentUser: action((state, user) => {
        state.currentUser = user;
        state.isLoading = false;
        if (user != null) {
            state.isAuthed = true;
          } else {
            state.isAuthed = false;
        }
     }),
     checkIfUserIsAuthed: thunk(async (actions) => {
        try {
            let currentUser = await UserController.getUpdatedUser()
            actions.setCurrentUser(currentUser);
        } catch (error) {
            actions.setCurrentUser(null);
        }
     }),
};
export default userStore;
```

userSore objektet fra kildekoden.

10.1.5 HTTP klient

Det er valgt å lage abstraksjoner av HTTP forespørslene i appen for å ha mindre gjentakende kode samt gjøre det enklere å jobbe med. I *src/services/BaseApiService.ts* filen er hjelpeklassen for disse forespørslene, mens under *src/controllers* mappen ligger det egne filer som baserer seg på kontrollerstrukturen til APIet.

BaseApiService klassen har abstraksjoner av de vanligste HTTP metodene (GET, POST, PUT, PATCH, DELETE), der *axios* biblioteket blir brukt i grunn.

```
    post(
        url: string,
        data: object,
        ctx: any = null,
    ): Promise<AxiosResponse<any>> {
        return axios.post(Configs.serverAPI + url, data, {
            headers: {
                ContentType: "application/json",
                Authorization: this.headerConfig(ctx),
            },
        });
    }
```

Post-metoden fra BaseApiService fra kildekoden.

De fleste metodene i denne klassen er ganske like, og ovenfor kan du se utdraget av *post* metoden. Denne klassen tar hånd om å sette riktige HTTP headers og sette rett domene som forespørselen skal sendes til. Vi kan da bruke denne klassen som vist nedenfor i *src/controllers/UserController.js* filen:

```
1.      listUsers = async (): Promise<User[]> => {
2.          try {
3.              let response = await BaseApiService.get("User/listUsers")
4.              if (response.status !== 200) {
5.                  throw new CustomError("Api error", response.status);
6.              }
7.              return response.data;
8.          } catch (error) {
9.              throw error;
10.           }
11.       }
```

listUser-metoden fra UserController.js.

Her kan man sende et HTTP-forespørsel til rett endepunkt med riktig konfigurering, uten noe særlig gjentagende kode.

Kontrollerfilene under *src/controllers* mappen baserer seg på mye av samme håndtering av HTTP forespørsler som vist i kodesnutten ovenfor. Det som går igjen i metodene i disse klassene er at det først blir sendt en HTTP-forespørsel, deretter blir det sjekket om responsen har ønsket statuskode, så blir dataen returnert. Vi kan da bruke kontrollerklassene som vist under, og her er det viktig med feilhåndtering ettersom metodene kan returnere feilmeldinger.

```
1.      try {
2.        const res = await UserController.listUsers();
3.        setUsers(res);
4.      } catch (error) {
5.        console.error(error);
6.      }
```

Eksempel på bruk av UserController.listUser().

10.1.6 Feilhåndtering

For å løse løfter (eng.: resolve promises) i appen, er det brukt *try/catch* metoden. For å generalisere måten vi håndterer feil på er det valgt å lage en egen React-hook kalt *useErrorHandler*, som blir brukt der feil blir fanget opp. Denne React-hooken sjekker statuskoden som er i feilmeldingen, og gir brukeren en feilmelding basert på dette. Nedenfor ser man først er lite utdrag av hovedfunksjonaliteten til *useErrorHandler*, samt et eksempel på hvordan vi bruker denne React-hooken.

```
1. let errorMessage = '';
2. let appearance: AppearanceTypes = 'error'
3.
4. // Unauthorized
5. if (error.response?.status === 401) {
6.     errorMessage = t('unauthorized');
7. }
8. // Length Reqired
9. if (error.response?.status === 411) {
10.      errorMessage = t('password-format-error');
11.  }

  ... // mer kode som ikke er vist her

12.  // Error not handled by the checks above
13.  if (errorMessage == '') {
14.      errorMessage = t('horrible-wrong');
15.  }
16.  addToast(errorMessage, { appearance: appearance, autoDismiss: true
   });
```

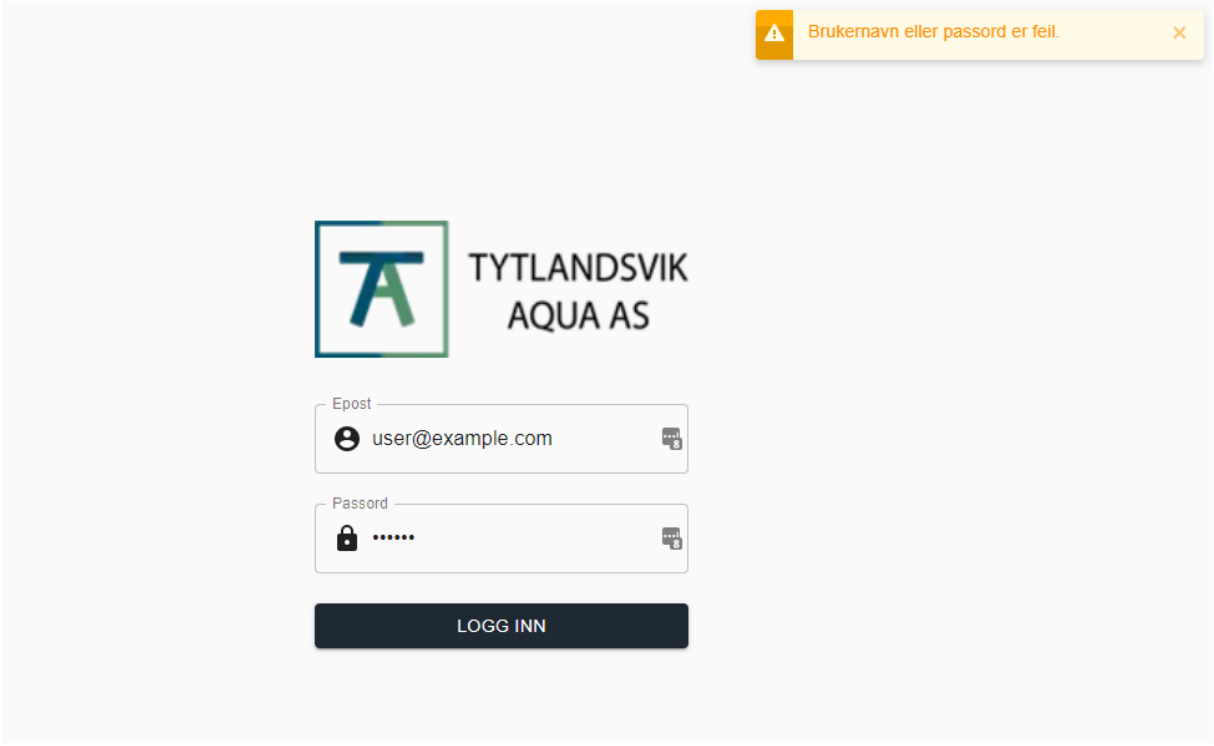
Kodeutsnitt av hoved funksjonaliteten til useErrorHandler react- hooken. Hentet fra hooks/errorHook.ts

```
1. // initialisering av hooken i en komponent
2. const handleError = useErrorHandler();

   ... // mer kode som ikke er vist her

3. const onRowDelete = async (oldData: any) => {
4.   try {
5.     await PeriodController.delete(oldData);

       ... // mer kode som ikke er vist her

6.   } catch (error) {
7.     // kallet til hooken
8.     handleError(error);
9.   }
10.  };
```

Kodeutsnitt av en komponent som bruker denne hooken i en funksjon som sender en forespørsel til APIet. Hentet fra «pages/periods/index.ts»

10.1.7 Autentisering

Brukeren blir autentisert gjennom en vanlig innloggingsform med e-post og passord, som sender en POST-forespørsel til APIet. Dersom brukeren skriver inn enten feil brukernavn eller passord så blir det returnert en dårlig forespørsel, HTTP-statuskode 400, og det blir gitt en feilmelding til brukeren, se figur 32 under. Skriver brukeren derimot inn riktig passord blir brukeren lagret som *currentUser*, som er en global tilstandsvariabel, og blir deretter omdirigert til dashbordet i hovedsidene. Det blir også lagret en API-token (JWT) i en informasjonskapsel som blir brukt ved HTTP-forespørsler.



Figur 32: Skjermbilde av innloggingssiden med feil brukernavn/passord

Når brukeren åpner nettsiden på nytt, etter å tidligere ha vært logget inn, blir det sendt en GET-forespørsel til endepunktet */user* som returnerer nåværende innloggede bruker basert på API-token. Hvis det blir returnert en 401-HTTP-statuskode (uautorisert), som vil si at API-token enten ikke finnes eller er ugyldig, vil informasjonskapselen bli slettet og brukeren blir omdirigert til innloggingssiden. Denne sjekken blir gjort på *getInitialProps* funksjonen til hovedkomponenten i applikasjonen, som er en funksjon som blir kjørt hver gang brukeren omdirigerer eller laster siden på ny.

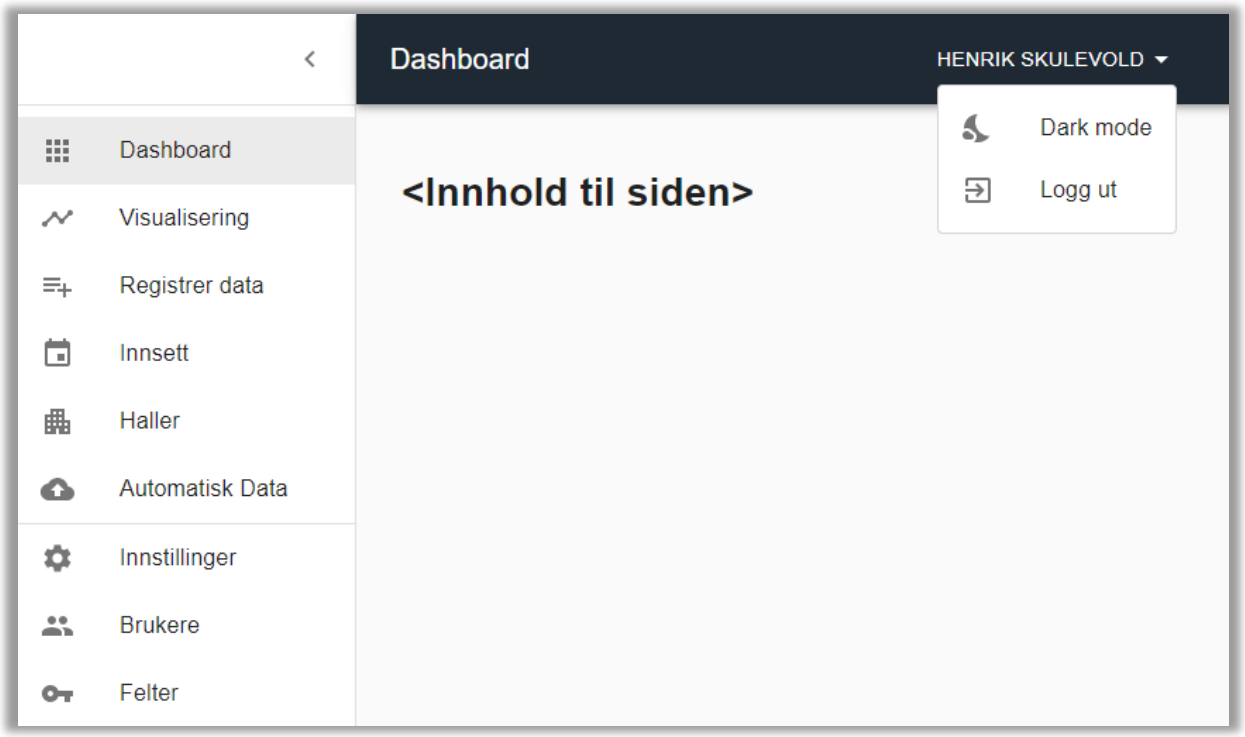
```
1. MyApp.getInitialProps = async ({ ctx }) => {
2.   const isOnLogin = ctx.pathname === '/login';
3.   let currentUser = null;
4.   try {
5.     currentUser = await UserController.getUpdatedUser(ctx);
6.
7.     if (isOnLogin && currentUser) {
8.       ctx.res.writeHead(302, {
9.         Location: '/'
10.        });
11.        ctx.res.end();
12.      } else if (!isOnLogin && !currentUser) {
13.        ctx.res.writeHead(302, {
14.          Location: '/login'
15.        });
16.        ctx.res.end();
17.      }
18.    } catch (error) {
19.      if (!isOnLogin) {
20.        ctx.res.writeHead(302, {
21.          Location: '/login'
22.        });
23.        ctx.res.end();
24.      }
25.    }
26.  return { currentUser };
27.  };
28.
```

getInitialProps funksjonen til hovedkomponenten i applikasjonen.

For at denne funksjonen ikke skal havne i en evig løkke, er det også lagt inn en sjekk i koden ovenfor på om brukeren er på innloggingsiden eller ikke. En bedre løsning ville vært å ha denne funksjonen på oppstart eller når brukeren laster siden på nytt, men ettersom det vil være lite brukere i denne applikasjonen så vil det ikke ha mye å si på ytelsen at denne funksjonen også blir kjørt ved omdirigeringer.

10.1.8 Layout og navigasjon

Hovedsidene er alle innpakket med en navigasjonsbar og toppbar. Toppbaren inneholder tittelen på siden og utloggingsmulighet, mens navigasjonsbaren blir brukt til å navigere mellom sidene. Dette er implementert i en egen komponent kalt *Layout*¸ slik at vi kan bruke det som en forelder-komponent til sidene. I figuren nedenfor kan man se hvordan dette ser ut for brukeren.



Figur 33: Navigasjonsbaren og toppbaren til hovedsidene

For å gjøre utviklingen av denne komponenten enklest mulig, er mye av utseende inspirert fra eksempler til Material-UI sin dokumentasjon. Implementasjonen av komponenten vil derfor ikke gjennomgås i detalj. Det er derimot lagt inn sjekker i navigasjonsbaren basert på autoriseringsgraden til brukeren, som bestemmer hvilke navigasjonselementer som skal vises til brukeren. Dette er vist i kodeutsnittet nedenfor, der man avbilder (eng.: mapper) en liste med objekter til listeelementer bare hvis rollen til brukeren er høy nok:

```
{[
  { title: t('dashboard'), path: '/', icon: <Apps />, role: ROLE.READER },
  { title: t('visualization'), path: '/visualization', icon: <TimelineOutlinedIcon />, role: ROLE.READER },
  { title: t('register-data'), path: '/register-data', icon: <PlaylistAddOutlinedIcon />, role: ROLE.READER },
  { title: t('insert'), path: '/periods', icon: <EventIcon />, role: ROLE.READER },
  { title: t('halls'), path: '/halls', icon: <ApartmentIcon />, role: ROLE.READER },
  { title: 'Automatisk Data', path: '/automatic-data', icon: <CloudUploadIcon />, role: ROLE.WRITER }
].map((object, index) => {
   // Autoriseringsgrad sjekken
  if (object.role <= currentUser?.role) {
    return (
      <ListItem
        button
        key={object.title}
        onClick={() => {
          router.push(object.path);
        }}
        selected={router.pathname == object.path}>
        <ListItemIcon>{object.icon}</ListItemIcon>
        <ListItemText primary={object.title} />
      </ListItem>
    );
  }
})}
```

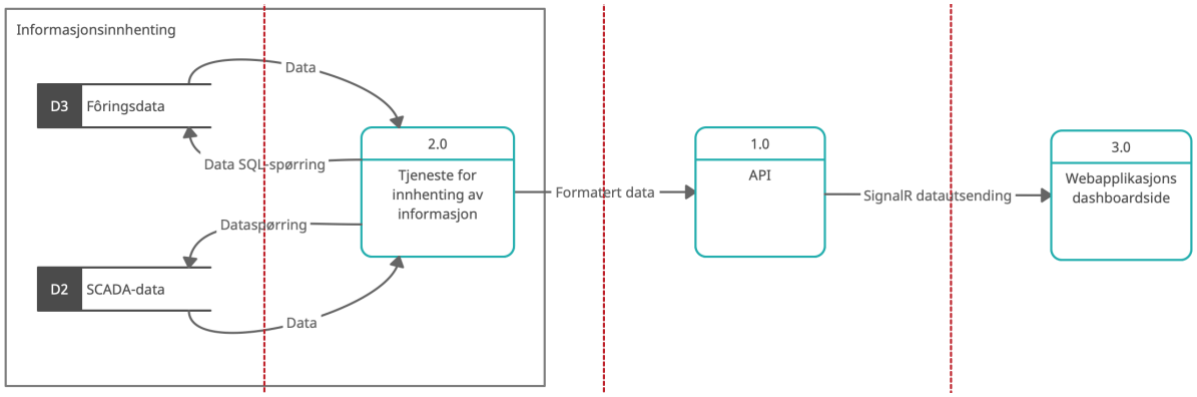
Mappe av liste med objekter til listeelementer, der det også er en autoriseringsgradsjekk. Kode hentet fra src/componentsl/Layout.tsx.

10.1.9 Dashbord

Første siden man ser når man logger inn er dashbordet. Tanken med å ha et dashbord er å ha en enkel førsteside som brukeren kommer inn på med generelle statuser. På denne siden ble det derfor i første omgang valgt å ha en graf med sanntidsdata for å tilfredsstille Krav 3. Krav 3, som omhandler at webapplikasjonen skal vise sanntidsdata, er ikke et krav som kan direkte utledes fra problemstillingen, men var et ønske som kom opp tidlig i utviklingen.

Sanntidsdataen blir sendt via SignalR, og presentert via grafbiblioteket Apexchart. Selve datamanipuleringen og presenteringen av dataen via Apexchart vil ikke bli gjennomgått i detalj før 10.1.13 Visualisering av målinger. Dette fordi visualiseringen av data på disse sidene er veldig like, men visualiseringssiden har flere filtreringsmuligheter og det vil derfor være mer hensiktsmessig å gjennomgå denne logikken da. Det vil derfor heller fokuseres på håndteringen av sanntidsdata i dette delkapittelet.

Dataen vi får inn fra Tytlandsvik Aqua automatisk er som kjent fra fôrings- og SCADA-systemet. Når denne dataen blir sendt til APIet, trigges det et event som sender dataen videre til klienter som er koblet til SignalR tjenesten. Hvordan eventet trigges skrives om i 10.2.7 Sanntidsutsending av data. Påfølgende figur viser dataflyten ved sending av sanntidsdata.



Figur 34: Dataflyt som trigger sending av sanntidsdata. Når data legges inn fra fôrings- og SCADA-systemet sendes sanntidsdata til dashboardet i webapplikasjonen ved hjelp av SignalR.

Det er valgt å lage en egen hjelpeklasse, kalt *LiveChartHandler*, for å håndtere lytting til ny data samt det å oppdatere selve grafen. Før man kan få inn sanntidsdata, må man også koble seg til SignalR-tjenesten til APIet. Dette gjøres i *UseEffect* hooken til dashbordsiden, og metoden til *LiveChartHandler* som trigges vises under. Koden er beskrevet i kommentarene.

```
1. static listenAndUpdate = () => {
2.     const ApexCharts = require('apexcharts');
3.     // Kobler seg til SignalR-tjenesten til APIet
4.     let connection = new signalR.HubConnectionBuilder()
5.         .withUrl(`${Configs.serverAPI}datahub`, {
6.             skipNegotiation: true,
7.             transport: signalR.HttpTransportType.WebSockets,
8.         })
9.         .build();
10.
11.      // Kaller metodene som lytter til data fra SCADA- og fôrings-
12.      // systemet
13.      LiveChartHandler.listenOnRasLog(connection);
14.      LiveChartHandler.listenOnFeedingLog(connection);
15.
16.      // setter opp intervall som kalles hvert andre sekund.
17.      // Dette intervallet oppdaterer grafen gjevnlig
18.      const interval = setInterval(() => {
19.          if (LiveChartHandler.chartIsUpdating) {
20.              // Oppdatering av graf har blitt trigget en annen
21.              // plass, og blir skippet her.
22.              LiveChartHandler.chartIsUpdating = false;
23.          } else {
24.              if (LiveChartHandler.series.length > 0) {
25.                  // Oppdaterer grafen med sanntidsfunksjon til
26.                  // Apexcharts ved bruk av variabelen
27.                  // LiveChartHandler.series.
28.                  ApexCharts.exec('realtime', 'updateSeries', LiveCha
   rtHandler.series);
29.              }
30.          }
31.      }, 2000);
32.      connection.start();
33.      // Statiske variabler som brukes når tilkobling skal avsluttes
34.      LiveChartHandler.connection = connection;
35.      LiveChartHandler.interval = interval;
36.  };
```

ListenAndUpdate metoden til LiveChartHandler. Her er all initialiserings kode som trengs for å lytte til ny data..

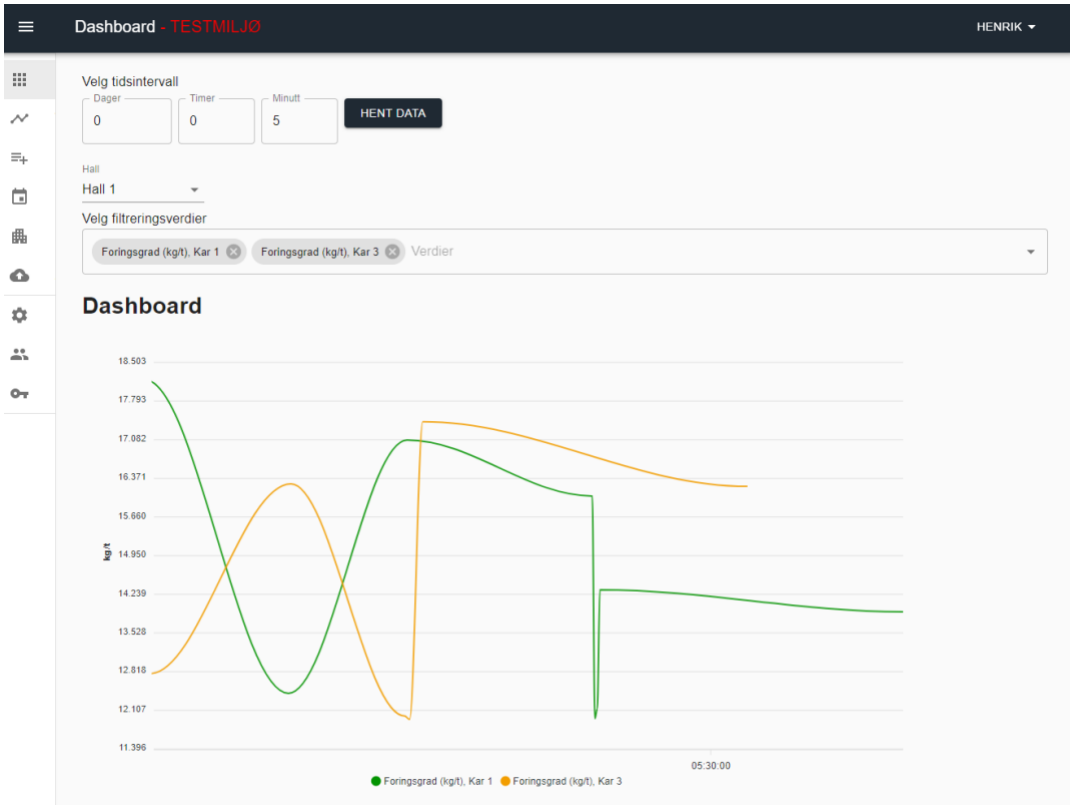
Kode hentet fra src/utils/LiveChartHandler.ts.

For at grafen skal vise ny data, må variabelen *LiveChartHandler.series* oppdateres. Dette gjøres hver gang ny data mottas fra APIet, under vises hvordan man kobler seg til et spesifikk event. *LiveChartHandler.PushFeedingLogs* tar for seg selve datamanipuleringen, og oppdaterer *LiveChartHandler.series*.

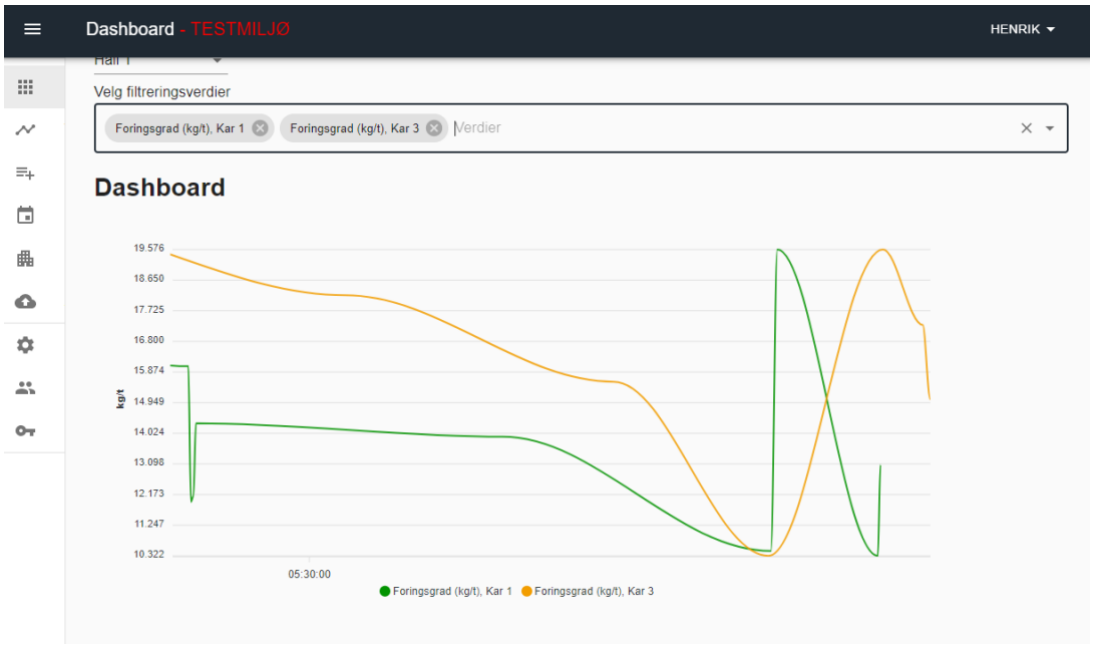
```
1. static listenOnFeedingLog = (connection) => {
2.     connection.on('ReceiveFeedingLogs', (data: FeedingLog[]) => {
3.         LiveChartHandler.pushFeedingLogs(data);
4.     });
5. }
```

ListenOnFeedingLog metoden til LiveChartHandler. Her motas det ny fôringslogg data hver gang det hentes inn til APIet. Kode hentet fra src/utils/LiveChartHandler.ts.

Når brukeren er på dashbordsiden, kan en velge tidsintervall samt hvilke verdier man vil få oppdatert data fra. De påfølgende figurene viser hvordan siden sees ut for brukeren, samt hvordan grafen har endret seg automatisk når ny data har kommet inn:



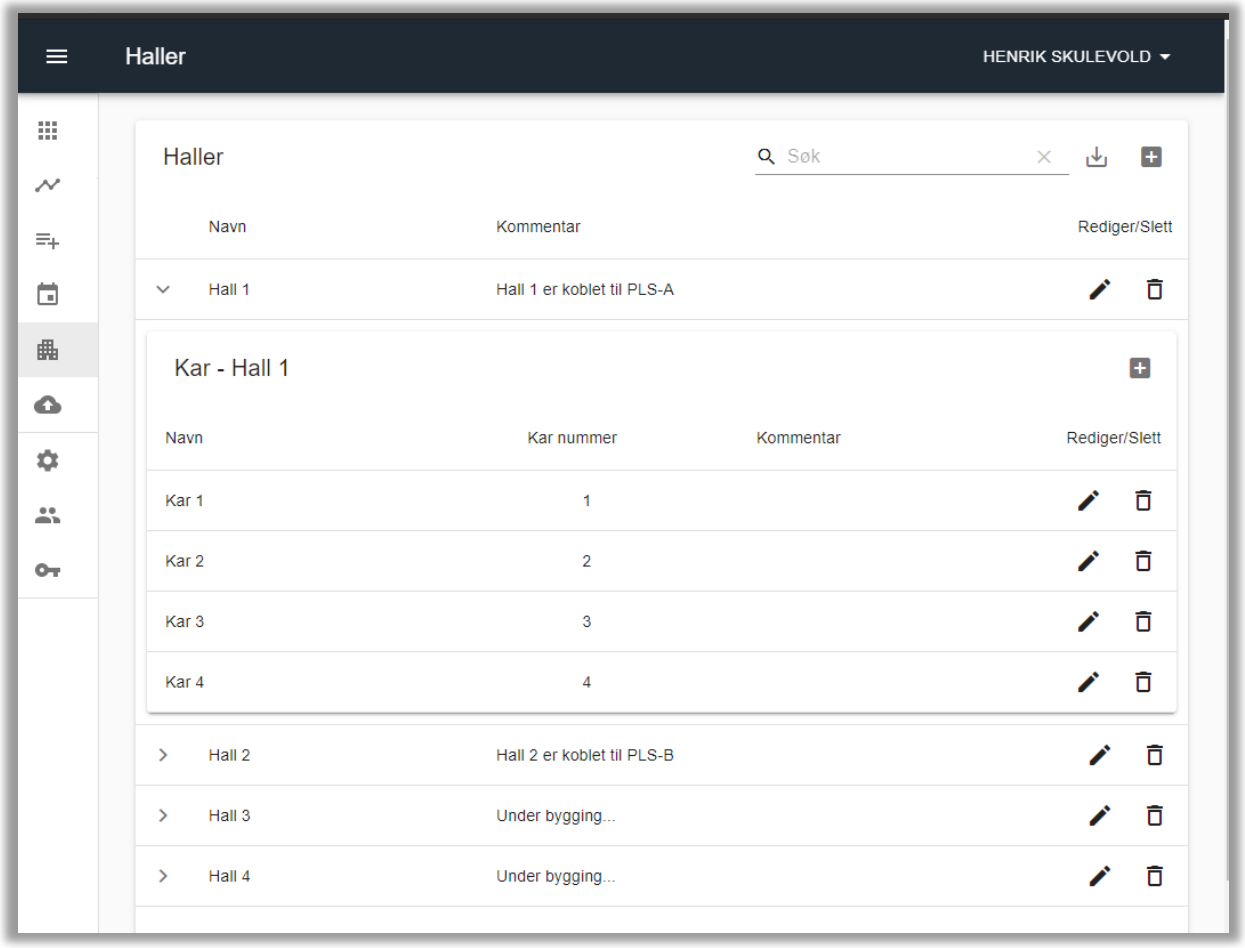
Figur 35: Skjermbilde av livegraf på dashbordsiden - del 1. I dette eksempelet får man oppdatert data siste 5 minutter. Merk: dette er testdata.



Figur 36: Skjermbilde av livegraf på dashbordsiden - del 2. Etter ny data har blitt mottatt, har grafen oppdatert seg av seg selv i sanntid. Merk: dette er testdata.

10.1.10 Haller og kar

Det er et indirekte nødvendig krav utledet fra krav 1 å ha oversikt over haller og kar for å kunne bruke de til å sortere innsett, foringsdata og SCADA-logger. For å gjøre applikasjonen mest mulig fleksibel for fremtidig drift og for å gjøre systemet mer justerbart, er det lagt til en egen side der brukeren selv kan legge til haller og kar. Dette er gjort istedenfor at det skal hardkodes. Her må man være administrator for å kunne endre/legge til haller/kar.



Figur 37: Skjermbilde an hallsiden. Her ser man en liste over hallene, samt underliste av karene til hallene.

Det er valgt å bruke Material Table for å håndtere disse listene, ettersom pakken tillater CRUD-operasjoner rett ut av boksen. Denne tabellen er også intuitiv for brukeren, noe som tilfredsstiller krav 14. Listen over haller og dens kar blir lagret i en lokal tilstandsvariabel, ettersom denne dataen ikke trengs å deles med utenforstående komponenter. Det blir brukt *useAsyncEffect* hook for å hente dataen den første gangen komponenten bygges. Under ser du koden på hvordan dette er gjort, samt endring av *loading* tilstanden (som viser en spinner når den er *true*).

```
1.   useAsyncEffect(async () => {
2.     try {
3.       setLoading(true);
4.       const halls = await HallController.getAll();
5.       setHalls(halls);
6.     } catch (error) {
7.       handleError(error);
8.     } finally {
9.       setLoading(false);
10.      }
11.    }, []);
```

Henting av halldata. Kode hentet fra pages/hall/indet.tsx.

For å kunne ha en liste med kar som en underliste til en spesifikk hall, kan man returnere en komponent ved attributtet *detailPanel* til MaterialTable komponenten, som vist fra linje 10 i kodeutsnittet nedenfor.

```
1. <MaterialTable
2.  isLoading={loading}
3.  title="Haller"
4.  columns={columns}
5.  data={halls}
6.  detailPanel={(rowData) => {
7.    if (rowData.pools === undefined) {
8.      rowData.pools = [];
9.    }
10.     return (
11.       // kar-liste komponenten
12.       <PoolTable pools={rowData.pools} hallId={rowData.id} hallname=
   {rowData.name} updatePools={updatePools} />
13.     );
14.   }}
15.   editable={{
16.     onRowAdd: (newData: Hall) => ...
17.     onRowUpdate: (newData: Hall, oldData) => ...
18.     onRowDelete: (oldData: Hall | any) =>...
19.   }}
20.   localization= ...
21.   options= ...
22.  }
23.  />
```

Visning av halldata ved bruk av Material Table. Irrelevant kode erstattet med <…>. Kode hentet fra pages/hall/indet.tsx.

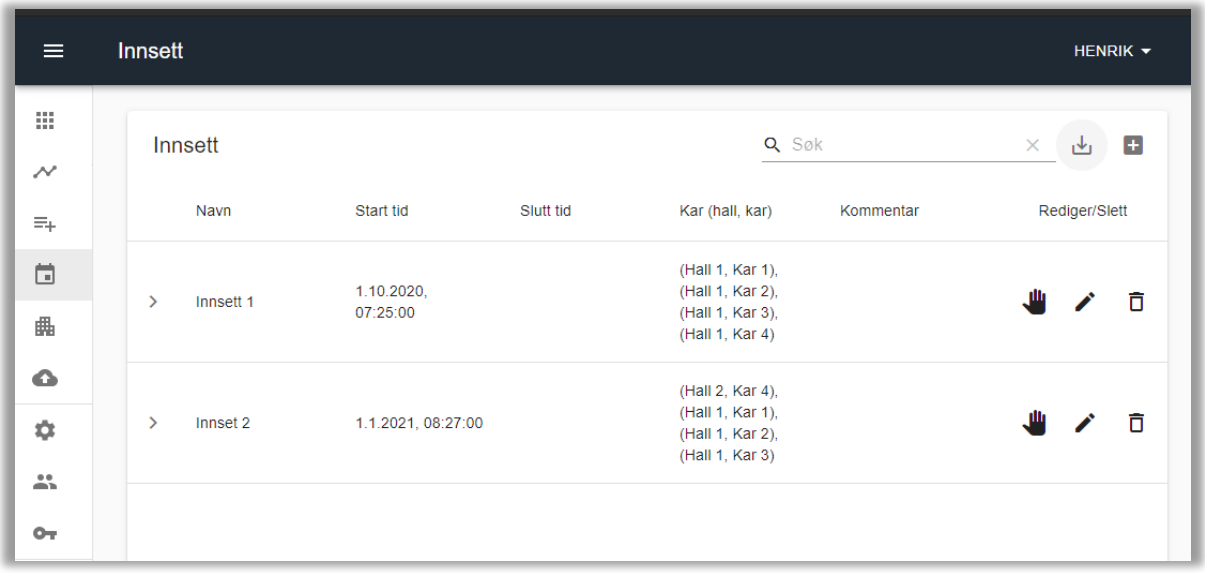
onRowAdd, *onRowDelete* og *onRowDelete* fra kodeutsnittet ovenfor er event-funksjoner som blir trigget når brukeren enten legger til, endrer eller sletter en hall. Disse funksjonene har blitt implementert overalt der Material Table komponenten blir tatt i bruk, og oppgavene til funksjonene er veldig like: Først sende en forespørsel til APIet, deretter endre på tilstanden i appen og gi tilbakemelding til brukeren.

```
1. onRowUpdate: (newData: Hall, oldData) =>
2.   new Promise(async (resolve, reject) => {
3.     try {
4.       // API forespørsel
5.       await HallController.update({ id: newData.id, name: newData.nam
   e, comment: newData.comment });
6.
7.       // endring av tilstand
8.       const dataUpdate = [...halls];
9.       const index = oldData.tableData.id;
10.       dataUpdate[index] = newData;
11.        setHalls([...dataUpdate]);
12.
13.        // toastbar tilbakemelding til brukeren
14.        addToast('Hall ble oppdatert.', { appearance: 'success', auto
   Dismiss: true });
15.        resolve({});
16.      } catch (err) {
17.        // kall på feilhåndterings hook
18.        handleError(err);
19.        reject({});
20.      }
21.    }),
```

Visning av halldata ved bruk av Material Table. Irrelevant kode erstattet med (…). Kode hentet fra pages/hall/indet.tsx.

10.1.11 Innsett og tidslinje

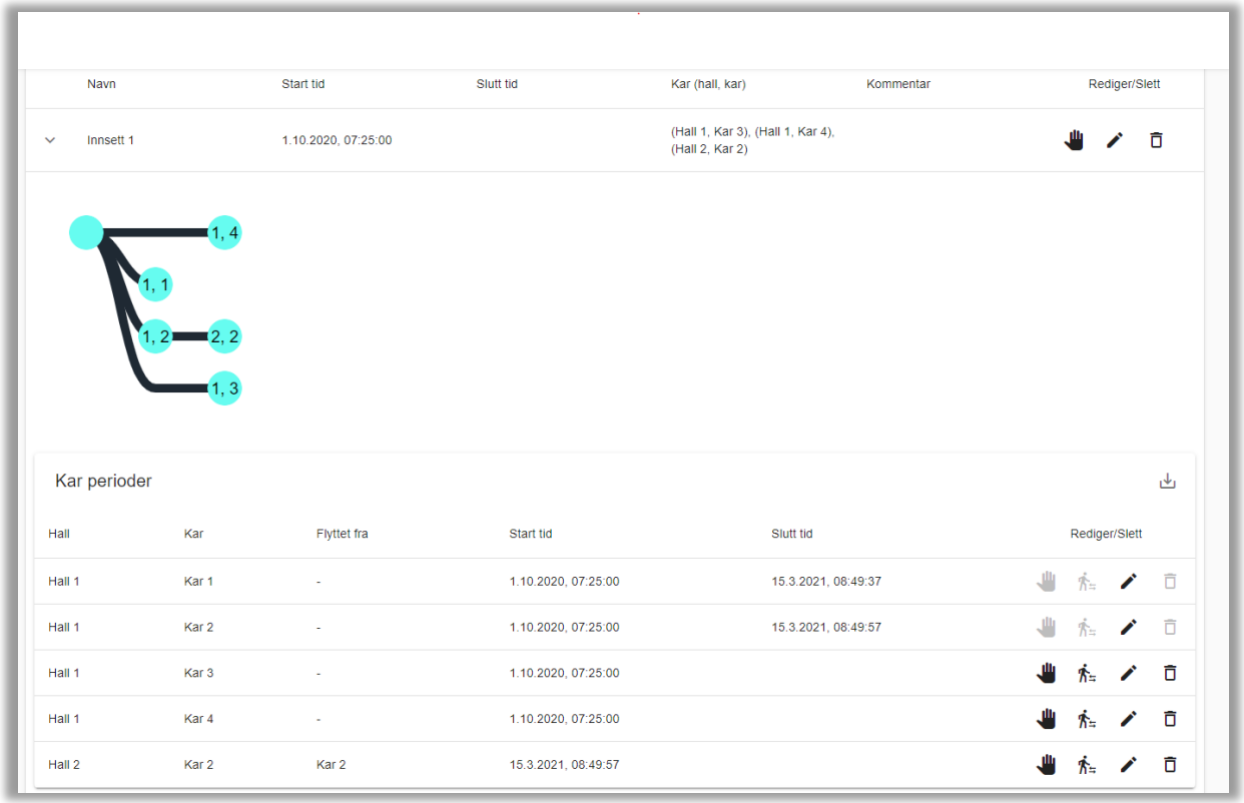
Når Tytlandsvik Aqua får inn en ny leveranse med fisk, så blir dette ført opp som et nytt innsett. Et innsett har en start tid, og etter hvert får det en slutt tid. Ettersom loggføringen av et innsett er sentralt for et oppdrettsanlegg, er dette en nødvendig del å ha med i systemet. Det er også nødvendig for å imøtekomme krav 1 ettersom dataen skal kunne filtreres på egendefinerte innsett.



Figur 38: Skjermbilde av Innsettsiden.

I figur 38 kan man se innsettsiden. Her er det også brukt komponenten Material Table for å enkelt kunne holde oversikt over dataen i en fleksibel tabell. Implementasjonen av denne er særdeles lik som i kapittelet ovenfor 10.1.10 Haller og kar. I kapittel 10.1.13 Visualisering av målinger nedenfor kan man se hvordan man filtrerer dataen basert på et innsett.

Det er et krav (Krav 5) fra Tytlandsvik Aqua som handler om at man skal ha mulighet til å spore hvilke kar fisken har vært i løpet av et innsett, som også er beskrevet i 9.5 Database ER-Diagram kapittelet. For å løse dette problemet visuelt, ble det valgt å implementere en slags tidslinje basert på hvilke kar fisken har vært i. Det ble også valgt å bruke komponenten *Material Table* her for å kunne redigere på denne historikken, samt flytte fisken til et nytt kar. Nedenfor i figur 39 kan man se hvordan dette ser ut i grensesnittet.



Figur 39: Skjermbilde av visualisering av historikken til et innsett

Tidslinjen øverst i figur 39 viser et innsett som starter med fisk i kar 1, 2, 3, og 4 i hall 1, der fisken i hall 1 kar 2 blir flyttet til hall 2 kar 2 den 15.03.2021. Tabellen nederst på siden i figuren har dataene som blir presentert i tidslinjen, samt det er her brukeren kan endre på en karperiode, eller eventuelt flytte fisk fra et kar til et annet. Denne dataen ligger i den litt mer kompliserte tabellen *PoolPeriods* som ble beskrevet i kapittelet 9.5 Database ER-Diagram.

For å finne en god måte å visualisere historikken på, ble dataen organisert i en trestruktur. Der rotnoden er leverandøren av fisk og de resterende nodene er kar i de ulike hallene. Nodene kan ha flere barn, fordi fisk fra leverandøren og kar kan splittes ut i flere kar. I tidslinjen i figur 39 ser man at rotnoden (leverandørenoden) har fire barn, fisken er altså splittet ut til fire forskjellige kar. Karnode (1, 2) har bare ett barn, som vil si at fisken er flyttet fra (1, 2) til (2, 2). Ved å følge en slik struktur vil fisken som kom fra leverandøren alltid befinne seg i bladnodene. Det vil si at fisken er i karene som er i enden av en gren. Alle andre karnoder representerer kar fisken har vært i, men ikke lenger er i. Høyden på treet kan være ubestemt høyt fordi man kan flytte fisken ubestemt antall ganger.

I kildekoden blir tidslinjen strukturert i et nøstet objekt. Det vil si at en node inneholder et felt med barn. Disse barnene er nye noder som igjen har et felt med barn. På denne måten vil det støtte muligheten til å ha flere barn i tillegg til at høyden på treet kan være ubestemt. Nodene blir konstruert av data fra *PoolPeriods* tabellen. Dataen fra denne tabellen blir hentet med et API kall hvor det kun er de relevante karperiodene som blir hentet ut. Det er altså de karperiodene som hører til det innsettet som blir sett på. Det blir laget en node for hver av karperiodene. Dette blir gjort på følgende måte:

```
1. period.poolPeriods.forEach((pp) => {
2.       // De første karnodene etter leverandørnoden(roten) har ingen
   foreldre og blir derfor lagt til med en gang.
3.       if (pp.parent.id == 0) {
4.         newRootNode.children.push({ name: pp.pool.name, branch: null,
   children: [], id: pp.id, hall: pp.pool.hall });
5.       } else {
6.         // Ser på de gjennværende bassengperiodene og setter de som
   barnnoder til forelderen de referer til
7.         let parentNode: PoolPeriodNode = searchTreeForId(newRootNode,
   pp.parent.id);
8.         if (parentNode != null) {
9.           parentNode.children.push({ name: pp.pool.name, branch:
   null, children: [], id: pp.id, hall: pp.pool.hall });
10.          }
11.        }
12.      });
13.  setRootNode(newRootNode);
```

Kode hentet fra src/components/PoolPeriodGraph.tsx.

searchTreeForId funksjonen ovenfor gjør et rekursivt søk gjennom treet etter en gitt ID. Dette blir gjort rekursivt da det eliminerer behovet for å vite om høyden på treet. ID-en i dette tilfellet er ID-en til forelderen til den aktuelle noden. Treet vil alltid ha de forelderen som

barna referer til. Grunnen til det er fordi listen *poolPeriods* alltid har foreldrene først, som vil si at foreldrene blir lagt inn i treet før barna sine. Hvis ikke foreldrene hadde blitt lagt inn først hadde heller ikke barna blitt lagt inn. Når treet er ferdigkonstruert kan følgende være ett eksempel på strukturen til treet:

```
{
  name: 'levrandør',
  children: [
    {
      name: 'Kar 1',
      children: [],
      branch: null,
      id: 1,
      hall: 1

    },
    {
      name: 'Kar 2',
      children: [
        {
          name: 'Kar 4',
          children: [],
          branch: null,
          id: 4,
          hall: 1

        },
      ],
      branch: null,
      id: 2,
      hall: 1

    },
    {
      name: 'Kar 3',
      children: [],
      branch: null,
      id: 3,
      hall: 1


    }
  ],
  branch: null,
  id: 5,
  hall: null

};
```

Eksempel på strukturen til et nøstet objekt

Når dataen er strukturert i nøstede objekter kan den visualiseres ved hjelp av tredjepartspakken GitGraph. GitGraph lager noder ved å kalle *commit* funskjonen på en branch. Rotnoden blir laget på denne måten:

```
1. // Leverandør -grein og -node
2.  const main = gitgraph.branch(rootNode.id.toString());
3.  main.commit({
4.     subject: rootNode.name,
5.     renderTooltip: renderTooltip,
6.     onClick: onClickCommit
7. });
```

Kode hentet fra src/components/PoolPeriodGraph.tsx.

De resterende nodene blir laget rekursivt for å eliminere behovet for å vite høyden til treet. Nodene blir laget på denne måten:

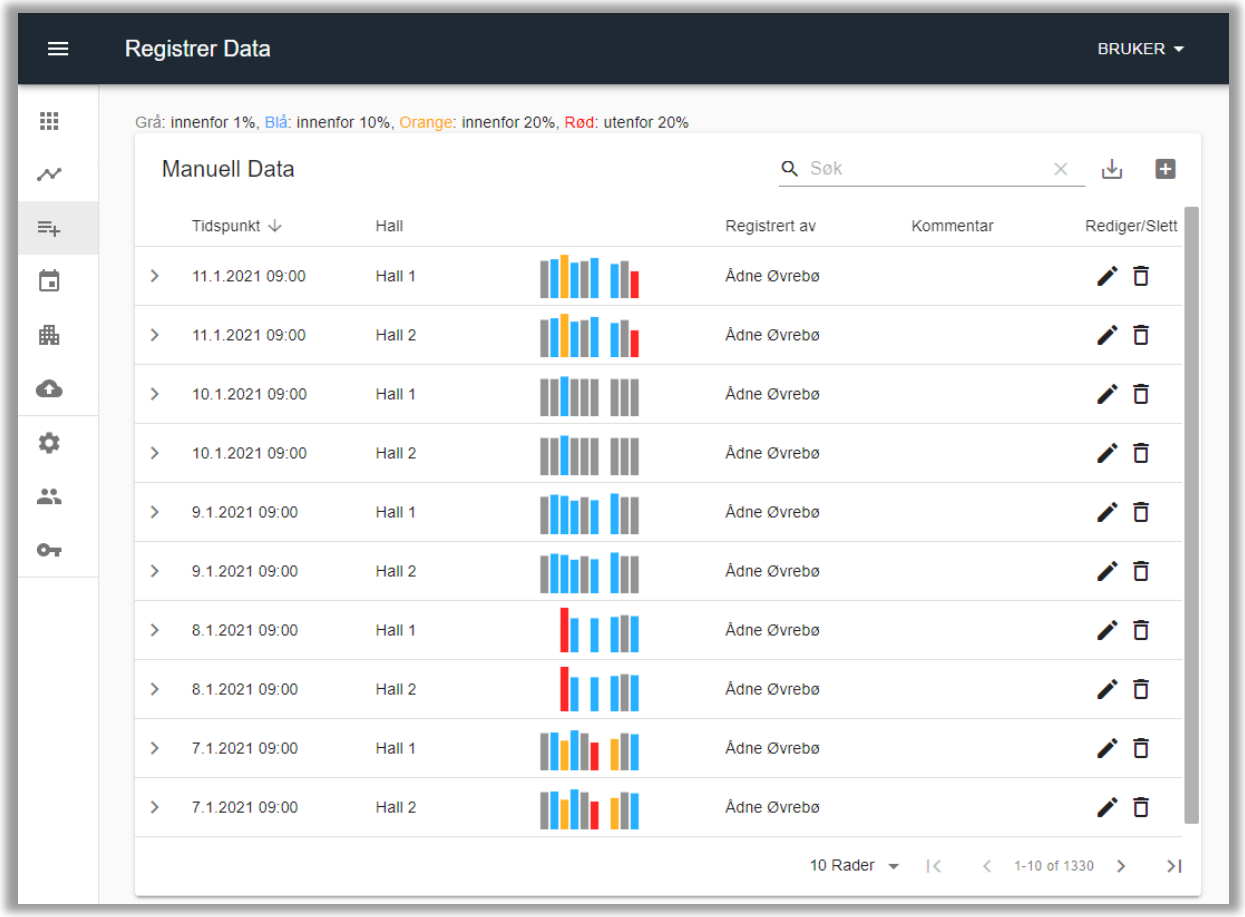
```
1.  const itterateChildren = (node: PoolPeriodNode) => {
2.   // Looper children til noden
3.   node.children.forEach((ppn, idx) => {
4.   // Sjekk om noden skal fortsette på samme grein eller lage en ny
5.     let branch;
6.     if (node.children.length > 1 && idx != node.children.length - 1)
   {
7.       branch = node.branch.branch(ppn.id.toString());
8.     } else {
9.       branch = node.branch;
10.      }
11.      // Legg til noden på valgt grein
12.      branch.commit({
13.        subject: ppn.hall.name + ', ' + ppn.name,
14.        renderTooltip: renderTooltip,
15.        onClick: onClickCommit,
16.        dotText: ppn.hall.name[ppn.hall.name.length - 1] + ', ' +
   ppn.name[ppn.name.length - 1]
17.      });
18.      ppn.branch = branch;
19.      // Det rekursive kallet, rekursjonen vil stoppe hvis ppn ikke
   har noen children
20.      itterateChildren(ppn);
21.    });
22.  };
```

Kode hentet fra src/components/PoolPeriodGraph.tsx.

Hvis et kar splittes, det vil si at man har to barn, lages det nye greiner og hvert barn blir roten av de nye greinene. Et unntak for dette er det første barnet som bare vil fortsette på forelderens grein. I koden over blir dette sjekket av if-sjekken på linje 6. I figur 39 ser vi dette ved at karnode (2, 2) ikke har en ny egen grein, men fortsetter på greinen til kar (1, 2).

10.1.12 Registrering av manuell data

Som tidligere nevnt gjør de med biologisk ansvar hos Tytlandsvik Aqua manuelle vannmålinger, som SCADA-systemet ikke fanger opp, og fører dem inn i et Excel-ark. Et av ønskene fra Tytlandsvik Aqua var at dette systemet skal erstatte Excel-arket ved at de manuelle målingene skal kunne føres rett inn her, noe som også representerer krav 2. Under kan man se siden for håndteringen av disse målingene



Figur 40: Skjermbilde av Registrer Data siden, som viser en liste med manuell data.

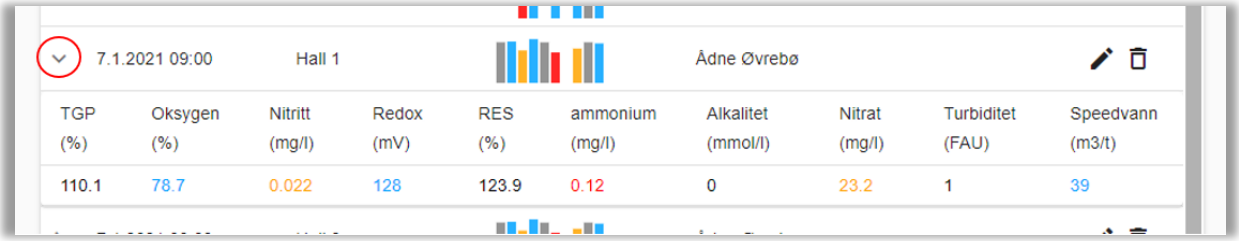
For å holde implementasjon og visuell fremstilling av data konsistent, er det også her valgt å benytte Material Table komponenten for å presentere manuelt logget data. Det som skiller seg ut i denne tabellen fremfor dem som tidligere er presentert, er søylediagrammet som er i midten av hver rad. Hensikten med dette diagrammet er å gi brukeren en enkel oversikt over alle parameterne til målingene. Det er lagt inn fargekoder på søylene som skal si noe om hvor ekstrem en verdi er i forhold til en brukerdefinert idealverdi, da dette kan være en motiverende faktor for at brukeren velger å nærmere undersøke en måling fremfor andre. Det er her valgt å bruke biblioteket Apexchart for å presentere diagrammet, et bibliotek som også

blir brukt andre steder i applikasjonen. Under vises hvordan hver rad i tabellen manipulerer dataen og returnerer en *Chart*-komponent.

```
1. {
2.   title: '',
3.   render: (row: ManualInput) => {
4.     const newData = [];
5.     // Options er et objekt med spesifikasjon
6.     // til hvordan grafen skal se ut
7.     const optionsCopy = deepcopy(options);
8.     dataKeys
9.         // filtrerer datakeys (alle parameterne det kan filtreres på
10.        // i visualiseringssiden) til å returnere bare som er for
11.        // manuell data
12.        .filter((key) => key.source === Source.Manual)
13.        .forEach((key) => {
14.          // Legger til verdien som presenteres i diagrammet
15.          // ut ifra idealverdi
16.          newData.push((row[key.key] / key.idealValue) * 100);
17.        });
18.      optionsCopy.series[0].data = newData;
19.      return <Chart width="90px" height="40px" options={optionsCopy}
   series={optionsCopy.series} type="bar" />;
20.    }
21.  },
```

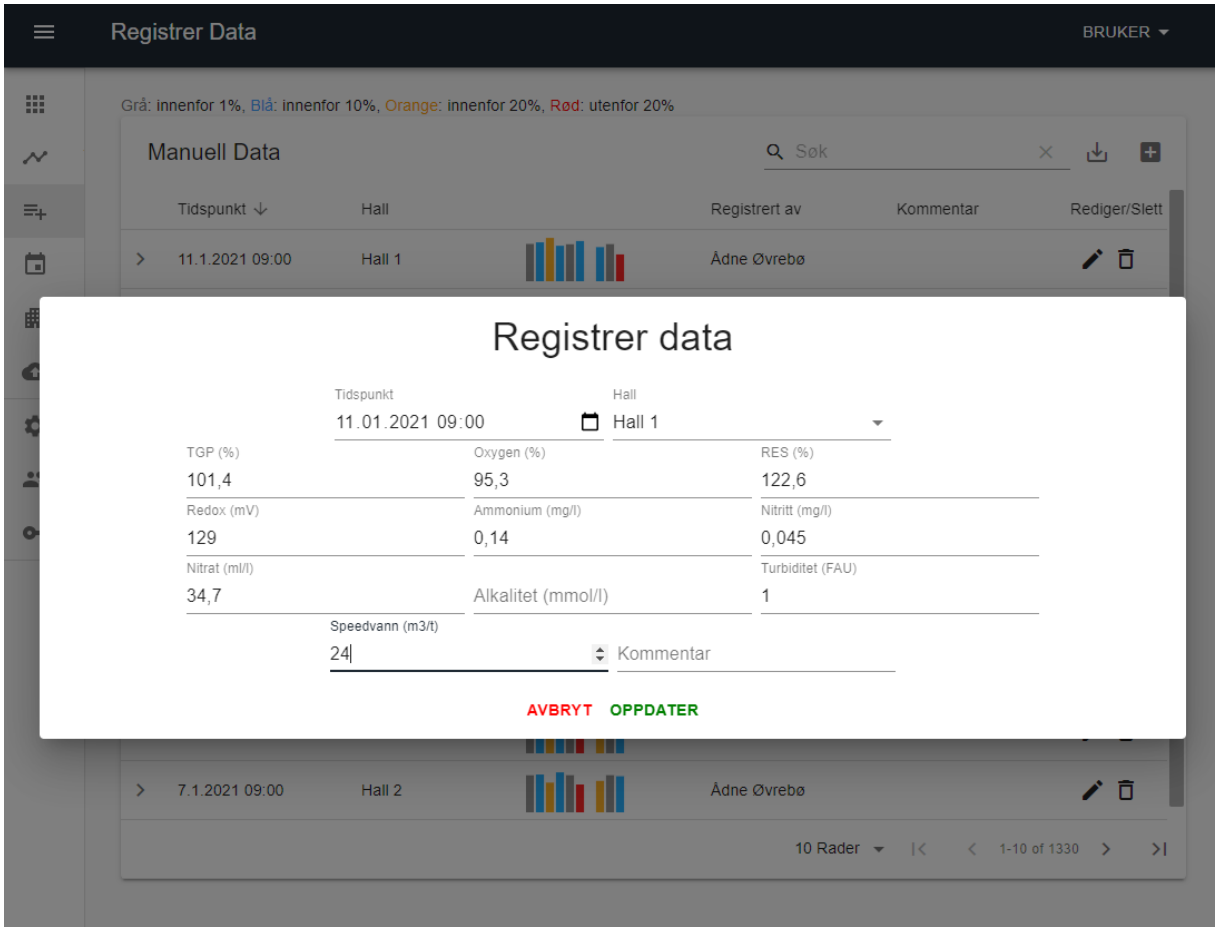
Kode hentet fra pages/register-data/index.tsx.

For at brukeren skal kunne se de faktiske verdiene til en måling, er det lagt til en ekspanderingsknapp som gir målingen mer informasjon. Dette gjør det enkelt for brukeren å navigere i listen, samt raskt få opp presis data om målingen. Eksempel på dette er vist under.



Figur 41: Eksempel på en måling som er ekspandert i listevisningen. Merk: dette er ikke reell data.

Ettersom det er mange verdier som kan legges til her, er det ikke valgt å bruke den innebygde metoden til Material Table komponenten for å legge til data. For å legge til/endre data er det her heller valgt å bruke en modal for å gjøre det mer oversiktlig. Denne modalen bygges på *Dialog* komponentene til Material UI. Under vises hvordan modalen ser ut for brukeren, samt hvordan returverdien til modalen er implementert. Koden gjennomgås ikke i detalj ettersom komponentnavnene er beskrivende, og *updateOrCreate* funksjonen bare sender en enkel forespørsel til APIet med tilstandsvariabelen *manualData*.



Figur 42: Modal for registrering/oppdatering av data. Merk: dette er ikke reell data.

```
1. <Dialog open={visible} onClose={closeModal} fullScreen={fullScreen} f
   ullWidth maxWidth={'lg'}>
2.   <DialogTitle disableTypography>
3.     <Typography variant="h4" align="center">
4.       Registrer data
5.     </Typography>
6.   </DialogTitle>
7.   <DialogContent>
8.     <Grid container direction="row" justify="center" alignItems="flex
   -start" spacing={1}>
9.       <Grid item className={classes.inputWidth}>
10.          <TextField
11.            fullWidth
12.            label="Tidspunkt"
13.            type="datetime-local"
14.            InputLabelProps={{
15.              shrink: true
16.            }}
17.            onChange={(e) => setManualData({ ...manualData, dateRecor
   ded: e.target.value })}
18.            defaultValue={isEdit ? manualData.dateRecorded : new Date
   ().toISOString().slice(0, 16)}
19.          />
20.        </Grid>
21.      </Grid>
22.             ... // flere TextField og Autocomplete komponenter
23.    </DialogContent>
24.    <DialogActions>
25.      <Grid container justify="center">
26.        <Button disabled={loading} style={{ color: 'red', fontWeight:
    'bold' }} onClick={closeModal}>
27.          Avbryt
28.        </Button>
29.        <Button disabled={loading} style={{ color: 'green', fontWeigh
   t: 'bold' }} autoFocus onClick={updateOrCreate}>
30.          {isEdit ? 'Oppdater' : 'Opprett'}
31.        </Button>
32.      </Grid>
33.    </DialogActions>
34.  </Dialog>
```

Returverdien til RegisterDataModal komponenten. Kode hentet fra src/components/RegisterDataModal.tsx.

10.1.13 Visualisering av målinger

Visualiseringssiden kan ses på som den viktigste delen av applikasjonen, iallfall for brukeren. Dette var et av de største og viktigste kravene vi kom fram til sammen med Tytlandsvik Aqua (Krav 1), og er hovedoppgaven dette systemet skal løse. På denne siden skal brukeren kunne hente målinger fra et innsett og hall, eller eventuelt definere perioden manuelt basert på dato. Brukeren skal her kunne filtrere målingsdataen basert på gitte verdier i en nedtrekksmeny, og det skal være mulig å sette innsett oppe på hverandre slik at de kan sammenlignes. Dataen skal bli presentert tydelig og klart, og det er her valgt å bruke graf-biblioteket Apexchart.

Figur 43: Skjermbilde av visualiseringssiden til applikasjonen

Figuren ovenfor viser hvordan denne siden ser ut. For å gi brukeren en enkel måte å velge samt holde oversikt over verdiene det filtreres på, er det valgt å bruke *Autocomplete* komponenten fra Material-UI. Vi definerer i denne komponenten hvilke verdier som kan velges (alle parameterne det kan filtreres på), grupperer dem basert på type (hvilket system parameteren er fra) samt lagrer listen i en lokal tilstandsvariabel. På neste side kan man se koden for denne komponenten, som også blir brukt på Dashbordet.

```
1. <Autocomplete
2.   noOptionsText="Ingen verdier"
3.   // kan velge flere verdier i listen
4.   multiple
5.   // alle parameterene
6.   options={dataKeys}
7.   // gruppering
8.   groupBy={(option) => {
9.     if (option.source === Source.Manual) {
10.        return "Manuell";
11.      } else if (option.source === Source.RAS) {
12.        return "RAS";
13.      } else if (option.source === Source.Feeding) {
14.        return "Fôring";
15.      }
16.    }}
17.    // setter lokal tilstand
18.    onChange={(_, newValue: DataKey[]) => setSelectedKeys(newValue)}
19.    // hva som vises I nedtrekkslisten
20.    getOptionLabel={(option: DataKey) => option.name}
21.    filterSelectedOptions
22.    renderInput={(params) => (
23.      <TextField {...params} variant="outlined" placeholder="Verdier"
   />
24.    )}
25.  />;
```

Hvordan Autocomplete-komponenten fra Material-UI blir brukt. Kode hentet fra pages/visualizaion/indet.tsx.

Tilstandsvariabelen *datakeys* i kodeeksempelet over inneholder alle parameterne brukeren skal kunne filtrere på. I tidlig fase av prosjektet var disse parameterne hardkodet, ettersom det da ikke var så mange parametere. Etter hvert, når vi fikk koblet oss til SCADA-systemet, ble det heller valgt å lagre dette i en ekstern tabell i databasen ettersom det da kom inn mange flere parametere, og vi ville gi brukeren mulighet til å endre på disse selv. Under kan man se datastrukturen til en datakey.

```
1. export type DataKey = {
2.   id: number;
3.   name: string;
4.   key: string;
5.   color: string;
6.   unit: string;
7.   source: Source;
8.   idealValue?: number;
9.   comment: string;
10.    deleted: boolean;
11.    // type blir ikke lagret i database, er bare et valg i klient
12.    // mtp graf type
13.    type?: string;
14.    poolNrInHall?: number,
15.  };
```

DataKey struktur. Kode hentet fra src/assets/dataKeys.ts.

Det ble skrevet et eget script for å lagre alle parameterne fra SCADA systemet inn i datakey tabellen, ettersom dette ville vært tidkrevende å gjøre manuelt samt tidkrevende å opprettholde. Dette ses ikke på som en viktig del av oppgaven, men bare et verktøy som gjorde prosessen enklere for oss, og koden vil derfor ikke gjennomgås her. Koden er lagt til i kildekoden, og har noe lignende logikk som er vist i 10.5 Databehandling. Det er også lagt til en egen side i klienten der brukeren kan endre på attributtene til datakeysene, ettersom det da er enklere for brukeren å endre på navn og farge til parameterne det filtreres på uten at det går gjennom utviklerne. Denne siden vil heller ikke bli gått nærmere inn på, ettersom det kun er et verktøy for brukeren og ikke var strengt tatt nødvendig for å løse hovedoppgaven.

For at dataen som hentes fra APIet skal kunne presenteres i *Apexchart*, må det noe manipulering til. Dataen skal presenteres i en liste med *series* (se 7.8.5.2 Apexcharts & react-apexcharts) samt *options* attributtene skal manipuleres basert på hvilke *datakeys* som er valgt av brukeren. Hver *serie* inneholder graftype og navn på serien basert på *datakey*. Kort sagt kjøres det en for løkke over alle datakeysene som er valgt av brukeren, og basert på disse datakeysene så blir rett data fra APIet lagt inn i riktig serie samt *options* objektet blir manipulert. Det vil altså være en *serie* per *datakey*. En detaljert gjennomgang av denne koden kan bli noe forvirrende, men under er en liten del av koden der selve dataen blir lagt til i riktig *serie*.

```
1. if (datakey.source === Source.Manual) {
2.   // henter rett data/attributt basert på key
3.   const fieldData = d[datakey.key];
4.   // lagrer dataen i rett serie
5.   newSeries[yAxisIndex].data.push({ x: date, y: fieldData });
6.
7.   // manipulerer minimums- og maksverdier options objektet
8.   // er noe forskjellig basert på om brukeren har valgt å
9.   // samle aksene eller ikke
10.    if (collectAxis) {
11.      indexMap[datakey.unit].forEach((indexes) => {
12.        if (fieldData && fieldData < newOptions.yaxis[indexes].min) {
13.          newOptions.yaxis[indexes].min = fieldData;
14.        }
15.
16.        if (fieldData && fieldData > newOptions.yaxis[indexes].max) {
17.          newOptions.yaxis[indexes].max = fieldData;
18.        }
19.      });
20.    } else {
21.      if (fieldData && fieldData < newOptions.yaxis[yAxisIndex].min){
22.        newOptions.yaxis[yAxisIndex].min = fieldData;
23.      }}}
```

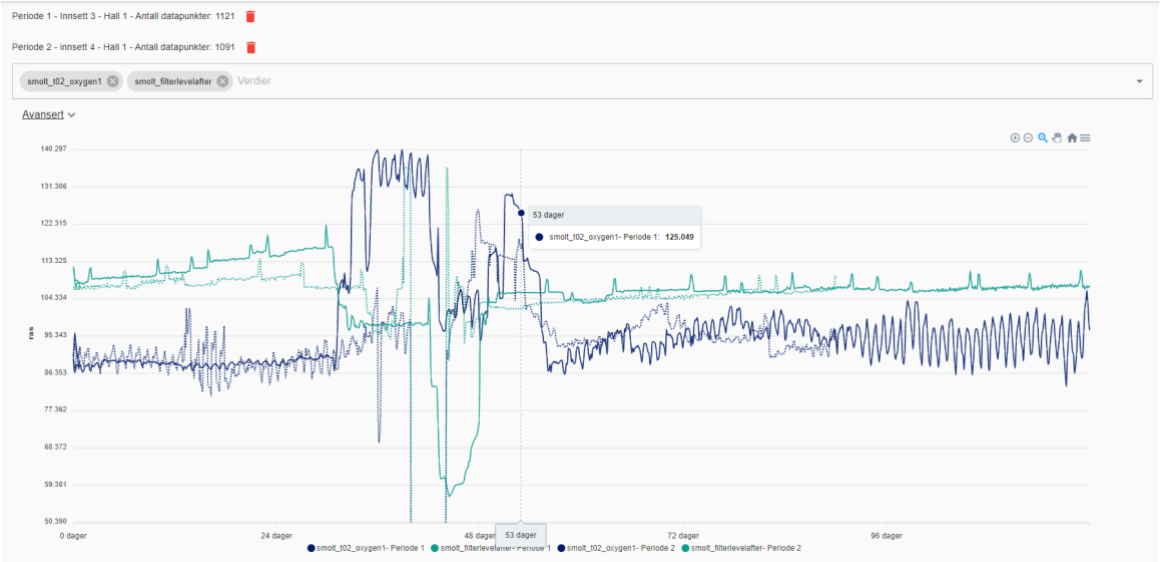
Delkode fra manipuleringen av dataen slik at den kan brukes av «Apexchart». Kode hentet fra src/utils/datamapper.ts.

For å kunne støtte muligheten til å sette flere innsett oppå hverandre, kjøres hele denne logikken flere ganger basert på hvor mange innsett som er valgt. X-aksen vil her da heller representere forskjell på datoene til dataene i forhold til startdato til alle valgte innsett, i stedet for at x-aksen faktisk inneholder datoen til dataen. Det vil si at alle innsettene får samme startpunkt i grafen, selv og de starter på forskjellig tidspunkt. Under vises hvordan datoen til dataen på grafen regnes ut.

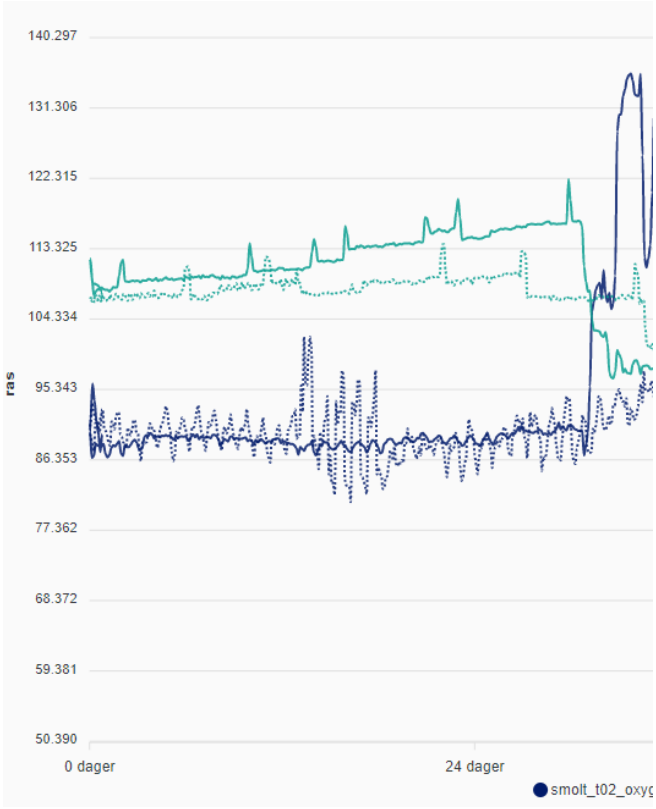
```
1. const date = dataPeriods.length > 1 ? Math.abs(new Date(d.dateRecorde
   d).getTime() - startDate.getTime()) : d.dateRecorded;
```

Hvordan datoen til dataen i grafen regnes ut. Kode hentet fra src/utils/datamapper.ts..

For å kunne skille på innsettene som legges oppe på hverandre på selve graf-presentasjonen er det valgt å sette stiplede linjer på innsett som legges til. Her vil mellomrommet på dem stiplede linjen bli større og større for nye innsett, som baserer seg på indeksen i for-løkken til innsettene. Under kan du se hvordan dette vil se ut i applikasjonen:

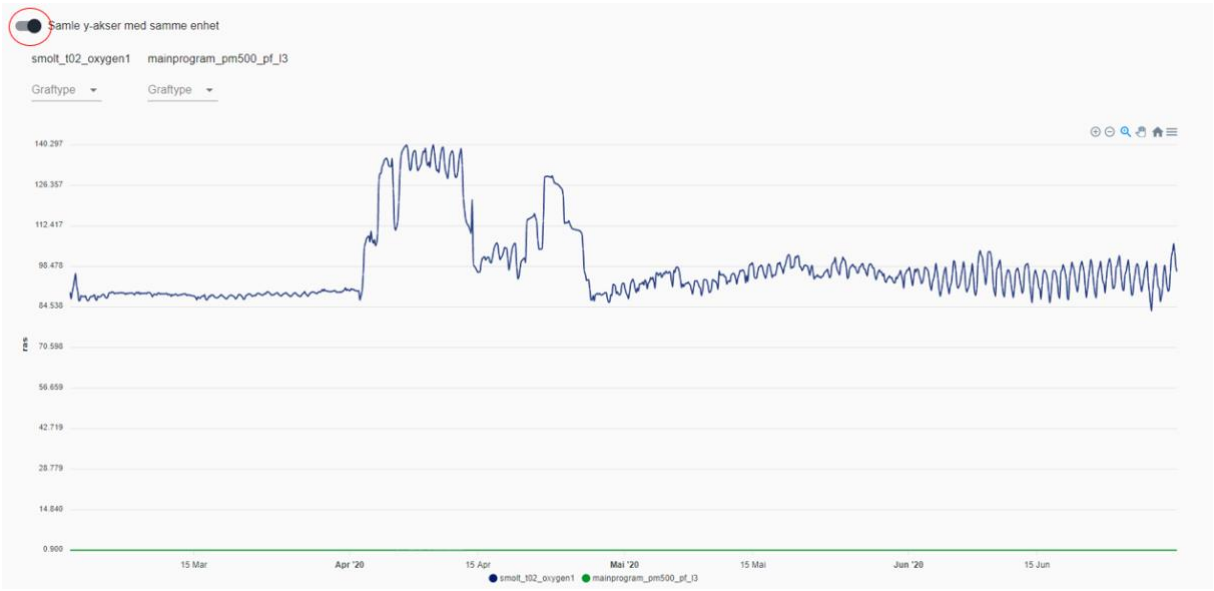


Figur 44: Skjermbilde av analyse der 2 innsett er satt oppå hverandre.

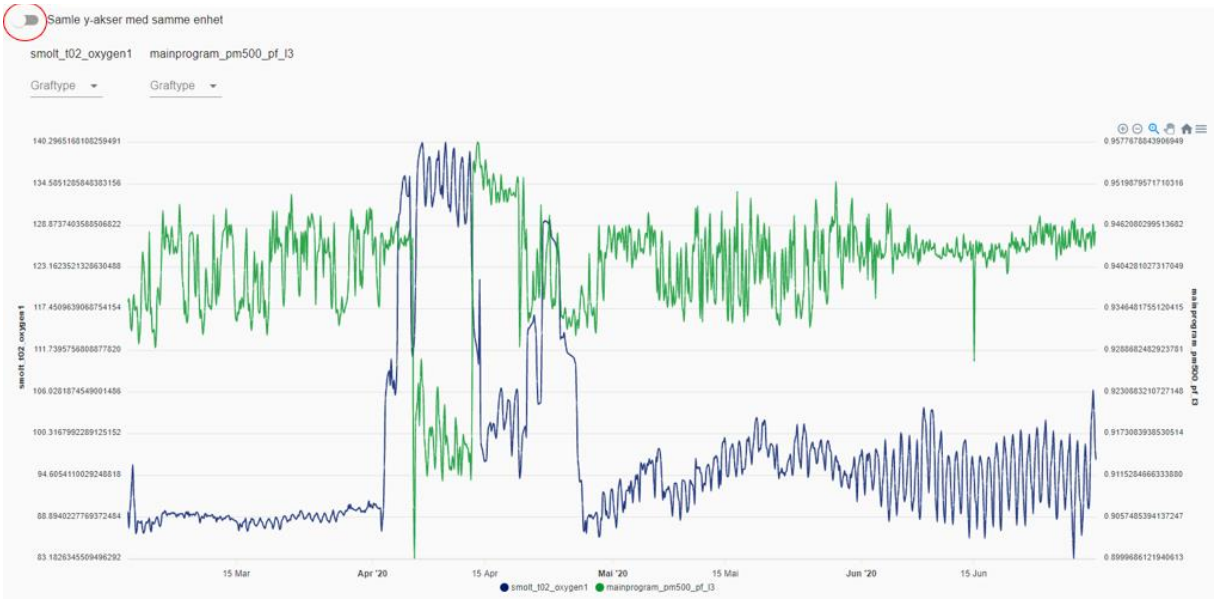


Figur 45: Innzoomet eksempel av skjermbildet over, for å tydeligere se de stiplede linjene. Her er benevningen ras fordi det ikke er lagt inn enhet på nøklene under felter-siden.

Ettersom noen av parameterne ikke har verdier som ligger like nærme hverandre som eksempelet over, selv om de har samme enhet, er det lagt til et valg for brukeren om å splitte y-aksen per serie. Hvis parameterne har forskjellige enheter, er y-aksen uansett splittet. Vi vil i koden legge til et *yaxis* objekt på *options* attributtet per *datakey* som er valgt, men hvis brukeren har valgt å samle aksene vil en *show*-attributt bli satt til *false* på objektet hvis det er et duplikat. Under vises et eksempel med og uten samlede y-akser, og hvorfor dette vil være en stor nødvendighet for brukeren å bruke i enkelte tilfeller.

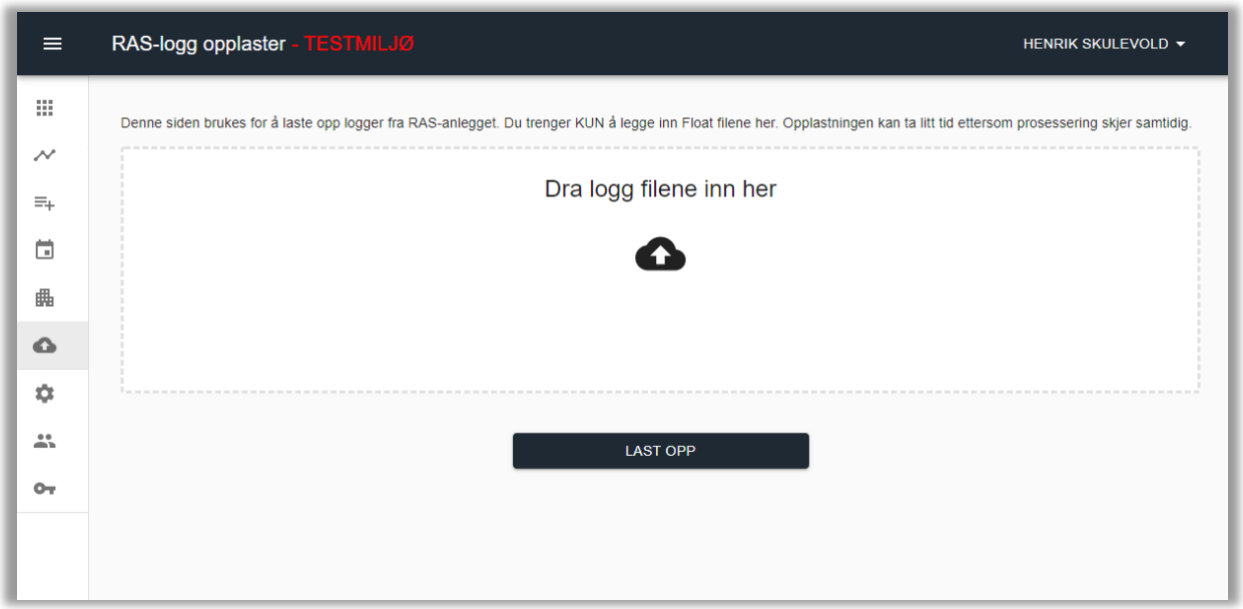


Figur 46: skjermbilde av en grafanalyse med samlede akser. Her er det umulig å tyde den grønne grafen nederst.



Figur 47: Skjermbilde av en grafanalyse uten samlede akser. Her er det enklere å sammenligne verdiene mot hverandre

10.1.14 Automatisk data



Figur 48: Skjermbilde av "Automatisk Opplasting" siden

De opprinnelige fasene ble noe endret på grunn av covid-19, og at vi ikke visste om vi fikk koblet oss til systemene hos Tytlandsvik Aqua (se kapittel 8.1.1 Smidig endring av faser). Derfor ble det laget en side der brukeren selv kan laste opp loggfilene (DAT-filer) fra SCADA-systemet manuelt til APIet. Siden vises ovenfor i figur 48. Denne siden erstattet midlertidig behovet for å koble oss til systemene deres og brukeren kan teste applikasjonen tidlig i prosjektløpet. APIet for denne logikken vil være lik, uansett om loggfilene blir hentet automatisk via tjenesten for innhenting av data eller lastet opp via denne siden. Derfor er det ikke lagt mye tid inn i utviklingen av denne siden ettersom den er tenkt på som en midlertidig løsning. APIet tar for seg det meste av det tunge løftet her, altså prosesseringen av filene som blir beskrevet i 10.5.2 SCADA-systemet.

Det ble valgt å bruke *DropzoneArea* komponenten fra material-ui-dropzone pakken for å gi brukeren en intuitiv måte å legge til filer på samt spare tid på implementasjon. Denne komponenten lagrer filene i en lokal tilstandsvariabel, som blir brukt når vi sender filene til APIet.

```
<DropzoneArea
  maxFileSize={1000000000}
  onChange={(files) => setFiles(files)}
  filesLimit={10000}
  showPreviews={true}
  showPreviewsInDropzone={false}
  useChipsForPreview
  previewGridProps={{ container: { spacing: 1, direction: 'row' } }}
  previewChipProps={{ classes: { root: classes.previewChip } }}
    previewText="Valgte filer"
    dropzoneText="Dra logg filene inn her"
/>
```

Implementasjon av DropzoneArea komponenten. Kode hentet fra pages/automatic-data/index.tsx.

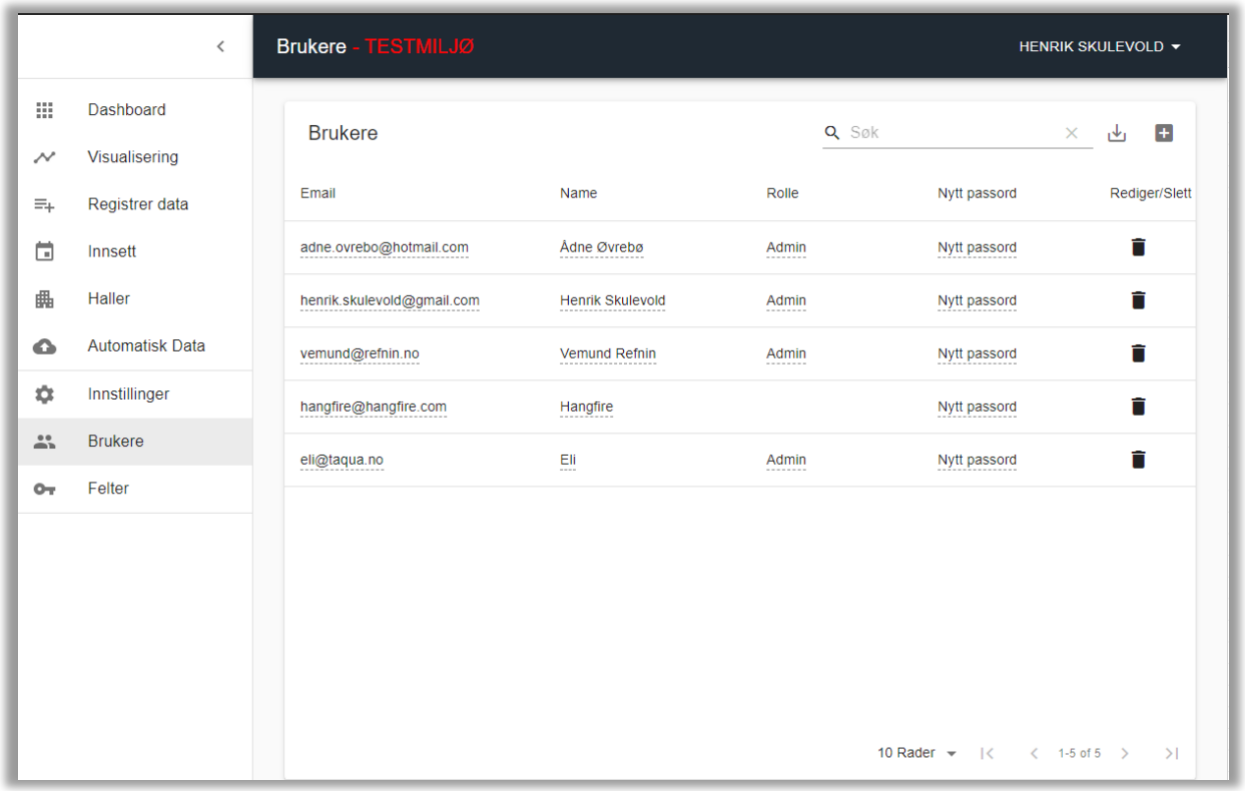
For å kunne sende filene over HTTPS, sendes dem som *FormData* med HTTP-headeren *content-type: multipart/form-data* gjennom en POST forespørsel. Under vises hoveddelen til funksjonen som manipulerer *FormData* objektet og sender filene til APIet.

```
var formData = new FormData();
for (let file of files) {
  formData.append('files', file);
}
let response = await BaseApiService.uploadFile('RasLog/processLogfile
', formData);
```

Utsnitt av funksjonen som manipulerer «FormData» objektet og sender filene til APIet. Kode hentet fra pages/automatic-data/index.tsx.

10.1.15 Brukere

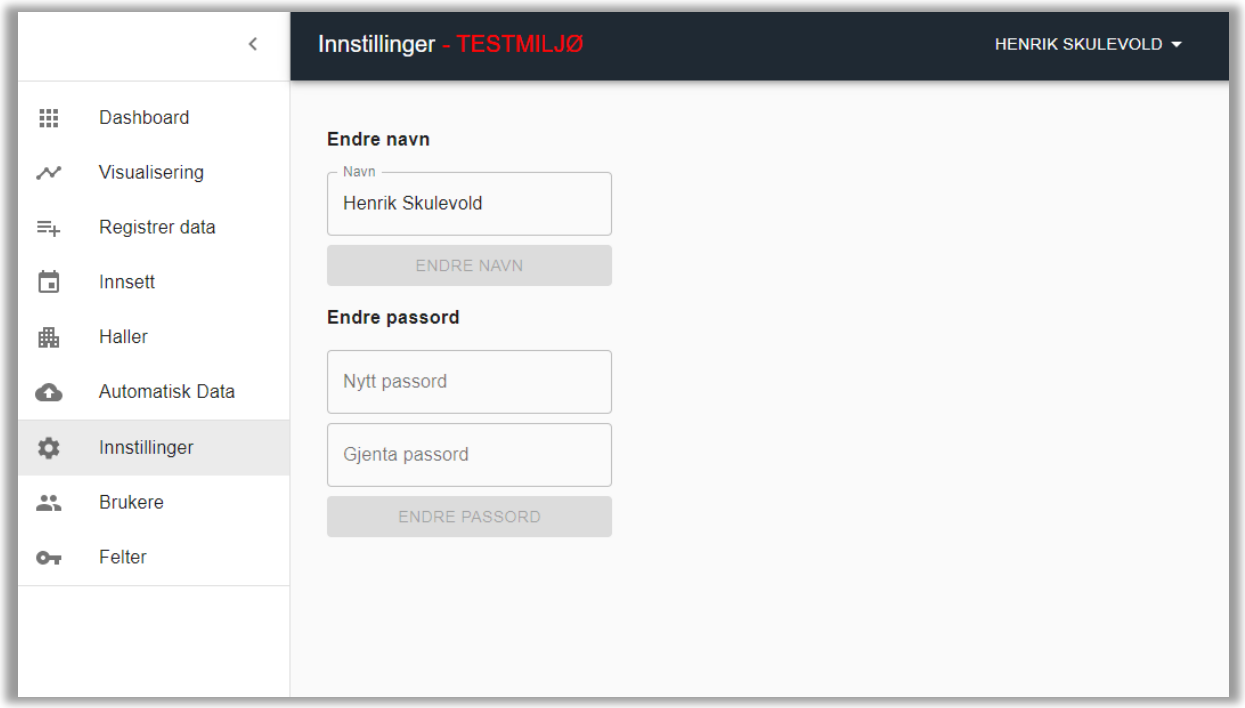
Ettersom det ikke er mange som skal bruke denne applikasjonen, ble det valgt å gjøre brukerhåndteringen enkel. Dette ble gjort ved å ha en egen side over brukerne, som bare administratorer har tilgang til, der det skal kunne legges til brukere, endres på brukerne, og slette brukere. Dette imøtekommer også krav 8. Her er det også valgt å bruke Material Table komponenten for enkel implementasjon og intuitiv bruk.



Figur 49: Skjermbilde av brukeresiden til applikasjonen

10.1.16 Innstillinger

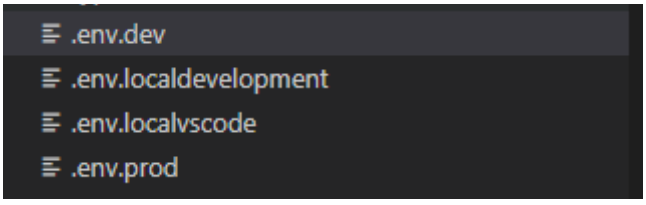
For at brukeren skal kunne endre på egne innstillinger, er det valgt å lage en enkel innstillingsside for dette. Her kan brukeren endre på eget navn og passord. Skjermbildet av siden kan ses under i figuren under.



Figur 50: Skjermbilde av innstillingssiden

10.1.17 Støtte for flere miljøer

Webapplikasjonen skal kunne kjøres på forskjellige miljøer, samt bygge docker-bilder basert på forskjellige miljøer. Grunnen til dette er forklart nærmere i 9.2 Miljøer kapittelet. For å få dette til, har vi valgt å bruke bibliotekene dotenv og env-cmd. Her har vi spesifisert egne *.env* filer for *dev*, *localdevelopment*, *localvscode* og *prod*. Grunnen til at det er to filer for lokalt miljø er siden APIet kan kjøres lokalt på to forskjellige porter. Hvis man kjører APIet i Visual Studio kjøres prosjektet på port 44315. Hvis man kjører APIet med *dotnet run*, kjøres det på port 5000.



Figur 51: .env filene som brukes i Webapplikasjonen.

Disse filene setter en miljøvariabel som blir sjekket i en konfigurasjonsfil som henter applikasjonsinnstillinger fra gitte JSON-filer. Det er en JSON-fil for hvert miljø, og per nå inneholder disse filene bare API-endepunktet som skal brukes. Det er dette som gjør at vi bare trenger å kalle *Configs.serverApi* i *BaseApiService* klassen, som vist under, for å sende forespørselen til rett endepunkt basert på miljø.

```
1. class BaseApiService {
2.     get(url: string, ctx: any = null,): Promise<AxiosResponse<any>> {
3.         return axios.get(Configs.serverAPI + url, {
4.             headers: {
5.                 Authorization: this.headerConfig(ctx),
6.             },
7.         });
8.     }
```

BaseApiService.get() metoden, der Configs.serverAPi variabelen blir brukt. Kode hentet fra src/services /BaseApiService.tsx.

Env-cmd blir brukt for å spesifisere hvilken *.env*-fil som skal brukes når applikasjonen kjøres/bygges. Pakken blir brukt rett i *package.json* filen der vi spesifiserer hvilke scripts som skal kunne kjøres i kommandolinjen. Under ser du hvordan dette er implementert.

```
1. "scripts": {
2.   "dev": "next dev",
3.   "local": "env-cmd -f .env.localdevelopment next",
4.   "local:vscode": "env-cmd -f .env.localvscode next",
5.   "prod": "env-cmd -f .env.prod next",
6.   "build": "next build",
7.   "build:dev": "env-cmd -f .env.dev next build",
8.   "start": "next start"
9. },
```

Alle scriptsene som kan kjøres mot kientapplikasjonen. Kode hentet fra «package.json».

Dette gjør eksempelvis at man i kommandolinjen kan kjøre *npm run local* for å kjøre applikasjonen mot lokalt API, eller *npm build:dev* for å bygge applikasjonen mot test API.

10.1.18 Datatyper

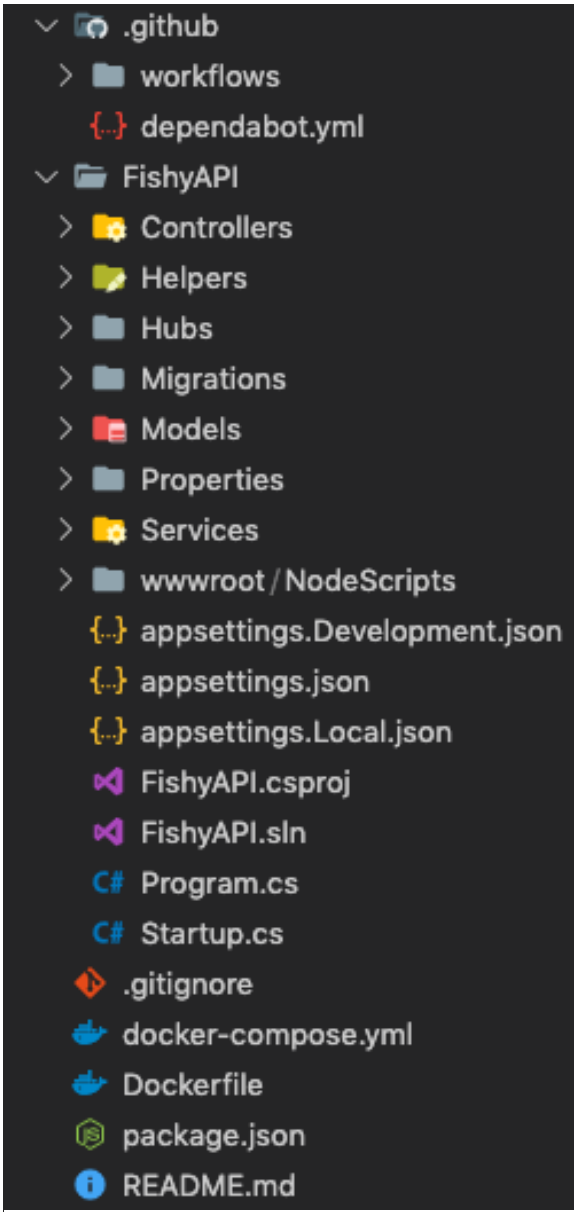
Ettersom vi bruker Typescript, har vi fordelen av at alle variabler, parametere og funksjoner støtter bruk av typer. Det er valgt å legge inn alle databasetypene inn i klientkoden ved bruk av en utvidelse i Vscode, kalt *C# to TypeScript*, som enkelt konverterer C# modeller til Typescript typer. Denne utvidelsen gjør at vi enkelt kan oppdatere typene begge plasser (API og frontend) når det er behov for det, og i klientkoden vil vi få opp feilmeldinger der koden videre må oppdateres. Ved denne prosessen vil man i teorien alltid kunne fange opp alle plasser i klientkoden der koden feiler etter en oppdatering av en modell i databasen, noe som sikrer en mer effektiv utvikling og tryggere kode.

10.2 API

Man kan argumentere for at APIet er den viktigste delen av løsningen ettersom den skal håndtere ny, eksisterende og endringer på data. Det er tross alt data og eksponeringen av denne som legger grunnlaget for systemet som skal utvikles.

10.2.1 Filstruktur

Prosjektet ble opprettet gjennom en standard .NET Core API mal i Visual Studio. Visual studio er et utvikler-verktøy som er utviklet av Microsoft og fokuserer på å forenkle utviklingen i særlig .NET. Filstrukturen for APIet er det som ligger under mappen FishyAPI. Filene som ligger utenfor denne mappen, er hjelpefiler som vi skal undersøkes senere i oppgaven. En oversiktlig og meningsfull filstruktur gjør det enklere å navigere i prosjektet.



Figur 52: Filstruktur API

Mappen *Controllers* inneholder alle kontrollerne for API-endepunktene, dette er kode som styrer logikken til de ulike endepunktene. Mappen *Helpers* implementerer hendig hjelpefunksjonalitet som brukes flere steder. *Hubs* inneholder endepunkter for en såkalt signalR hub, logikken tilknyttet WebSockets. *Migrations* inneholder migrasjonene tilknyttet database-konteksten, dette er altså hvordan endringer i databasen håndteres. *Models* inneholder definisjonen på de ulike modellene som blir anvendt, dette er strukturen på ulike objekter. *Services* holder på tjenester som blir injisert ved hjelp av avhengighetsinjeksjon. Mappen *wwwroot* inneholder filer som skal være tilgjengelige i den aktuelle mappestrukturen i kjøretid. Filene som starter med *appsettings*, inneholder innstillinger og variabler for programmet, her er det opprettet flere for de ulike miljøene. *Program.cs* kjører selve programmet mens *Startup.cs* er koden som initialiserer programmet og ulike innstillinger.

10.2.2 Modeller

ER-Diagrammet må oversettes til C# modeller slik at Entity Framework kan tolke dem og opprette tabellene og forholdene mellom dem som er nødvendig. Modellene i APIet definerer derfor hvordan dataen er strukturert. Modellene vil også bli anvendt til å bygge database-konteksten til APIet. For å forklare hvordan modellene blir opprettet ligger koden som danner strukturen for *User*-objektet under.

```
1. namespace FishyAPI.Models.DBModels
2. {
3.     public class User
4.     {
5.         public int Id { get; set; }
6.
7.         [Required(ErrorMessage = «Name required»)]
8.         public string Name { get; set; }
9.
10.         [Required(ErrorMessage = «Email required»)]
11.         [EmailAddress(ErrorMessage = "Invalid Email Address")]
12.         public string Email { get; set; }
13.
14.         [Required(ErrorMessage = «Password required»)]
15.         [JsonIgnore]
16.          public string Password { get; set; }
17.          public Roles Role { get; set; }
18.          public bool IsDeleted { get; set; }
19.     }
20. }
```

User-objektet definer i C#. Hentet fra /Models/User.cs

Som dette kodeutdraget viser definerer man flere attributter på objektet, for eksempel ID og navn. Typen blir også definert. ID er satt til å være et heltall (eng.: integer) mens navn er satt til å være en streng. Ettersom disse modellene skal brukes i APIet kan man definere ulike dekoratører, som er kode som tilegner attributtene ekstra funksjonalitet. Et eksempel på dette er dekoratøren *Required* som tilsier at dette feltet er påbudt. Dersom dette feltet mangler når man skal opprette en ny instans kan man spesifisere en feilmelding som skal vises. Fra kodeutsnittet over er også *EmailAddress* og *JsonIgnore* slike dekoratører. All dataen som blir lagret er definert i slike modeller, samtlige modeller kan man finne i *Models* mappen i API-prosjektet. Denne strategien for implementasjon av modeller er fulgt for de resterende modellene som er vist under 9.5 Database ER-Diagram.

Relasjoner

Å opprette nye tabeller med felter uten relasjoner er enkelt, men det blir noe mer komplekst når det også trengs relasjoner mellom modellene. Dersom det for eksempel skal opprettes en mange til én relasjon, kan det spesifiseres med en *ICollection*-type på objektet som skal ha mange referanser til et annet objekt. Dersom man kun skal referere til ett objekt trenger man kun å spesifisere at et felt er av typen man ønsker å referere til. Man kan også gjøre objekter nullbare ved å bruke klassen *Nullable<type>*, dette vil si at attributtene kan ha verdien null.

Under følger et eksempel for implementasjon av *PoolPeriod* som implementerer strategiene som er diskutert over, noe som også vil være likt for de resterende modellene.

```
1. public class PoolPeriod
2. {
3.     public int Id { get; set; }
4.     public Pool Pool { get; set; }
5.     public Period Period { get; set; }
6.     public DateTime StartDate { get; set; }
7.
8.     // Self join parent child relation
9.     public PoolPeriod Parent { get; set; }
10.      public ICollection<PoolPeriod> Children { get; set; }
11.      public Nullable<DateTime> EndDate { get; set; }
12.  }
```

PoolPeriod-objektet definert i C#. Henter fra /Models/PoolPeriod.cs

Database-kontekst i Entity Framework

Som nevnt tidligere er det valgt å benytte ORM-rammeverket Entity Framework. For at Entity Framework skal vite hvilke tabeller som skal opprettes i databasen og relasjonene mellom dem må dette spesifisere i database-kontekst. Dette gjøres ved å opprette noe som i Entity Framework kalles for *DbSets*, i realiteten spesifiserer dette hvilke tabeller som skal opprettes og hva navnet på denne tabellen skal være.

Et eksempel på spesifiseringen av en slik tabell er vist under. Dette må spesifiseres for alle de modellene man ønsker å lagre. Det komplette objektet kan undersøkes nærmere i filen *DatabaseContext.cs* under mappen Models.

```
1. public DbSet<User> Users { get; set; }
```

DbSet eksempel med User-objektet. Henter fra /Models/DatabaseContext.cs

I oppstartfilen må database-konteksten spesifiseres på måten som er vist i neste kodeutsnitt. Denne koden må legges under *ConfigureServices* metoden i filen *Startup.cs*. Ved å initialisere database-konteksten som en tjeneste kan denne injiseres hvor som helst, hvordan man anvender en slik injeksjon blir forklart i nærmere detalj under delkapittelet 10.2.6 Kontrollere. I kodeutsnittet under ser man også at tilkoblingsstrengen og andre database innstillinger settes. Tilkoblingsstrengen spesifiserer det som trengs for å koble seg til databasen, denne er spesifisert i *appsettingsfilene*.

```
1. services.AddDbContext<DatabaseContext>(options =>
2. {
3. options.UseSqlServer(Configuration.GetConnectionString("DefaultConnec
   tion"),
4.    sqlServerOptionsAction: sqlOptions =>
5.      {
6.          sqlOptions.EnableRetryOnFailure(
7.          maxRetryCount: 5,
8.          maxRetryDelay: TimeSpan.FromSeconds(30),
9.       });
10.  });
```

Legge til Databasekontekst. Hentet fra: /Startup.cs

Etter databasekonteksten er implementert kan denne brukes på måten som er vist i kodeutsnittet under. Her benyttes database-konteksten til å hente den første brukeren som har en epost som er lik en annen epost.

```
1. _context.Users.SingleOrDefault(x => x.Email == model.Email);
```

Eksempel på å hente ut en entitet fra databasen der eposten er lik en gitt epost.

Migreringer

For å kunne håndtere endringer i modellene er det nødvendig å kunne gjøre migreringer på databasen. Dette støtter heldigvis Entity Framework ut av boksen og man kan kjøre *dotnet* kommandoer i CLI for å opprette migreringer. Ved å benytte migrasjoner sikrer man at dataen blir tatt vare på i riktig format.

Dersom man har gjort endringer på databasekonteksten opprettes en ny migrasjon på følgende måte:

```
dotnet ef migrations add <navn på migrasjon>
```

Når en slik migrasjon opprettes, vil det bli opprettet en fil i *Migrations* mappen. En slik migrasjonsfil spesifiserer endringene fra forrige kontekst til den nye, ved bruk av en *Up*-funksjon og en *Down*-funksjon. *Up* spesifiserer hva som skal opprettes, for eksempel dersom man legger til nye attributter på en modell blir disse spesifisert her. *Down* spesifiserer hva som skal destrueres. *Up* kjøres alltid før Down slik at egendefinerte migrasjoner må spesifiseres i *Up* før *Down* eventuelt fjerner data. Man kan oppdatere databasen direkte ved å bruke følgende kommando:

```
dotnet ef database update <navn på migrasjon>
```

Denne kommandoen oppdaterer databasen til den spesifiserte migrasjonen, man kan også bruke denne for å fjerne migrasjoner fra databasen, altså rulle tilbake til en gammel migrasjon.

For å slippe å gjøre denne oppdateringen manuelt er det valgt å legge inn en kommando i oppstartfilen som automatisk gjør denne migrasjonen ved oppstart. *Migrate* metoden nedenfor sjekker om det ligger migrasjoner i kildekoden som ikke er i databasen, dersom dette er tilfellet kjøres det en oppdatering av databasen under oppstarten av API-et.

```
context.Database.Migrate();
```

Kode for å kjøre migrering av database. Hentet fra /Startup.cs

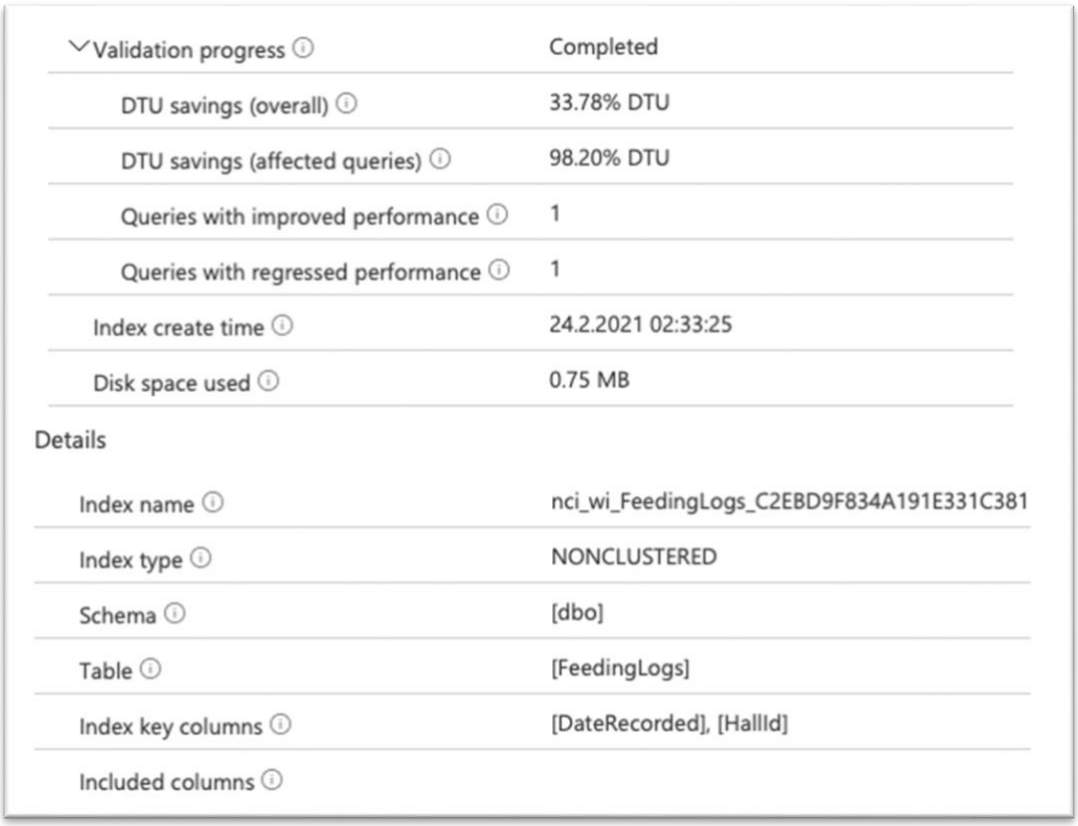
Databaseoptimalisering

Ettersom det er AzureSQL som er benyttet skjer optimaliseringen av databasen automatisk. Denne optimaliseringen inkluderer oppretting av nye indekser, dropping av indekser og det som heter Force Plan. Force Plan velger automatisk bedre forespørsler dersom man skriver en treg SQL forespørsel (microsoft, 2021). AzureSQL har analysert flere millioner databaser for å automatisk kunne detektere optimaliseringer. Etter erfaring fungerer denne svært bra. Man kan tillate automatisk optimalisering på AzureSQL ressursen i Azure portalen under siden Automatic Tuning.

Eksempelvis: Etter opprettelse av tabellen *FeedingLogs* og innsetting av data blir det raskt foreslått en ny indeks på tabellen som går på *DateRecorded* og *HallId*, dette er felter som blir anvendt til filtrering av denne dataen. Figuren under viser eksempel på opprettelse av en slik indeks, dette blir presentert under AzureSQL ressursen i Azure Portal.

Figur 53: Skjermbilde av automatisk optimaliseringsforslag i Azure

Forslagene blir automatisk implementert og testet, man kan videre undersøke hvor stor effekt en slik optimalisering har. Dersom man går inn på detaljene rundt optimaliseringen som er gjort kan man se hvor stor effekt denne har. I figuren under kan man se at man sparer svært mange prosent DTU (Data Tier Unit). DTU er en gitt mengde I/O (Input/Output) og prosesseringskraft, dette er standard enhet i Azure for å kjøpe og bruke database I/O og prosesseringskraft (Microsoft, 2021). Det er også enkelt å reversere en slik automatisk optimalisering dersom man ser at den ikke fungerer. Ved å benytte seg av denne funksjonaliteten slipper man å tenke på optimalisering når man oppretter nye tabeller etc., noe som er svært tidsbesparende.



Figur 54: Effekt av optimalisering

10.2.3 JWT

Som nevnt i teorikapittelet 7.10.8 Sikkerhetsteknologier brukes tredjepartsbiblioteket *system.IdentityModel.Tokens.Jwt* for handtering av JWTer. Det er implementert ved å bruke en JWT mellomvare (eng.: middleware). Mellomvare er noe som blir kjørt hver gang det blir gjort kall og gitt en respons. Denne mellomvaren blir initialisert i APIet sin *startup.cs* fil med denne kodelinjen:

```
1. // custom jwt auth middleware
2. app.UseMiddleware<JwtMiddleware>();
```

Legger til JWT mellomvar. Hentet fra /Startup.cs

Mellomvaren sjekker om det er en token i *Athorization* attributt i HTTP hodet. Hvis det er en token der vil mellomvaren gjøre følgene steg:

1. Validere tokenen.
2. Hente bruker-IDen fra tokenen.
3. Legger bruker-IDen til konteksten.

På denne måten vet man at alle bruker-IDene i konteksten er autoriserte.

Svakheter

Ifølge boka The JWT Handbook (Peyrott, Version 0.14.1, 2016-2018) kan man enkelt unngå autoriseringen dersom man fjerner signaturen og serveren ikke håndterer dette på riktig måte. Dette gjør man ved å sette *alg: none* i hodet og fjerner signaturen, altså tredje delen av tokenen. Her er en signert token kryptert med *alg: HS256* (tre biter separert med punktum).

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjciLCJuYmYiOjE2MTA0Mzc5
NjgsImV4cCI6MTYxMTA0Mjc2OCwiaWF0IjoxNjEwNDM3OTY4fQ.xIySm_c1McHvj8Z6xZ
4lWRwAMaRr0Q6cy6lnS30JUrE
```

Dersom man fjerner signeringen og gjør tokenen ukryptert ved å sette *alg: none* i hodet vil tokenen se slik ut (kun to biter, separert med punktum):

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.ewogICJpZCI6ICI3IiwKICAibmJmIjogM
TYxMDQzNzk2OCwKICAiZXhwIjogMTYxMTA0Mjc2OCwKICAiaWF0IjogMTYxMDQzNzk2OA
p9
```

I vår løsning blir tokenen sjekket i */Helpers/JwtMiddleware.cs* i APIet. Den usignerte tokenen vi føre til at *validatedToken* ikke blir satt og man får ikke autorisert tilgang.

```
1. tokenHandler.ValidateToken(token, new TokenValidationParameters
2. {
3.       ValidateIssuerSigningKey = true,
4.       IssuerSigningKey = new SymmetricSecurityKey(key),
5.       ValidateIssuer = false,
6.       ValidateAudience = false,
7.       ClockSkew = TimeSpan.Zero
8. }, out SecurityToken validatedToken);
```

Validere JWT token. Hentet fra /Helpers/JwtMiddelware.cs

10.2.4 Roller

I databasen har en bruker et nummerflagg som representer rollen. Rollene ble strukturert i et enum med økende autoritet. Høyere tall gir mer autoritet. Dette enumet ser slikt ut:

```
1. public enum Roles
2. {
3.       Reader = 0,
4.       Writer = 1,
5.       Admin = 2,
6.       Scheduler = 3,
7. }
```

Enum for roller. Hentet fra Models/Roles.cs

Scheduler er et unntak. Til tross for at denne har det høyeste tallet har ikke den høyest autoritet. Dette er en spesialrolle som blir beskrevet i kapittelet 9.4.1 Roller.

I APIet ble det konstruert en dekoratør for å sjekke om en brukere er autorisert med en spesifisert rolle. Denne dekoratøren har som hensikt å fange når brukeren ikke er logget inn eller har feil rolle. Disse betingelsene blir fanget ved å returnere et uautorisert resultat.

```
1. public void OnAuthorization(AuthorizationFilterContext context)
2.     {
3.         var user = (User)context.HttpContext.Items["User"];
4.         if (user == null)
5.         {
6.             // Ikke logget inn
7.             context.Result = new JsonResult(new { message =
   "Unauthorized" }) { StatusCode = StatusCodes.Status401Unauthorized };
8.         }
9.         else if (user.Role == Roles.Scheduler && _minRole ==
   Roles.Scheduler)
10.          {
11.                 // Spesialrollen Scheduler har tilgang og returnerer
                    derfor ingenting
12.          }
13.          else if (user.Role < _minRole || user.Role ==
   Roles.Scheduler)
14.          {
15.              // Ikke riktig rolle
16.              context.Result = new JsonResult(new { message =
   "Unauthorized, needs elevated privilege." }) { StatusCode =
   StatusCodes.Status401Unauthorized };
17.          }
18.
19.      }
```

Autoriseringsattributt. Hentet fra Helpers/AuthorizeAttribute.cs

Hvis *context.Result* blir satt i en av sjekkene, så vil ikke metoden dekoratøren er satt på bli kjørt. Et eksempel på bruk av attributtet er som følger, der den innloggede brukeren minimum må ha rollen *Writer* for at funksjonen *Put* skal kjøres:

```
1. [HttpPut("{id}")]
2. [Authorize(Roles.Writer)]
3. public ActionResult<Hall> Put(int id, [FromBody] HallRequest model)
4. {
5. ...
6. }
```

Eksempel på bruk av Autoriseringsattributtet. Hentet fra Helpers/AuthorizeAttribute.cs

10.2.5 Tjenester

For å enkelt kunne benytte samme kode fra samme instans flere steder er det valgt å bruke det som i .Net kalles for tjenester (eng.: services). Dette gjøres ved å definere en klasse som implementerer et grensesnitt (eng.: interface). Videre legger man til denne tjenesten i *Startup.cs* filen, som vist under. Eksempelet under legger til en *UserService* som implementerer funksjonalitet tilknyttet brukerhåndteringen. *IUserService* definerer grensesnitt for tjenesten, vanligvis i .NET prefikser man denne med I, mens *UserService* implementerer selve funksjonaliteten. Man kan her for eksempel bruke *AddScoped* som oppretter en ny instans per *HttpContext* eller *AddSingleton* som oppretter en instans for hele programmet.

```
services.AddScoped<IUserService, UserService>();
```

Eksempel på å legge til en scoped tjeneste.

Tjenesten kan videre injiseres i en kontroller på følgende måte:

```
1. private IUserService _userService;
2. public AuthController(IUserService userService)
3. {
4.     _userService = userService;
5. }
```

Eksempel på injisering av tjeneste i en kontroller.

Den injiserte tjenesten kan nå brukes som en initialisert instans. Det er nå ikke behov for å initialisere en ny tjeneste hver gang man oppretter et objekt, man benytter den samme.

Injisering er benyttet for følgende tjenester:

- Brukertjeneste: Utføre handlinger på en bruker baser på http konteksten.
- Logging: Tjeneste for å logge meldinger fra applikasjonen.
- Databasekontekst: Utføre handlinger på databasen i kontekst av Entity Framework.
- I minne cache tjeneste: Tjeneste for håndtering av cache.
- Node tjeneste: Kjøre node.js funksjoner i .NET

10.2.6 Kontrollere

Kontrollere styrer logikken til endepunktene og returnerer det riktige resultatet, det er kontrollerne som er beskrevet i kapitelet tilknyttet arkitektur som her blir implementert. En kontroller kan håndterer flere endepunkt. Navnet på en kontroller brukes i navnet på endepunktet ettersom det er valgt å dekorere kontrolleren på måten som vist påfølgende. Her spesifiseres også at denne klassen er en API-kontroller som arver fra *ControllerBase* klassen, som er basen en API-kontroller bygger på. Eksempelet er hentet fra *AuthController* kontrolleren.

```
1. [ApiController]
2. [Route("[controller]")]
3. public class AuthController : ControllerBase
```

Eksempel på initialisering av en kontroller. Hentet fra: /Controllers/AuthController.cs

Videre i konstruktøren blir tjenestene som er satt opp i *Startup.cs* injisert til kontrolleren på følgende måte:

```
1. private IUserService _userService;
2. private readonly DatabaseContext _context;
3. private readonly IMemoryCache _cache;
4. private readonly ILogger _logger;
5.
6. public AuthController(IUserService userService, DatabaseContext
   context, IMemoryCache cache, ILogger<AuthController> logger)
7. {
8.   _userService = userService;
9.   _context = context;
10.  _cache = cache;
11.  _logger = logger;
12.  }
```

Injisering av tjeneste i til en kontroller. Hentet fra: /Controllers/AuthController.

Her injiseres for eksempel database-konteksten slik at den kan brukes. Variablene settes til *readonly* ettersom man ikke ønsker å endre på disse attributtene etter at kontrolleren er initiert. Det er valgt å sette _ som prefiks på tjenestene som injiseres slik at man enkelt kan vite at det er en injisert tjeneste som blir benyttet.

Videre implementerer kontrolleren funksjoner som representerer endepunktene.

```
1. [HttpPost("signUp")]
2. [Authorize(Roles.Admin)]
3. public IActionResult SignUp(SignUpRequest model)
4. {
5.     var user = _userService.CreateUser(model);
6.     if (user != null)
7.     {
8.         _cache.Remove("get-all-users-key");
9.         return Ok(user);
10.     }
11.     else
12.     {
13.         return StatusCode(409, "User already exists");
14.     }
15. }
```

Eksempel på en metode som representerer et endepunkt. Hentet fra: /Controllers/AuthController.cs

Eksempelet over viser implementasjonen av endepunktet */Auth/signUp*. Man spesifiserer at dette er et POST endepunkt ved å benytte *HttpPost* dekoratøren, det finnes også slike dekoratører for GET, PUT, PATCH og DELETE. *Authorize* dekoratøren ovenfor spesifiserer at man må være autorisert med rollen administrator for å kunne bruke dette endepunktet. Videre opprettes det en bruker, dersom denne ble opprettet riktig fjernes cache med alle brukere og returnerer statuskode OK (200) med den nyopprettede brukeren. Dersom man får returnert null finnes brukeren allerede og det returneres en konflikt (409) som sier at brukeren allerede eksisterer. Dataen som sendes inn hentes fra kroppen til forespørselen ved å spesifisere at modellen er av typen *SignUpRequest* som er definert som en modell i mappen *Models*. Det er denne fremgangsmåten som er benyttet på alle endepunktene som er implementert, der det kun er logikken i funksjonskroppen som er endret. Det er noen unntak som skal undersøkes nærmere.

Spørring etter store mengder data

For å hente ut data fra databasen kan man bruke Entity Framework sin LINQ. Dette følger med Entity framework og lar deg gjøre spørringer direkte i C#. LINQ blir brukt flere steder i APIet, for eksempel for å hente ut alle brukerne.

```
1. public IEnumerable<User> GetAll()
2. {
3.     // LINQ:
4.     return _context.Users.Where(o=> o.IsDeleted == false).ToList();
5. }
```

Eksempel på å hente ut all data som ikke er slettet. Hentet fra: /Controllers/UserController.cs

Dette ble også brukt for å hente ut dataen fra SCADA- og Fôringstabellene. Når disse tabellene fikk mye data brukte LINQ alt for lang tid, ettersom LINQ lagrer en referanse til alle objektene. Dette gjør at endringer i objektene enkelt kan bli lagret og oppdatere databasen. I tillegg trengtes feltene til nøstede objekter. I utgangspunktet returneres disse nøstede objektene som *null* med mindre man spesifiserer at de skal hentes ut. Man henter ut disse objektene ved å bruke funksjonen *Include()*, noe som førte til at det tok enda lenger tid å hente ut all dataen.

For å minske tiden spørringen tok, byttet vi til rå SQL ved noen endepunkt der det hentes ut mye data. Dette gikk raskere enn LINQ, men brukte fortsatt lang tid dersom man skulle hente ut flere hundre tusen rader fra databasen. I samarbeid med Tytlandsvik Aqua ble det enighet om at det var behov å maks presentere ett tusen punkter, og bruke snittregning hvis det var flere punkter enn dette. Dette vil både gjøre det enklere å tolke dataen som blir presentert, i tillegg til at SQL spørringen kan optimaliseres.

I SQL kan man bruke *NTILE* til å dele opp radene i ett gitt antall grupper. (julieMSFT, 2017) Dette ble brukt for å dele opp radene i 1000 grupper. Antall rader som blir hentet ut fra databasen bestemmer hvor store gruppene skal være. Dersom det hentes ut 5000 rader vil en gruppe bestå av 5 rader. *NTILE* ble brukt på denne måten, linje 4:

```
1.  FROM
2.      (
3.          SELECT
4.              Chunk = NTILE(1000) OVER(ORDER BY DateRecorded),
5.              *
6.          FROM
7.              [dbo].[RasLogs] as Raslog
8.          Where HallId = {0} AND DateRecorded >= {1} AND DateRecorded
    <= {2}
9.      ) AS T
10.
11.  GROUP BY
12.      Chunk
13.  ORDER BY
14.      ChunkStart;
```

Eksempel på bruk av NTILE

Får å gjøre klar dataen til å presenteres ble det gjort et gjennomsnitt i hver av gruppene. Det vil da si at man ender opp med 1000 gjennomsnitt. Koden for dette ser du under. Gjennomsnittet av ID på linje 9 blir gjort litt annerledes enn det på linje 13, fordi ID-en ikke kan være et desimaltall.

```
1. var valuesString = "";
2. PropertyInfo[] properties = typeof(RasLog).GetProperties();
3. foreach (PropertyInfo property in properties)
4. {
5.     if (property.Name != "Hall" || property.Name != "DateRecorded")
6.     {
7.        if (property.Name == "Id")
8.        {
9.         valuesString += "," + property.Name + " = Cast(AVG( Cast(" +
    property.Name + " as float)) as int)";
10.        }
11.        else
12.        {
13.          valuesString += "," + property.Name + " = AVG(" +
    property.Name + ")";
14.        }
15.     }
16.  }
```

Generere forespørsel basert på Propertyinfo til objektet RasLog. Hentet fra /Controllers/RasLogControllers.cs

Det å ta gjennomsnittet av verdiene på måten det blir gjort her kan føre til at ekstremverdier ikke synes, eller påvirker dataen på en uønsket måte. Dette er noe som potensielt kan endres, men etter dialog med Tytlandsvik Aqua kom det frem en forståelse om at dette ikke er et umiddelbart behov for at ekstremverdier skal bli tatt høyde for. I tillegg kom det frem at et enkelt gjennomsnitt er tilstrekkelig ettersom det eksisterer andre systemer som håndterer ekstremverdier i sanntid, slik at de kan utføre handling på problemer umiddelbart.

Prosessering av filer

Ettersom loggfilene fra foringssystemet skal prosesseres ble det valgt å benytte det Node.js skriptet som ble anvendt til å tolke disse loggfilene. Dette skriptet blir gått nærmere inn på i kapittel 10.5 Databehandling. Ved å gjenbruke dette skriptet sparer man en del tid på denne implementasjonen, Microsoft har til og med tatt seg bryet med å lage en .Net pakke for å kjøre node.js funksjoner i .Net. Denne heter *NodeServices*. For å initialisere en *NodeService* legges følgende til i *Startup.cs* filen, under *ConfigureServices*:

```
services.AddNodeServices();
```

Videre injiseres *NodeServicen* til *RasLogController* slik at man kan nå bruke denne tjenesten til å kjøre *node.js* funksjoner fra kontrolleren.

Kodeutsnittet under viser hvordan filene blir sendt til endepunktet og videre prosessert av node.js skriptet. Det første som skjer er at filene blir sendt inn som en liste med form-filer, dette er listen med filer som skal prosesseres. For hver fil i listen finner man et midlertidig filnavn der filen lagres. Etter at filen er lagret til filsystemet der APIet kjører kalles skriptet som prosesserer logg-filene og returnerer en liste med RAS-logg rader. Disse radene blir videre lagt inn i databasen ved hjelp av Post funksjonen til denne kontrolleren, dette er et enkelt endepunkt som legger til lister med RAS-logg rader inn i databasen, den sjekker også at raden ikke eksister fra før av. Til slutt returnerer endepunktet antall rader som ble prosessert.

```
1. [HttpPost("processLogfile")]
2. [Authorize(Roles.Writer)]
3. public async Task<ActionResult<int>>
   PostFormDataAsync(List<IFormFile> files)
4. {
5.       try
6.       {
7.            var totalAdded = 0;
8.            foreach (var file in files)
9.            {
10.               var filePath = Path.GetTempFileName();
11.                using (var stream = System.IO.File.Create(filePath))
12.                {
13.                     await file.CopyToAsync(stream);
14.                }
15.                var result = await
   _nodeServices.InvokeAsync<List<RasLogRequest>>("wwwroot/NodeScripts/R
   asLogProcessor.js", filePath);
16.                     Post(result);
17.                     totalAdded += result.Count();
18.                     System.IO.File.Delete(filePath);
19.                }
20.                return Ok(totalAdded);
21.             }catch (Exception e)
22.             {
23.                 return StatusCode(500, e.ToString());
24.             }
25.         }
```

Endepunkt som håndterer opplasting av logfiler fra SCADA-systemet. Hentet fra /Controllers/RasLogControllers.cs

10.2.7 Sanntidsutsending av data

Som nevnt tidligere er det valgt at SignalR skal benyttes for sanntidskommunikasjon (krav 3) av data til webapplikasjonen. Frontend kapitelet tar for seg konsumeringen av sanntidsdataen gjennom SignalR, nå skal vi se på logikken som sender ut denne dataen.

For å sette oppe .NET Core prosjektet til å benytte signalR må påfølgende kode legges til i *ConfigureServices* metoden i *Startup.cs* filen. *AddJsonProtocol* spesifiserer at dataen som blir sendt i meldingene er i JSON format, slik at når man sender et C#-objekt vil dette blir omgjort til JSON. Dette er svært nyttig med tanke på at det skal konsumeres av JavaScript i Webapplikasjonen. Man kunne sendt det som ren teks, men da måtte man selv gjort teksten om til JSON.

```
1. services.AddSignalR().AddJsonProtocol();
```

Oppsett av SignalR med JSON protokoll. Hentet fra /Startup.cs

Videre må det legges til et endepunkt for å koble seg til SignalR tjeneren. Dette legges til i *app.useEndpoints* under *Configure* metoden i *Startup.cs*, det spesifiseres også at websockets skal brukes som transportmåte. Her spesifiseres det også en hub, denne er i vår implementasjon tom ettersom det ikke skal sendes data fra klient gjennom SignalR.

```
1. endpoints.MapHub<DataHub>("/datahub", options =>
2. {
3.     options.Transports =
4.         HttpTransportType.WebSockets;
5. });
```

Legge til en hub til et endepunkt, endepunktet /datahub blir her eksponert. Hentet fra /Startup.cs

I kontrolleren man ønsker å sende ut data injiseres tjenesten på samme måte som beskrevet i 10.2.4 Tjenester. Videre for å sende ut data til alle tilkoblede klienter benyttes konteksten på måten vist under. Eksempelet er fra endepunktet *POST /RasLog*, og metoden sender ut den nye dataen hver gang den blir lagt til. Dataen kan videre konsumeres av klientene til å vise sanntidsdata, noe som blir gjort ved dashboardet.

```
this._hubcontext.Clients.All.SendAsync("ReceiveRasLogs", newRasLogs);
```

Sender nye raslogger til alle tilkoblede klienter.. Hentet fra Controllers/RasLogController.cs

10.2.8 Dokumentasjon

For å dokumentere endepunktene og samtidig muliggjøre manuell testing er det valgt å bruke Swagger. Det er særdeles enkelt å sette opp swagger med .NET, følgende kode må legges til i *Configure* metoden i *Startup.cs* filen:

```
1. app.UseSwagger();
2. app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
   "FishyAPI v1"));
```

Oppsett av swagger. Hentet fra /Startup.cs

Swagger ser nå på alle kontroller-filene og genererer en dokumentasjonsside som kan brukes for å manuelt teste API-endepunktene. Denne siden kan nås på *<base-url>/swagger/index.html*.

10.2.9 Caching

Når det kommer til caching er det benyttet to strategier på de fleste endepunktene. Dersom man gjør spørringer som henter data, forsøker man først å hente fra cache, og deretter fra databasen hvis cachen ikke finner noe data. Hente spørringer som ikke finner data i cachen vil da sette ny data til cachen. På spørringer der man endrer på dataen sletter man cache. I kodeutsnittene under kan man se strategien for spørringer hvor man kun henter data og spørringer hvor man endrer på data:

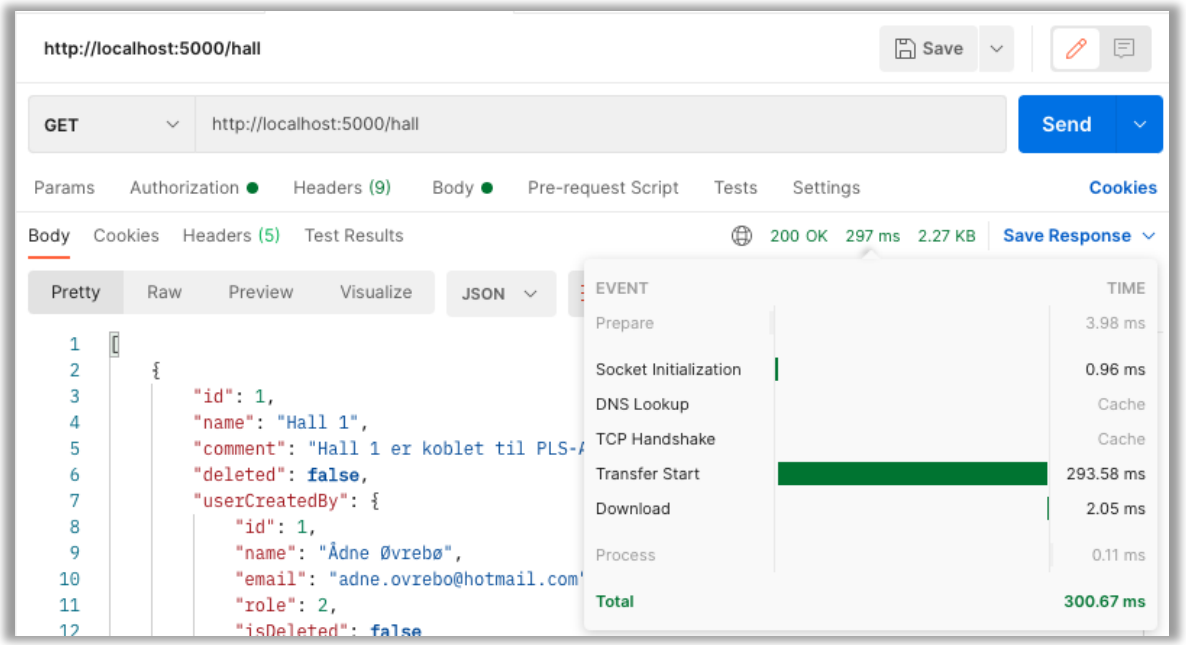
```
1.  [HttpGet("{id}")] // GET <DataKeyController>/<id>
2.  public ActionResult<DataKey> Get(int id)
3.    {
4.          try
5.          {
6.               if (_cache.TryGetValue($"datakey-{id}", out DataKey cachedKeys))
7.               {
8.                   return Ok(cachedKeys); # Dersom cachet verdi finnes
    returneres denne
9.               }
10.
11.              var datakey = _context.DataKeys.Where(datakey => datakey.Id == id
    && !datakey.Deleted).FirstOrDefault();
12.
13.             if (datakey == null)
14.             {
15.                  return NotFound();
16.             }
17.             _cache.Set($"datakey-{id}", datakey); # Cache settes.
18.             return Ok(datakey);
19.     }
20.     catch
21.     {
22.          return StatusCode(500);
23.     }}
```

Eksempel på håndtering av caching i endepunkter som henter data. Hentet fra /Controllers/DataKeyController.cs

```
1. // POST <HallController>
2. [HttpPost]
3. [Authorize(Roles.Writer)]
4. public ActionResult<Hall> Post(HallRequest model)
5. {
6.       try
7.       {
8.          if (_context.Halls.Any(o => o.Name == model.Name &&
   !o.Deleted))
9.           {
10.                 return StatusCode(409, "Hall already exists"); ;
11.           }
12.
13.          var currentUser = _userService.GetCurrentUser(HttpContext);
14.          if (currentUser == null)
15.          {
16.                return StatusCode(500);
17.          }
18.
19.          var hall = _context.Halls.Add(new Hall
20.          {
21.               Name = model.Name,
22.               Comment = model.Comment,
23.               UserCreatedBy = currentUser,
24.
25.           });
26.           _context.SaveChanges();
27.             _cache.Remove(AllHallsCacheKey); # Cache slettes
28.             return Ok(hall.Entity);
29.         }
30.         catch
31.         {
32.             return StatusCode(500);
33.         }}
```

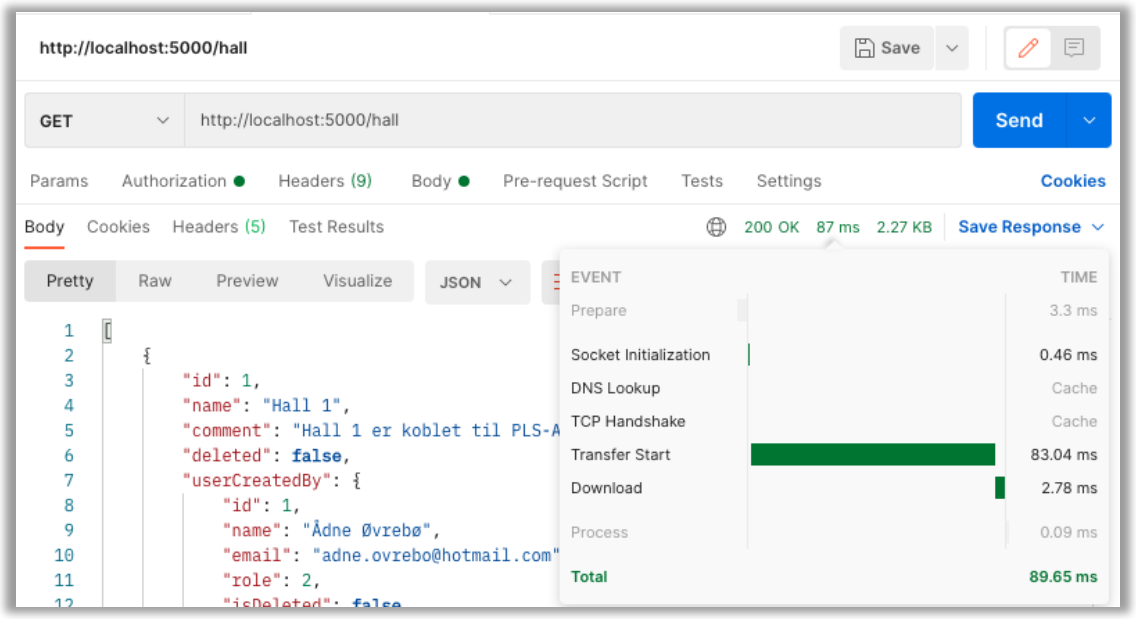
Eksempel på håndtering av caching i endepunkter legger til data. Hentet fra /Controllers/HallController.cs

Men hvor stor er forskjellen med og uten cache? Man kan gjøre et enkelt eksperiment for å undersøke dette: man sender to forsepørsler til det samme endepunktet, et med caching og ett uten. Verktøyet Postman er benyttet til å gjøre denne undersøkelsen. Figuren under viser responstiden uten caching. Tiden som er interessant her er *Transfer Start*, da dette er tiden fra forespørselen er sendt til resultatet skal sendes tilbake. I den påfølgende figuren er denne 293,68 ms. Forespørselen i dette eksempelet er gjort mot endepunktet */hall* som lister ut alle hallene som er registrert.



Figur 55: Forespørsel til /hall uten caching

Påfølgende figur viser tiden dette tok etter caching, her ser man at *Transfer Start* er 83,04 ms.



Figur 56: Forespørsel til /hall med caching

Ved dette eksempelet, som gjør en enkel spørring mot databasen, ser man at det tar ca. tre ganger så lang tid uten caching. Man kan ut ifra dette se at man får en signifikant forskjell i responstiden samtidig som man unngår unødvendige spørringer mot databasen.

10.2.10 Logging

Det er valgt å benytte logging i applikasjonen på steder hvor det kan være viktig å kunne spore tilbake hva som har skjedd. Et eksempel på dette vil være brukerinnlogginger. Her vil det være fordelaktig å kunne spore innloggingsforsøk og hvilken bruker som ble forsøkt å logge inn med. I underkapittelet 10.2.5 Tjenester vises det det hvordan man kan benytte injiserte tjenester, logger er en slik. En logger krever også et litt annet oppsett, .NET kommer med innebygd støtte for logging.

For å sette opp logging i .NET må følgende kode legges inn i *progam.cs* filen:

```
Host.CreateDefaultBuilder(args).ConfigureLogging()
```

Denne må kjøres på *CreateHostBuilder* etter at *HostBuilderen* er opprettet.

```
_logger.LogInformation($"Failed login attempt by {model.Email}");
```

Eksempelet er hentet fra Auth-kontrolleren hvor innloggingsforsøk som feiler logger hvilken epost som forsøkte å logge inn. Ved å logge hvilke brukere som forsøker å logge seg inn er det mulig å oppdage uautoriserte personer

10.2.11 IP-Ratebegrensing

Endepunktet */signIn* kan være sårbart for brute force angrep. For å forhindre et brute force angrep vil det i dette tilfellet være lurt å legge inn en mekanisme som forhindrer angrepet, der det i dette tilfellet blir benyttet en pakke som heter *AspNetCoreRateLimit*.
Pakken blir initialisert og videre konfigurert ved å legge følgende initialiseringskode i *Startup.cs* under *Configure* Service. Kort fortalt henter man først ut innstillingene fra *appsetting*. Videre legges det til en *IpPolicyStore* som benytter i minne caching for å ta vare på tilstanden, dette er fordi antall forsøk lagres per IPadresse. Deretter initialiserer man tjenesten som håndterer rate grensen.

```
1. services.Configure<IpRateLimitOptions>(Configuration.GetSection("IpRa
   teLimit"));
2. services.AddSingleton<IIpPolicyStore,
3. MemoryCacheIpPolicyStore>();
4. services.AddSingleton<IRateLimitCounterStore,
5. MemoryCacheRateLimitCounterStore>();
6. services.AddSingleton<IRateLimitConfiguration,
7. RateLimitConfiguration>();
```

Oppsett av IP-Rate begrenser. Hentet fra /Startup.cs

I *appsettings.json* konfigurerer man hvordan denne rate grensen skal fungere. I eksempelet under er det satt opp hva navnet på de nødvendige feltene i HTTP-forespørsel-hodet skal være og hvilken statuskode som skal returneres (429: For mange forespørsler). Videre er det spesifisert under *GeneralRules* at endepunktet Auth skal begrenses til 5 forespørsler i minuttet.

```
1.   "IpRateLimit": {
2.     "EnableEndpointRateLimiting": true,
3.     "RealIPHeader": "X-Real-IP",
4.     "ClientIdHeader": "X-ClientId",
5.     "HttpStatusCode": 429,
6.     "GeneralRules": [
7.       {
8.         "Endpoint": "*:/Auth/*",
9.         "Period": "1m",
10.          "Limit": 5
11.        }
12.      ]
13.    }
```

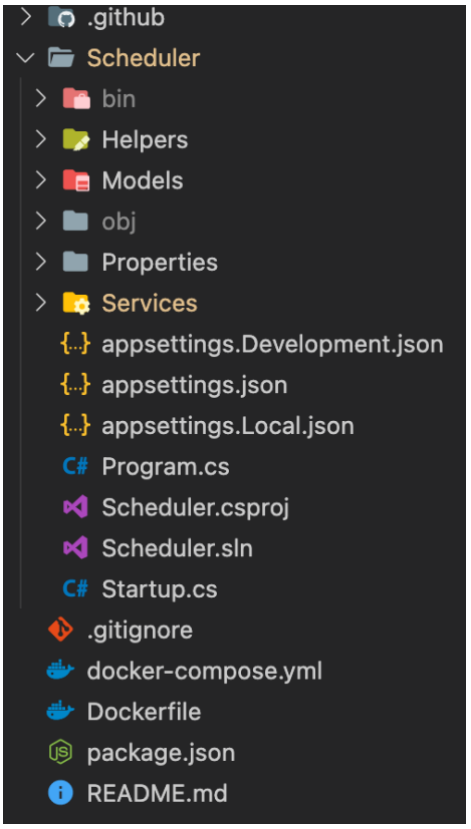
Konfigurering fil for endepunktene under /Auth. Hentet fra /appsettings.json

10.3 Tjeneste for innhenting av informasjon

Oppsettet for denne tjenesten er i bunn og grunn helt lik som for APIet, et .NET Core prosjekt. Kodestrukturen er noe annerledes ettersom denne fungerer som en klient i motsetning til API-tjenesten som fungerer som en tjener.

10.3.1 Kodestruktur

Som allerede nevnt er strukturen svært lik for API, under følger en figur som viser kodestrukturen. Mappene som er av interesse i denne tjenesten er *Helpers*, *Models* og *Services*, de resterende hjelpefilene har samme funksjonalitet som beskrevet tidligere i kapittelet rundt implementasjonen av API.



Figur 57: Kodestruktur tjeneste for innhenting av data

10.3.2 Autentisering

Det er opprettet en rolle av typen *Scheduler* som kun har tilgang til å legge inn data fra SCADA- og fôringssystemet. En bruker med denne rollen blir opprettet for å kunne autentisere denne tjenesten mot APIet.

Klassen *JWTHelper* under *Helpers* inneholder funksjoner for å håndtere JWT som blir benyttet. Denne klassen er implementert som en statisk klasse som vil si at man ikke behøver å opprette en instans av objektet før klassen kan benyttes. Det er henholdsvis to funksjoner i denne klassen: en for å undersøke om JWT er utgått og en funksjon som logger inn og får en ny gyldig token.

Under følge koden for metoden som logger inn tjenesten. Se kommentarer for inngående funksjonalitet.

```
1. public static async Task GetNewTokenAsync()
2. {
3.     try
4.     {
5.         // Oppreter en forespørsel med brukernavn og passord fra appsettings
6.         var payload = new AuthRequest
7.         {
8.             Email = Startup.StaticConfig.GetConfig("LoginUser"),
9.             Password = Startup.StaticConfig.GetConfig("LoginPass"),
10.        };
11.        // Converterer objektet til json
12.        var strigifyedPayload = JsonConvert.SerializeObject(payload;
13.
14.        // Opprettet innholdet til http forespørselen
15.        var httpContent = new StringContent(strigifyedPayload, Encoding.UTF8,
   "application/json");
16.
17.        using (var httpClient = new HttpClient())
18.        {
19.            // Kjører post mot APIet
20.            var httpResponse = await
   httpClient.PostAsync(Startup.StaticConfig.GetConfig("ApiEndpoint") +
   "Auth/signIn", httpContent);
21.
22.            // Dersom forespørselen har et innhold leses innhold og APIToken
   (JWTToken) settes
23.            if (httpResponse.Content != null)
24.            {
25.                var responseContent = await
   httpResponse.Content.ReadAsStringAsync();
26.                var response =
   JsonConvert.DeserializeObject<AuthResponse>(responseContent);
27.                APIToken = response.Token;
28.            }
29.        }
30.    }
31.    catch (HttpRequestException e)
32.    {
33.        Console.WriteLine("\nException Caught!");
34.        Console.WriteLine("Message :{0} ", e.Message);
35.    }
36.}
```

Metode for innlogging av tjenesten for innhenting av informasjon. Hentet fra /Services/JWTServices.cs

I neste kodeutsnitt er metoden som undersøker om JWT er gyldig. Se kommentarer for detaljert funksjonalitet.

```
1. public static async Task InspectTokenAsync()
2. {
3.     // Dersom JWTtoken ikke er tom
4.     if (APIToken != "")
5.     {
6.         // Undersøker hvor lenge til token går ut
7.         var tokenExpiresAt = JWTService.GetExpiryTimestamp(APIToken);
8.         int result = DateTime.Compare(tokenExpiresAt, new DateTime());
9.         // Dersom JWT token har gått ut hentes en ny
10.        if (result < 0)
11.        {
12.            await JwtHelper.GetNewTokenAsync();
13.        }
14.    }
15.    else { await JwtHelper.GetNewTokenAsync(); };
16. }
```

Metode for å inspisere om token er gyldig, evt. hente ny dersom den er gått ut. Hentet fra /Services/JWTServices.cs

Disse metodene blir brukt i forbindelse med å benytte en JWT ved kall mot API-tjenesten for å sikre at man har en gyldig token.

10.3.3 De ulike jobbene

Det er opprettet to klasser, en for å laste opp data fra SCADA-systemet og en for å laste opp data fra fôringssystemet. Ettersom Tytlandsvik Aqua hadde problemer med sine systemer, som videre førte til problemer med å gi oss de riktige tilgangene i tide, ble det i første omgang opprettet metoder som lastet opp tilfeldig data ved jevne mellomrom. Disse klassene ligger under *Services* mappen.

Metodene som genererer tilfeldig data er skilt ut fra metodene som sender data, slik at man enklere kan koble datahentingen til systemene til Tytlandsvik Aqua i fremtiden i stedet for å generere tilfeldig data. Kodeutsnittet under viser hvordan man først sjekker og eventuelt oppdaterer JWTen, henter data og kaller metoden som sender data til APIet:

```
1. public static async Task GetAndUploadRasAsync()
2. {
3.     await JwtHelper.InspectTokenAsync();
4.     var data = GetAllResData();
5.     _ = UploadDataAsync(data);
6. }
```

Metode for å laste opp raslogger. Hentet fra Scheduler/Services/Rasservice.cs.

Metoden *UploadDataAsync* over setter først autoriseringstoken til HTTP-hodet. Deretter konverteres dataen til JSON og sender deretter dataen til riktig API-endepunkt ut ifra konfigurasjonsinnstillinger, som vist under:

```
1.  public static async Task UploadDataAsync(List<RasLog> data)
2.  {
3.      try
4.      {
5.          using (var httpClient = new HttpClient())
6.          {
7.              // Setter autoriserings header
8.              httpClient.DefaultRequestHeaders.Authorization
9.                          = new AuthenticationHeaderValue("Bearer", JwtHelper.A
    PIToken);
10.             // konverterer dataen til JSON
11.             var strigifyedPayload = JsonConvert.SerializeObject(data);
12.             var httpContent = new StringContent(strigifyedPayload, Encoding.U
    TF8, "application/json");
13.             // Sender dataen til APIet
14.             var httpResponse = await httpClient.PostAsync(Startup.StaticConfi
    g.GetConnectionString("ApiEndpoint") + "RasLog", httpContent);
15.         }
16.     }
17.     catch (HttpRequestException e)
18.     {
19.         Console.WriteLine("\nException Caught!");
20.         Console.WriteLine("Message :{0} ", e.Message);
21.     }
22. }
```

Metode for å laste opp data til et endepunkt. Hentet fra Scheduler/Services/Rasservice.cs.

10.3.4 Hangfire

Tredjepartsbiblioteket Hangfire er benyttet for å kjøre funksjoner med jevne mellomrom, samtidig som det tilbys et brukergrensesnitt for å observere og trigge jobbene.

Følgende kode ble lagt til i *Startup.cs* filen for å sette opp Hangfire-tjenesten. Koden setter opp hangfire og sier hvilken database som skal brukes gjennom å sette en tilkoblingsstreng som blir hentet fra *appsettings.json*. Det er valgt å benytte en separat database for å skille data som brukes i API mot data som Hangfire lagrer.

```
1. services.AddHangfire(configuration => configuration
2.     .UseRecommendedSerializerSettings()
   .UseSqlServerStorage(Configuration.GetConnectionString("HangfireConne
   ction"));
3.
4. services.AddHangfireServer();
```

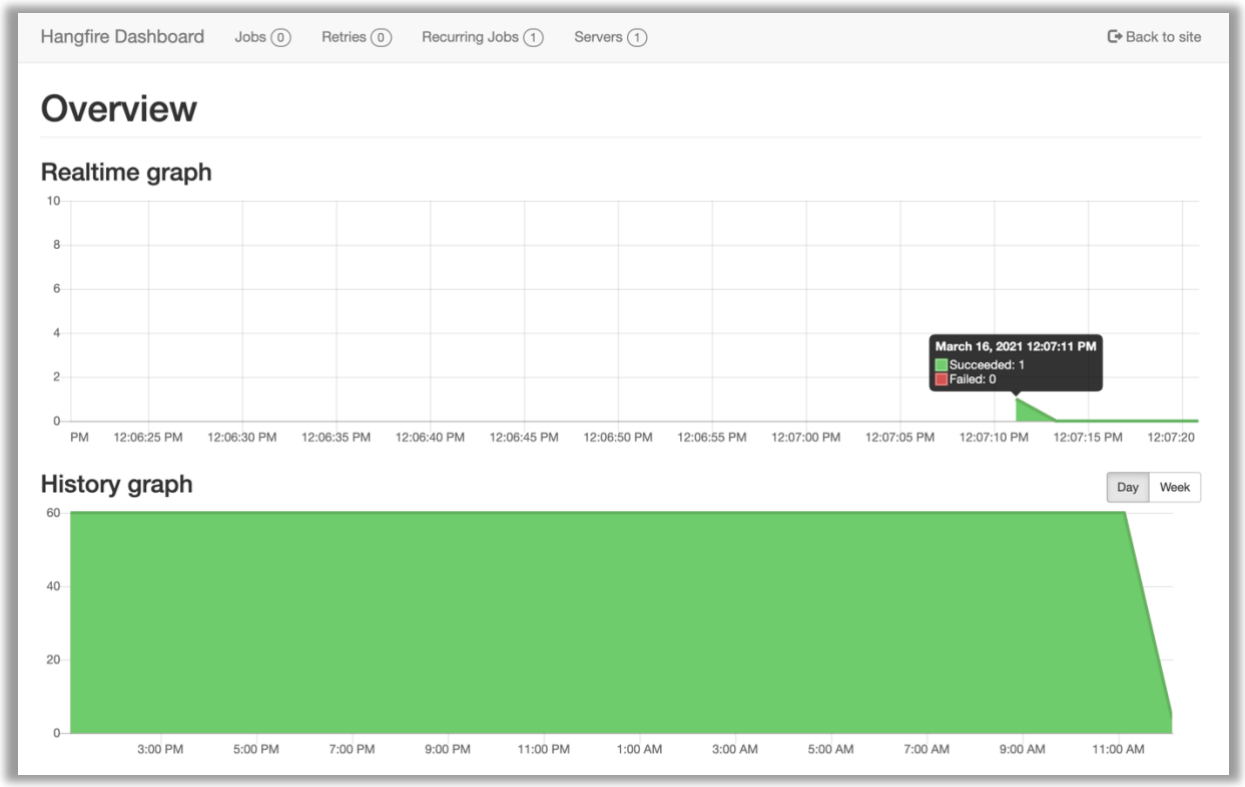
Oppsett av Hangfire. Hentet fra /Startup.cs

For å sette opp et Hangfire-dashboard ble koden under benyttet i *Startup.cs* filen under *Configure* metoden:

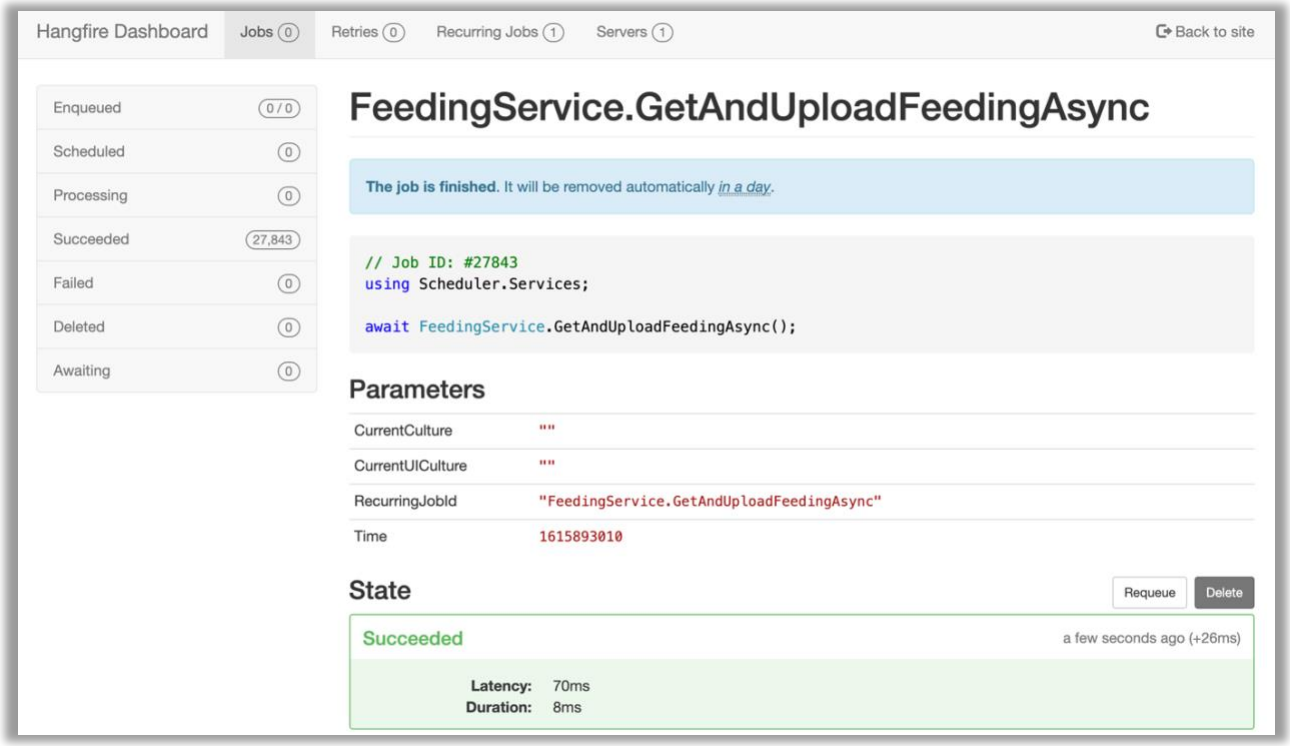
```
1. app.UseHangfireDashboard("", new DashboardOptions
2. {
3.     Authorization = new[] { new
   HangfireCustomBasicAuthenticationFilter { User = "<brukernavn>", Pass
   = "<passord>" } }
4. });
```

Oppsett av Hangfire Dashboard. Hentet fra /Startup.cs

Dette setter opp et dashboard som kan brukes til å observere status på jobbene, dashbordet er beskyttet med et brukernavn og passord. Under vises eksempel på sanntidsgraf for kjørende jobber samt oversikt over en jobb som er fullført. Dersom en jobb ikke fullfører, vil man her kunne se feilmelding og hvor i koden jobben feilet.



Figur 58: Hangfire dashboard sanntidsgraf over jobber som fullfører og feiler.



Figur 59: Hangfire dashboard viser jobbene som er registrert, som også kan trigges her.

Jobbene settes opp ved å registrere dem i *Startup.cs* filen på som vist under. Her kjøres de to metodene som laster opp data hvert 10. minutt.

```
1. CronExpression tenMinutes = CronExpression.Parse("*/10 * * * *");
2. RecurringJob.AddOrUpdate(() => RasService.GetAndUploadRasAsync(),
   tenMinutes.ToString);
3. RecurringJob.AddOrUpdate(() =>
   FeedingService.GetAndUploadFeedingAsync(), tenMinutes.ToString);
```

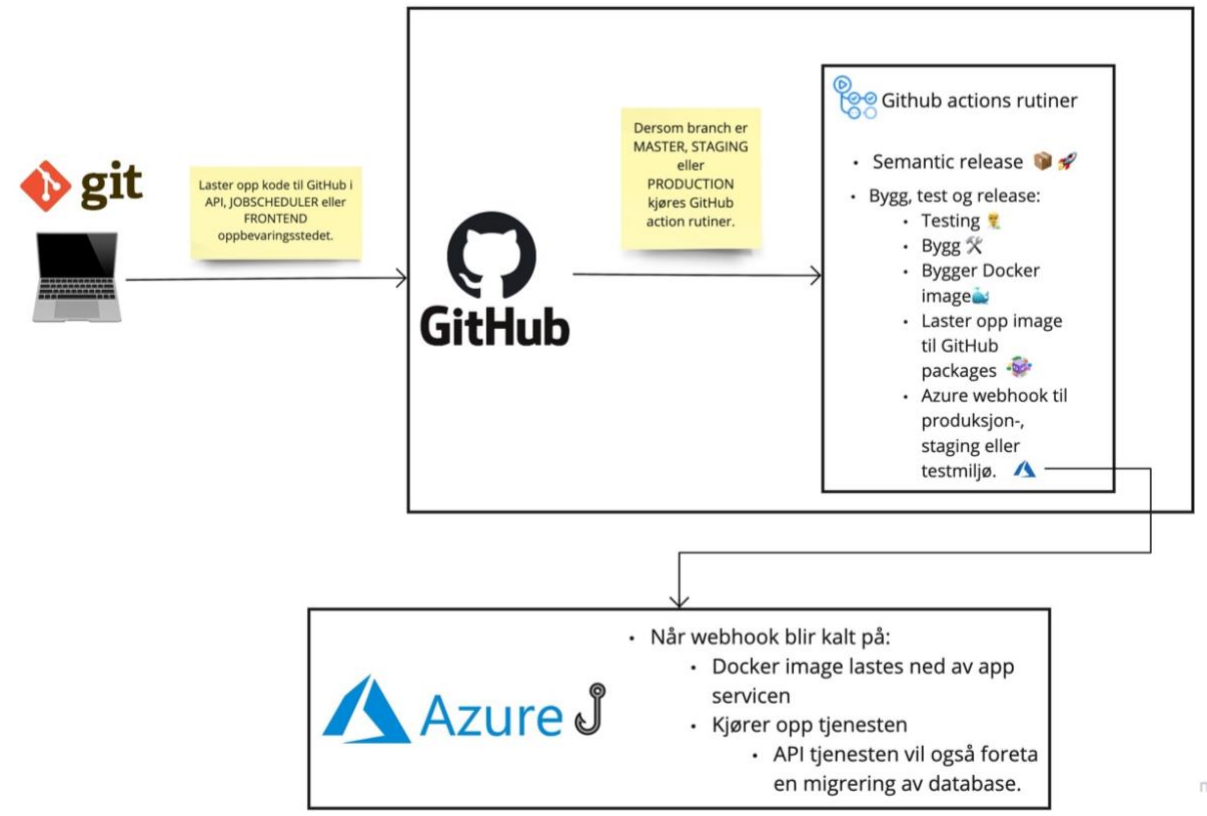
Oppsett av jobber i hangifre eksempel. Hentet fra /Startup.cs

10.4 Automatisk bygging og utrulling

For å effektivisere bygging og utrulling av de ulike tjenestene er det valgt å sette opp en flyt som automatisk bygger og ruller ut nye versjoner til de ulike miljøene. Dette var ikke et dirkete krav, men vil være fordelaktig opp mot krav 13 som omhandler å legge opp til enklere videreutvikling. Byggestegene kan også sees på som en dokumentasjon på hvordan tjenesten kan bygges lokalt. Figur 60 viser hvordan denne flyten er satt opp, der følgende steg skjer:

1. Man laster opp koden sin til GitHub til en av oppbevaringsstedene til prosjektet.
2. Dersom branchen som er pushet til er enten MASTER, STAGING eller PRODUKSJON kjøres det en GitHub Actions rutine som gjør følgende.
    - Trigger Semantic release.
    - Tester og bygger løsningen.
    - Laster opp det nybygde Dockerbildet til GitHub Packages (Github Container Registry).
    - Dersom alle stegene går gjennom uten feil trigges en webhook mot Azure.
3. Azure Webhook blir kalt, som laster ned nyeste Dockerbildet og kjører opp tjenesten.

Denne flyten er benyttet for samtlige tjenester.



Figur 60: Automatisk bygging og utrulling flyt

10.4.1 Dockerfil

For å kunne kjøre tjenestene i Dockerbeholdere er det nødvendig med Dockerfiler som bygger applikasjonene til Dockerbilder. Under følger et eksempel på hvordan det er gjort i API-prosjektet. De resterende Dockerfilene ligger i rot-mappen til alle tjenestene, der filene starter med navnet *Dockerfile*. Det er videre opprettet en Dockerfil per miljø, for eksempel vil en Dockerfil som bygger til miljøet development heter *Dockerfile.Development*.

```
1. # Bildet som blir brukt til å bygge applikasjonen (.NET SDK)
2. FROM mcr.microsoft.com/dotnet/sdk:5.0-alpine AS builder
3.
4. # Setter navnet på mappen som man "arbeider" i.
5. WORKDIR /sln
6.
7. # Kopierer alle filene over fil docker-bildet.
8. COPY ./FishyAPI .
9.
10.  # Installerer nodejs og installerer pakkene brukt i NodeScripts
11.  RUN apk add nodejs-current npm
12.  RUN cd wwwroot/NodeScripts && npm install
13.
14.  # Bygger applikasjonen for utgivelse
15.  RUN dotnet publish -c Release -o /sln/artifacts
16.
17.  # Kjøretid bildet som skal benyttes
18.  FROM mcr.microsoft.com/dotnet/aspnet:5.0
19.
20.  # Installerer node på bildet som skal bruker i kjøretid.
21.  RUN apt-get update -yq \
22.      && apt-get install curl gnupg -yq \
23.      && curl -sL https://deb.nodesource.com/setup_10.x | bash \
24.      && apt-get install nodejs -yq
25.
26.
27.  WORKDIR /app
28.
29.  # Koden som blir kjørt når man starter docker-beholderen.
30.  ENTRYPOINT ["dotnet", "FishyAPI.dll", "--environment=Development"]
31.  COPY --from=builder ./sln/artifacts .
```

Eksempel på Dockerfile for bygging av API. Hentet fra Dockerfile.development i API prosjektet

For å kjøre en bygg lokalt for å teste om dette fungerer kan man kjøre kommandoen.

```
docker build -t ImageNavn Dockerfile.<Miljø>
```

Dockerbildet kan videre kjøres opp og eksponeres mot en lokal port. Eksempelvis som kommandoen vist under, hvor port 80 i beholderen eksponeres gjennom localhost på port 3000 dermed kan nås på localhost:3000.

```
docker run -p 3000:80 ImageNavn
```

10.4.2 Husky

Husky er satt opp slik at man verifiserer at koden bygger lokalt for å unngå at man laster opp kode til GitHub som ikke fungerer.

Følgende utsnitt er lagt i *package.json* filen til eksempelvis frontend prosjektet. Denne konfigureringen tilsier at før man pusher kode i Git skal man kjøre denne valgte kommandoen, i vårt tilfelle en bygg av prosjektet kjøres før koden lastes opp.

```
1.   "husky": {
2.     "hooks": {
3.       "pre-push": "npm run build"
4.     }
5.   },
```

Oppsett av Husky for å bygge løsningen før push. Hentet fra /package.json i Frontend prosjektet.

10.4.3 Github actions

Ettersom koden for bygging og testing er lagt inn i Dockerfiler skal det nå kun være nødvendig å bygge disse filene og laste opp Dockerbildet til et oppbevaringssted. Det er valgt å anvende GitHub Packages slik at koden og Dockerbildene oppbevares sammen og da gjør det enkelt å finne et Dockerbilde til en bestemt versjon. Under følger eksempelet på oppsett av GitHub actions skriptet som er benyttet for å automatisk bygge og utgi kode til produksjonsmiljøet i FRONTEND. De ulike stegene er forklart med kommentarer.

```
1.  name: Production - Docker Build Frontend
2.
3.  # Når man laster opp kode til produksjons branchen trigges denne
    arbeidsflyten.
4.  on:
5.    push:
6.      branches: production
7.
8.  jobs:
9.    main:
10.     runs-on: ubuntu-latest
11.     steps:
12.       # Setter opp miljø for å kunne kjøre docker
13.       - name: Set up QEMU
14.         uses: docker/setup-qemu-action@v1
15.       # Setter opp miljø for å kunne kjøre docker
16.       - name: Set up Docker Buildx
17.         uses: docker/setup-buildx-action@v1
18.       # Logger inn til GitHub conatiner registry
19.       - name: Login to GitHub conatiner registry.
20.         uses: docker/login-action@v1
21.         with:
22.           registry: ghcr.io
23.           # Secrets blir hentet fra instillinger i GitHub
24.           username: ${{ secrets.DOCKER_USERNAME }}
25.           password: ${{ secrets.DOCKER_REGISTRY_TOKEN }}
26.         # Docker-bildet blir bygget og lastet opp til den innloggede
    container registry brukeren.
27.       - name: Build and push
28.         id: docker_build
29.         uses: docker/build-push-action@v2
30.         with:
31.           # Spesifiserer dockerfilen som skal benyttes
32.           file: Dockerfile.production
33.           push: true
34.           tags: ghcr.io/fishy-bachelor/frontend:production
35.
36.       # Skriver ut informasjon om det opplastede bildet for å kunne se
    tilbake på.
37.       - name: Image digest
38.         run: echo ${{ steps.docker_build.outputs.digest }}
39.
40.       # Gjør et kall mot en WebHook i azure som laster ned det nye docker-
    bildet og erstatter instansen som allerede kjører med den nye.
41.       - name: Trigger deploy
42.          run: curl -X POST '${{ secrets.AZURE_APPSERVICES_HOOK_PROD }}' -H ""
    -d ""
```

GitHub Actions skript for automatisk bygging og utrulling av FRONTEND til Azure «.github/wokrflows/docker-build-production.yaml».

For å tillate at Azure automatisk laster ned det nyeste Dockerbildet må man tillate bruk av webhook i AppServicen i Azure Portal. Dette gjør man under *Container setting* og man får da en link til denne webhooken.

Det er nå satt opp en flyt som automatisk bygger og utgir koden. Ved å gjøre dette sikrer man at bygg-stegene alltid er de like samtidig som det skjer uten noen form for interaksjon som vil være tidsbesparende i lengden.

10.5 Databehandling

Hovedmomentet i dette prosjektet er dataen som skal behandles. For å kunne benytte dataen fra de eksisterende systemene må denne struktureres på et format som er mer anvendelig. Enkelte systemer strukturerer dataen på en slik måte at man ikke kan bruke den direkte, man må gjøre en restrukturering.

10.5.1 Manuell data

Den manuelle dataen er per dags dato lagret i Excel-filer. Ved å eksportere disse filene til formatet CSV kan man lage et script som leser inn dataen og legger det inn i API-et på riktig måte. CSV er en forkortelse for Comma Seperated Values (Lahar, 12), som er filer med rader og kolonner separert ved bruk av komma.

For å lese inn dataen til APIet ble det opprettet et node.js script som leser inn radene fra den aktuelle CSV-filen og legger til *HallID* slik at dataen kan knyttes til riktig hall. Dataen ble videre lagt inn i et JSON-objekt og deretter sendt til APIet via POST */ManualInput*.
POST av data til API-endepunktet ble gjort som vist i kodeutsnittet under.
Tredjepartsbiblioteket fetch ble her anvendt, denne fremgangsmåten blir brukt i samtlige skripts hvor data skal sendes til et API-endepunkt.

```
1. const result = await fetch(url, {
2.    method: "POST", // Hvilken API-metode som skal brukes
3.    headers: {
4.      "Content-Type": "application/json",// Hvilket format dataen
   sendes i
5.       Authorization: apiKey, // API-nøkkel for autorisering
6.      },
7.    body: JSON.stringify(payload), // Det som skal sendes gjøres om
   til JSON
8. });
```

Eksempel på å poste data med autentiseringstoken i node.js.

Skriptet som ble brukt kan finnes i mappen *SCRIPTS/manual-data-transfer/index.js* i den vedlagte koden. Koden for dette skriptet er svært simpel og blir derfor ikke gått noe nærmere inn på.

10.5.2 SCADA-systemet

Dataen fra SCADA-systemet ble gitt i form av logger som hadde en .DAT ending, som ligger på en FTP-server. Loggene er delt inn i 3 filer per dag, der hver dag har en <Dato>-String, -Float og - Tagname. Det var vanskelig å finne noe nyttig informasjon rundt dette systemet ettersom leverandøren heller ikke hadde informasjon om formatet til loggene. Det er også opplyst om at loggene blir skrevet til fra en database. Arbeidet med å lese logg-filene programmatisk ble mer krevende enn først antatt.

Første del av dette arbeidet var å finne ut hvilket format filene hadde. Dersom .DAT filene åpnes direkte i en IDE får man ingenting fornuftig ettersom dette er binærfiler av et bestemt format. Filtypen .DAT ga heller ingen indikasjon rundt hvilket format dette kunne være ettersom .DAT filer er en generell datafil som inneholder binærdata. Et gammelt 32-bits Windows-program ble overlevert som kunne lese filene, informasjon rundt dette programmet var ikke til å finne.

Utgangspunktet er relativt diffust, men det er noe å gå etter:

- Loggene er delt opp i tre ulike filer per dag
- Filene blir logget til fra en database
- Systemet er levert av Rockwell Automation.
- Verdier å verifisere mot.

En del søking på nettet gjør at fokuset rettes mot et filformat som har forkortelsen DBF. Filtypen DBF er en gammel filtype som tidligere ble mye brukt i databaser. Dette ble starten på et node.js script for å programmatisk kunne lese dataen fra loggene.

Tredjeparts bibliotek @episage/dbf-parser ble anvendt til å forsøke å lese disse filene. Koden under forklarer fremgangsmåten for å lese filen.

```
1. var fs = require("fs");
2. var Parser = require("@episage/dbf-parser");
3.
4. // Oppretter en ny parser til å lese filen
5. var parser = Parser(
6.     fs.createReadStream(__dirname + `/data/input/2020 02 17 0000
   (Float).DAT`),
7.     "binary"
8.   );
9.
10.  // Logger header
11.  parser.on("header", (h) => {
12.      console.log(h)
13.  });
14.
15.  // Logger radene
16.  parser.on("record", (record) => {
17.      console.log(record)
18.  });
```

Lesing av dbf-filer i node.js

Dette enkle skriptet leser hodet og radene i den tilhørende DBF-filen. I første omgang testes det på en Float fil.

Hodet som ble skrevet ut var følgende:

```
1.  {
2.      type: '\x03',
3.      dateUpdated: 2020-02-17T23:00:00.000Z,
4.      numberOfRecords: 240814,
5.      start: 289,
6.      recordLength: 39,
7.      fields: [
8.        {
9.          name: 'Date',
10.         type: 'D',
11.         displacement: 1,
12.         length: 8,
13.         decimalPlaces: 0
14.       },
15.       {
16.         name: 'Time',
17.         type: 'C',
18.         displacement: 9,
19.         length: 8,
20.         decimalPlaces: 0
21.       },
22.       {
23.         name: 'Millitm',
24.         type: 'N',
25.         displacement: 17,
26.         length: 3,
27.         decimalPlaces: 0
28.       },
29.       {
30.         name: 'TagIndex',
```

```
31.         type: 'N',
32.         displacement: 20,
33.         length: 5,
34.         decimalPlaces: 0
35.       },
36.       {
37.         name: 'Value',
38.         type: 'C',
39.         displacement: 25,
40.         length: 8,
41.         decimalPlaces: 0
42.       },
43.       {
44.         name: 'Status',
45.         type: 'C',
46.         displacement: 33,
47.         length: 1,
48.         decimalPlaces: 0
49.       },
50.       {
51.         name: 'Marker',
52.         type: 'C',
53.         displacement: 34,
54.         length: 1,
55.         decimalPlaces: 0
56.       },
57.       {
58.         name: 'Internal',
59.         type: 'C',
60.         displacement: 35,
61.         length: 4,
62.         decimalPlaces: 0
63.       }
64.     ]
65.   }
```

Hodet til dbf-loggfilene.

Dette var lovende! Nå har man en oversikt over hvilke felter som er i strukturen og hvilken datatype disse er.

Forkortelse	Type	Tillegg
C	Character	ASCII tekst.
D	Date	Dato for opprettelse
F	Floating Point	
N	Numeric	

Tabellen er utarbeidet fra vedlegg 16.4 DBF field types and specifications.

De tekniske detaljene rundt den videre strukturen er irrelevant ettersom det benyttes et tredjepartsbibliotek for å lese de ulike radene, men det kan nevnes at informasjon i hodet er essensiell dersom man skal skrive en egen DBF-leser.

Videre må det undersøkes hvordan en rad faktisk ser ut. Skriptet som ble anvendt tidligere gir ut følgende på en *record*.

```
{
    '@sequenceNumber': 1,
    '@deleted': false,
    Date: '20200217',
    Time: '00:00:01',
    Millitm: 210,
    TagIndex: 1,
    Value: '\x00\x00\x00\x00\x00dZ@',
    Status: '',
     Marker: 'B',
     Internal: 'ÿÿÿÿ'
}
```

Eksempel på én linje i dbf filen.

I utsnittet over kan man se at man får mange av feltene. De viktigste for oss er *Date*, *Time*, *TagIndex* og Value. *TagIndex* sier kanskje ikke seg selv helt enda, men denne skal undersøkes senere. Som man kan se gir feltet *Value* ikke et tall som er det man ønsker. Det kan være rimelig å tro at dette feltet er lagret med feil format, man har heldigvis noe å gå etter. Definisjonen på type C (Character) er at det er ASCII-tekst, dette kan forhåpentligvis være en del av løsningen på å få dette over til en verdi som kan anvendes.
Dersom man printer ut raden på følgende måte:

```
parser.on("record", (record) => {
    console.log(record.Value)
});
```

Logge hver linje fra dbf-filen.

Får man at verdien på raden over er: \x00\x00\x00\x00\x00dZ@, eller dZ@. Det kan være rimelig å anta at binærrepresentasjonen av denne verdien vil være det tallet man leter etter. En litt mer håndterlig form for representasjon av denne strengen vil være på heksadesimal.

Ved å bruke tabellen fra http://www.asciitable.com kommer man frem til følgende resultat på heksadesimal form: 0x655a40. Videre ble metoden for å finne frem til riktig format å prøve ulike kalkulatorer på nettet for å undersøke hvilken talltype dette var. Etter en del testing ga følgende omformer: https://gregstoll.com/~gregstoll/floattohex/ riktig resultat. 0x645A40 reversert og konvertert til en double ga resultatet: 105.5625 som stemmer overens med hva det utleverte programmet ga. Det var altså ikke en float som man kunne tolke ut ifra filnavnet. Nå gjenstår det kun å gjøre denne omformingen i skriptet.

I kodeutsnittet under gjøres ASCII-tegnene om til heksadesimal i reversert rekkefølge, derav *while(i--).* Ettersom heksadesimale skal gjøres om til en double-verdi som er av lengde 16 (64 bit) må det legges til resterende tomme verdier for å kunne gjøre denne konverteringen. Videre leses verdien fra heksadesimal til double, vist på linje 14. Den nye verdien overskriver den gamle verdien og man har nå riktig verdi her, 105.5625.

```
1.   parser.on("record", (record) => {
2.     var i = record.Value.length;
3.     hexastring = "";
4.     while (i--) {
5.       const char = record.Value[i];
6.       const hexRep = hex(char);
7.       hexastring += hexRep.toString(16);
8.     }
9.
10.      while (hexastring.length < 16) {
11.        hexastring = hexastring + "0";
12.      }
13.
14.      const value = Buffer(hexastring, "hex").readDoubleBE(0);
15.
16.      record.Value = value;
17.    });
```

Omgjøring av ascii streng til double verdi.

Videre må det undersøkes hvilken sensor fra anlegget denne verdien kommer fra, her må filen *<Dato>-(Tagname)* undersøkes ettersom en verdi er knyttet til en *TagIndex*. Samme skript som fra starten på dette under-kapitelet gir følgende resultat:

```
1. {
2.   '@sequenceNumber': 1,
3.   '@deleted': false,
4.   Tagname: '[PLC_A]PROGRAM:SMOLT_A.AOI_DEGASLEVEL.HAI_PV',
5.   TTagIndex: 1,
6.   TagType: NaN,
7.   TagDataTyp: NaN
8. }
```

En linje fra en TagIndex fil som tilhører en loggfil.

Man kan bruke denne indeksen til å knytte verdien til en sensor. Alle navnene med indeks til sensorene skrives til en CSV fil for å enkelt kunne få en oversikt over strukturen på navnene til de ulike sensorene.

Etter litt undersøkelse kunne man se at alle verdiene var prefikset med enten PLC_A, PLC_B eller PLC_C. Dette viste seg å være hvilken PLS (eng.: Programmable logic controller) sensorene var koblet til. PLC_A og PLC_B var henholdsvis koblet til Hall 1 og Hall 2, PLC_C er koblet til et kalkanlegg. Dette gjorde det mulig å koble dataen til de ulike hallene. Skriptet ble modifisert til å fjerne prefikser og andre kjennetegn som kunne knytte en sensor til en hall, på denne måten får man de samme nøklene til de ulike hallene, i forespørselen til APIet sendes det med hvilken *HallId* dataen hører til. Dataen fra SCADA-systemet er nå strukturert på en slik måte at den kan sendes inn til APIet for deretter å bli lagt inn i databasen. Skriptet ble videreutviklet til å laste opp dataen fra alle filene i *input* mappa. Resterende delen av skriptet kan finnes under *SCRIPTS/ras-logs /index.js* i vedlagt kildekode.

10.5.3 Foringssystemet

Foringssystemet lagrer logger i en SQL-database, noe som gjør det enklere å hente ut denne dataen ettersom det er mer kjent for oss. Strukturen til dataen på foringssystemet er strukturert på en slik måte at man nærmest kan legge dataen rett inn i vårt system, det eneste som trengs er å koble dataen fra en logg til et kar. Dette blir gjort ved å tilegne raden riktig *PoolID*.

Dataen er strukturert på følgende måte:

- **LogDateTime**: Tidspunkt for logging.
- **Type**: Type fôr.
- **FromEquipment**: Fra hvilket utstyr.
- **ToEquipment**: Til hvilket utstyr.
- **Amount**: Mengde fôring.

Det eneste som blir lagt til her er *PoolID*. Hvilket kar den hører til blir bestemt ut ifra *ToEquipment*. Ettersom det følges en konvensjon som ved navngivingen av utstyret vil et kar bestemmes ut ifra HX0Y, hvor X bestemmer hallnummer og Y bestemmer karnummer i denne hallen.

11 Sikkerhetsanalyse

Dette kapittelet tar for seg sikkerheten til hele systemet og viser hvordan de overordnede sikkerhetskravene blir innfridd. Ettersom dette er et system som kan bli satt i produksjon er det viktig at sikkerhetskravene er innfridd slik at man unngår sikkerhetsbrudd til våre system samt de eksisterende systemene dataen hentes fra. De overordnede sikkerhetskravene, hentet fra kravlisten, er som følger:

- **Krav 9**: Webapplikasjonen må være sikret med innlogging og APIet skal kreve autorisering.
- **Krav 11**: Dataen må lagres på en sikker måte, for å unngå uautorisert tilgang samt tap av data ved systemsvikt.
- **Krav 12:** Dataflyten i systemet må være sikker slik at ondsinnede aktører ikke kan se dataflyten mellom de ulike tjenestene.

Dette kapittelet starter med å vise hvordan arbeidsprosessen har tatt hensyn til sikkerhet og hvilke detaljerte sikkerhetskrav som er definert. Deretter kommer trusselmodell som gir en visuell oversikt over truslene til systemet. De etterfølgende delkapitlene vil være en risikoanalyse av systemet samt tar for seg hvordan truslene er håndtert.

11.1 Sikkerhet i utviklingsprosessen

I oppstartsfasen av det utviklede systemet har det blitt utarbeidet en plan på hvordan utviklingsprosessen skal følge SDL-Agile. Det viktigste delen går ut på å definere sikkerhetskravene og fordele de i ulike sikkerhetskravgruppene. Alle sikkerhetskravene er definert med hensyn på de overordnede sikkerhetskravene som har fremkommet i samarbeid med Tytlandsvik Aqua. For at oppgaven ikke skal bli for omfattende legges kravene til som vedlegg (16.3 Sikkerhetskrav fra Microsoft), og de enkelte kravene vil verken bli forklart hva er eller beskrevet hvorfor de er med. Siden det blir brukt flere av Microsoft sine tjenester i dette systemet er det også gunstig å følge deres anbefalinger på sikkerhetskrav. Det er valgt å jobbe ut ifra deres liste på førstekrav, bøttekrav og engangskrav/ombordstigningskrav. Noen av kravene i vedleggene er ikke like relevante med det systemet som er på plass nå, men de er fortsatt med for å sikre at de blir fulgt ved fremtidige utvidelser.

11.2 Trusselmodellering

Et sikkerhetskrav i engangsgruppen er at man skal lage en grunnleggende trussel modell. Microsofts har fem trinn de mener skal med i en trussel modell:

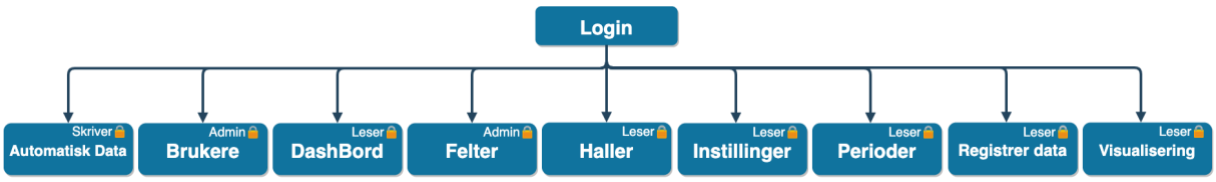
- Definere sikkerhetskrav.
- Lage et applikasjonsdiagram.
- Identifisere trusler.
- Minske trusler.
- Validere om truslene er minsket.

Det første trinnet som går ut på å definere sikkerhetskravene er definert i vedlegget 16.3 Sikkerhetskrav fra Microsoft.

Som et trinn i Microsofts guide og et punkt i OWASPs beskrivelse av trusselmodellering er det viktig å ha en oversikt over applikasjonen og hva som blir laget. En generell forståelse av hva som blir laget i dette systemet er beskrevet i detalj innledningsvis i denne oppgaven. Ulike visuelle diagrammer av applikasjonen er presentert i tidligere kapitler, men vises igjen i de neste delkapitlene med fokus på sikkerhetsaspekter.

11.2.1 Sidekart

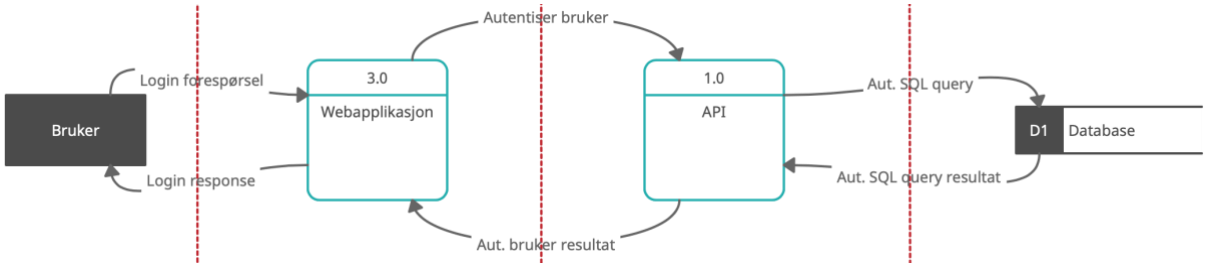
I påfølgende figur ser vi de ulike sidene i systemet en bruker kan ha tilgang på. En viktig ting å fokusere på i en sikkerhetssammenheng er hvilke roller som kreves for å være autorisert for de ulike sidene. Roller er beskrevet i kapittel 9.4.1 Roller. En mer detaljert forklaring av figuren er beskrevet i kapittel 9.3.1 Sidekart.



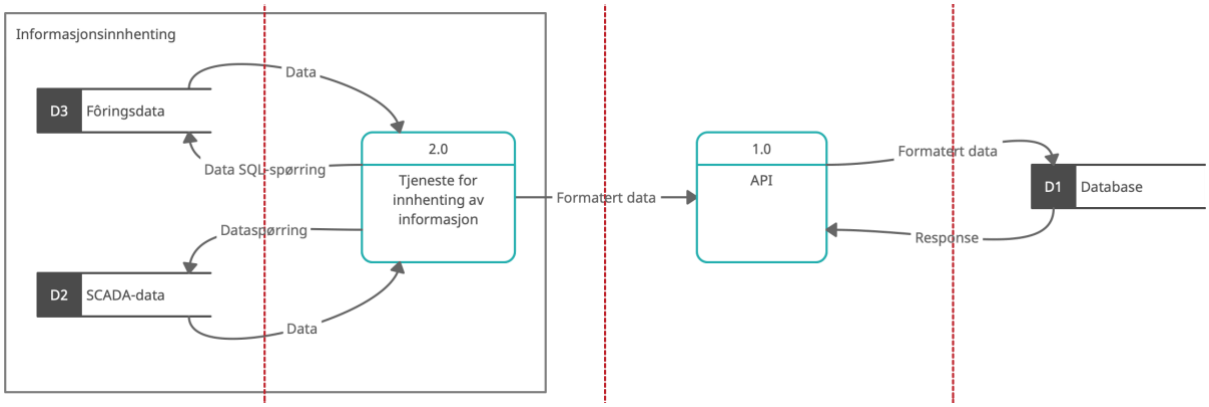
Figur 61: Sidekart

11.2.2 Dataflytdiagram

For å få ytterligere forståelse over hvilke områder av systemet som kan ha svakheter er det laget dataflytdiagrammer, som vist under:



Figur 62: Innloggingsprosess



Figur 63: Datainnhentningsprosess

Det er flere mulige sikkerhetssvakheter i en slik prosess. Det er blant annet viktig å se på svakheter i konfigureringen av grensene. En mer fyldig identifisering av truslene kommer i de neste kapitlene.

11.2.3 Metode for identifisering av trusler

Som en del av trusselmodelleringen skal man identifisere trusler til systemet som er under utvikling. Basisen gitt fra den generelle informasjonen om hva som skal lages, sidekartet og dataflytdiagrammene gjør det lettere å identifisere truslene.

Det er flere fremgangsmåter å gjøre dette på. En måte er å følge en struktur som eksempelvis STRIDE. I denne oppgaven er det ikke valgt å bruke STRIDE strukturen, men heller se på OWASP topp 10. Dette er valgt fordi det er tenkt at mange av de svakhetene som blir funnet ved å bruke en STRIDE struktur kommer til å ende opp med å bli de samme som OWASP topp 10. Ved å gå rett på svakhetene beskrevet i OWASP topp 10 sparer man tid.

11.3 Risikoanalyse

I SDL tilsier dette punkt 15: "Conduct Final Security Review". I Microsoft trussel modellering beskrivelsen tilsier dette delen som går ut på å identifisere trusler, minske trusler og validere om truslene er minsket.

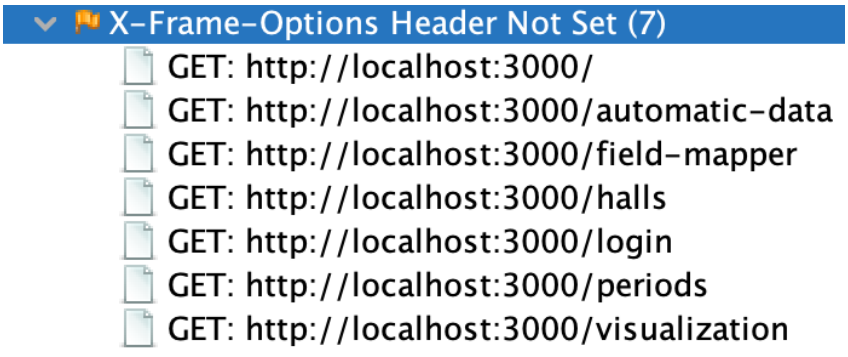
For at de identifiserte truslene skal få noen verdi må de settes opp mot de konkrete delene av systemet. Det gjøres i den samme kronologiske rekkefølgen som i OWASP. Får å raskest mulig finne disse truslene kan man bruke noen sikkerhetsverktøy. I denne oppgaven har vi valgt å bruke OWASP ZAP.

11.3.1 Analyseverktøy

På grunn av at endepunktene til APIet krever autorisering var det vanskelig å sette opp automatiske skanninger med dette programmet. ZAP har også støtte for manuelle skanninger og det var dette som ble anvendt. For å få programmet til å kjøre måtte vi skru av CORS i appen.

Ved å klikke seg gjennom løsningen skanner ZAP for svakheter. Svakhetene som blir funnet blir gradert i risikonivå. ZAP fant ingen svakheter med høy risiko, men fant tre med middels risiko samt noe informasjon. Det ble også funnet noen med lav risiko, men de vil ikke bli sett på fordi de er lite sannsynlig at inntreffer i tillegg til at de har liten innvirkning på sikkerheten.

Den første svakheten som ble funnet var følgende:

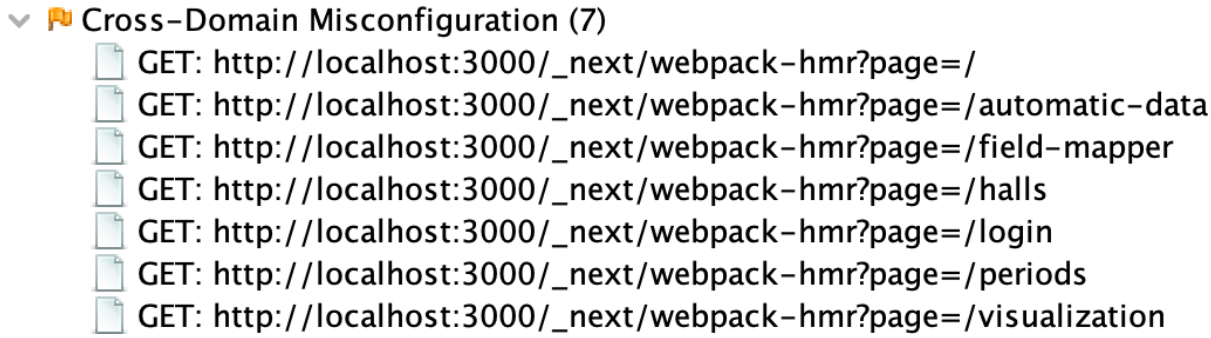


Figur 64: X-frame-optinos svakhet fra ZAP

Med beskrivelsen "*X-Frame-Options header is not included in the HTTP response to protect against ClickJacking attacks*". ClickJacking går ut på å vise en annen nettside i en ramme. Dette kan få brukeren til å tro at man er på nettsiden som vises i rammen. Angriperen kan da overvåke brukeren, man kan for eksempel se hvilke knapper brukeren trykker når man skriver

inn passordet. Angriperen kan også tvinge frem uønsket oppførsel, som for eksempel å like en side. Dette kan forhindres ved at nettsiden ikke er mulig å vises i en ramme. Dette kan man gjøre ved bruk av x-frame-options. Implementasjonen for dette i løsningen er ikke prioritert siden den ikke skal bli brukt av mange brukere. De få brukerne får tilsendt en link og så lenge denne blir brukt er ikke clickjacking noe reelt problem.

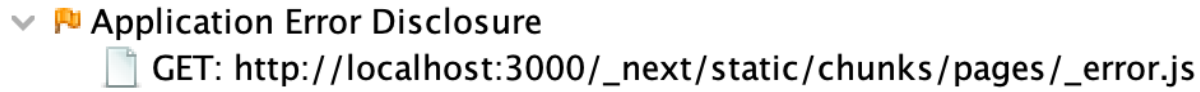
Den neste svakheten er som følger:



Figur 65: Cross-domain svakhet fra ZAP.

Den har besrkivelsen «*Web browser data loading may be possible, due to a Cross Origin Resource Sharing (CORS) misconfiguration on the web server*». Grunnen til at denne svakheten kommer er fordi CORS måtte skrus av for at ZAP skulle fungere ordentlig, og er derfor ikke et problem i produksjonsmiljøet.

Den siste svakheten med middels risiko var:



Figur 66: Application Error Discolsure svakhet fra ZAP.

Beskrivelsen var "*This page contains an error/warning message that may disclose sensitive information like the location of the file that produced the unhandled exception. This information can be used to launch further attacks against the web application. The alert could be a false positive if the error message is found inside a documentation page."* Den siste setningen i beskrivelsen er tilfellet I denne sammenhengen. _error.js er sider som følger med i next.js. Disse sidene vise frem 404 feil og 500 feil, men vil ikke vise hvor i kallstabelen feilen inntreffer så lenge man er i produksjonsmiljøet.

11.3.2 Manuell identifisering og minimering av trusler

Som nevnt tidligere brukes OWASP topp 10 for å identifisere trusler. Disse vil i dette delkapittelet gås igjennom stegvis.

1. Injeksjon

Som nevnt i teoridelen er de vanligste formene for injeksjon utført i SQL, NoSQL, OS og LDAP. Av disse er det bare SQL som er brukt i dette systemet. De aller fleste spørringene er LINQ. LINQ stopper injeksjon fordi det ikke består av strengmanipulering og konkatenering (LINQ Prevent SQL Injection, u.d.).

Det er to steder i API-et som bruker SQL kall. Det er i fôringstabellens og SCADA-tabellens filterendepunkt, og dem er ganske like. SQL-spørringen ved filterendepunktet til fôringsdataen er vist under:

```
1.  SELECT
2.      DateRecorded = Min(DateRecorded),
3.      ChunkStart = Min(DateRecorded),
4.      ChunkEnd = Max(DateRecorded),
5.      Amount = AVG([Amount])
6.      FROM
7.      (
8.           SELECT
9.           Chunk = NTILE(200) OVER(ORDER BY DateRecorded), *
10.          FROM [dbo].[FeedingLogs]
11.          // Parametere på de neste tre linjene
12.          WHERE PoolId = @p0
13.          AND DateRecorded >= @p1
14.          AND DateRecorded <= @p2
15.      ) AS T
16.
17.  GROUP BY
18.     Chunk
19.  ORDER BY
20.     ChunkStart;
```

Kodeutsnitt fra feedingLogController som henter fôringsdata ved bruk av SQL.

Tegnet *@p* på linje 12-14 er parameterne. Det er her injeksjonsangrepet kan skje. Hvis ikke parameterne blir lest på en sikker måte vil man kunne skrive om dette SQL-kallet til å gjøre noe helt annet. Man kan for eksempel slette data i databasen, eller hente data man ikke skulle hatt tilgang til.

For å kjøre dette kallet bruker vi *SqlHelper.RawSqlQuery()*. Dette er en funksjon som vi har laget selv og finnes i filen *helper/SqlHelper.cs*. Denne kjører .NETs innebygde *ExecuteReader()* med parametere. Siden vi bruker *Parameters.Add(parameter)* er dette kallet sikker for SQL-injeksjon.

2. Ødelagt autorisering

Dette handler om at angriperne får uautorisert tilgang ved å stjele brukere på en eller annen måte. I dette punktet er det flere ting så kan gå galt. Den viktigste tingen er å handtere økter på en sikker måte. I dette systemet brukes JWT til handtering av økter og kan leses mer om i kapittel 7.10.8 Sikkerhetsteknologier.

Andre ting som er brukt i dette systemet for å unngå uautorisert tilgang er beskyttelse mot brute force angrep. Dette gjør at en angriper verken kan benytte seg av ordboksangrep eller et verktøy med passordlister. Som tidligere nevnt i 10.2.11 IP-Ratebegrensing, er dette løst ved at endepunktet Auth bare kan kalles fem ganger i minuttet, noe som gjør at det vil ta veldig lang tid å gjette seg fram til riktig passord. For å illustrere med et eksempel:
Vi antar et passord må ha minst seks bokstaver og kan inneholde alle tegnene på tastaturet (rundt 100). Hvis en angriper da bare kan gjette fem passord i minuttet og treffer når det har blitt gjettet halvparten av alle mulige passord, så vil tiden det tar å gjette riktig passord kunne regnes ut slik:

$$\frac{100^6\ kombinasjoner}{(5\ ganger\ pr.min * 60\,\text{min} * 24\ timer * 365\ dager)/2} = \ 190259\ år$$

Dette viser at brute force angrep vil være svært ineffektivt ettersom man vil bruke over 190.000 år på å gjette seg frem til svaret med denne begrensingen.

For å gi en angriper minst mulig informasjon som de kan utnytte til å gjøre mer sofistikerte angrep, gis det samme tilbakemelding uansett om man har feil e-post eller passord. Dette gjør at angriperen ikke kan vite om det er e-post eller passordet som er feil, noe som gjør det vanskeligere å bryte seg inn. En annen svakhet angripere kan lete etter er standardbrukere. I dette systemet brukes det ikke standardbrukere og det er derfor ikke ett problem.

For å unngå at passordene blir for lette å gjette kan man følge passordstandarder som for eksempel *NIST 800-63 B's guidelines in section 5.1.1 for Memorized Secrets* (Paul A. Grassi,

2021). I dette systemet kreves det foreløpig bare at passordet må være lenger enn seks tegn. Mer om passord kan leses om i kapittelet 7.10.5 Passord.

3. Sensitiv data eksponering

Det er flere steder en angriper kan prøve å få tak i sensitiv data (Exposure, u.d.). Svake punkt er der data blir overført. Ved bruk av dataflytdiagrammene kan man få en oversikt over hvor disse punktene er. Her er det viktig at dataen er kryptert. I systemet brukes det HTTPS, som betyr at dataen som sendes over HTTP er kryptert. I tillegg har Azure støtte for å kryptere hele databasen, noe som er i bruk i dette systemet. Krypteringsalgoritmene baserer seg på nøkler og det er derfor viktig at disse blir handtert ordentlig. Nøklene er derfor beskyttet i Azure som miljøvariabler.

4. XML eksterne enheter (XXE)

Dette systemet bruker ikke XML og det er derfor ikke en relevant svakhet. Hvis vi istedenfor hadde brukt et SOAP API, noe som returnerer XML, kunne systemet vært utsatt for XXE (A4:2017-XML External Entities (XXE), u.d.).

5. Ødelagt tilgangskontroll

Angriperen kan utnytte at det utviklede systemet ikke har implementert tilgangskontroll ordentlig. Det kan eksempelvis være endepunkter som burde krevd autorisering som ikke gjør det (A5:2017-Broken Access Control, u.d.).

Tilgangskontrollmekanismen til APIet er skrevet ett sted og gjenbrukes overalt. I tillegg er flere av endepunktene satt opp som «*deny by default*». Dette er gjort ved å sette opp en *Authorize* dekoratør på selve kontrollerklassen, som vil si at alle endepunktene til kontrolleren krever autorisering. Innloggingsendepunktet er åpent for alle, så her er der valgt å logge alle innloggingsforsøk, noe som gir administratorer tilgang til å se potensielle angrep og derfor verne seg mot dem.

6. Feil konfigurert sikkerhet

Her er det viktig å forsikre seg om at alle miljøer er unikt konfigurerte og med forskjellige legitimasjon (A6:2017-Security Misconfiguration, u.d.). Det vil si at utviklermiljøet skal kobles opp mot utviklerdatabasen, og slik må også alle de andre miljøene fungere.

7. Skripting på tvers av nettsider (XSS)

XSS er i dette systemet relevant for frontend. Siden det er brukt React er ikke XSS mulig. React er XSS sikkert fordi strengvariabler blir automatisk rømt, slik at en angriper ikke kan kjøre skripts hos en motpart gjennom applikasjonen. I tillegg vil JSX sørge for at det er funksjoner som blir sendt som event håndterere, istedenfor strenger som potensielt har ondsinnet kode.

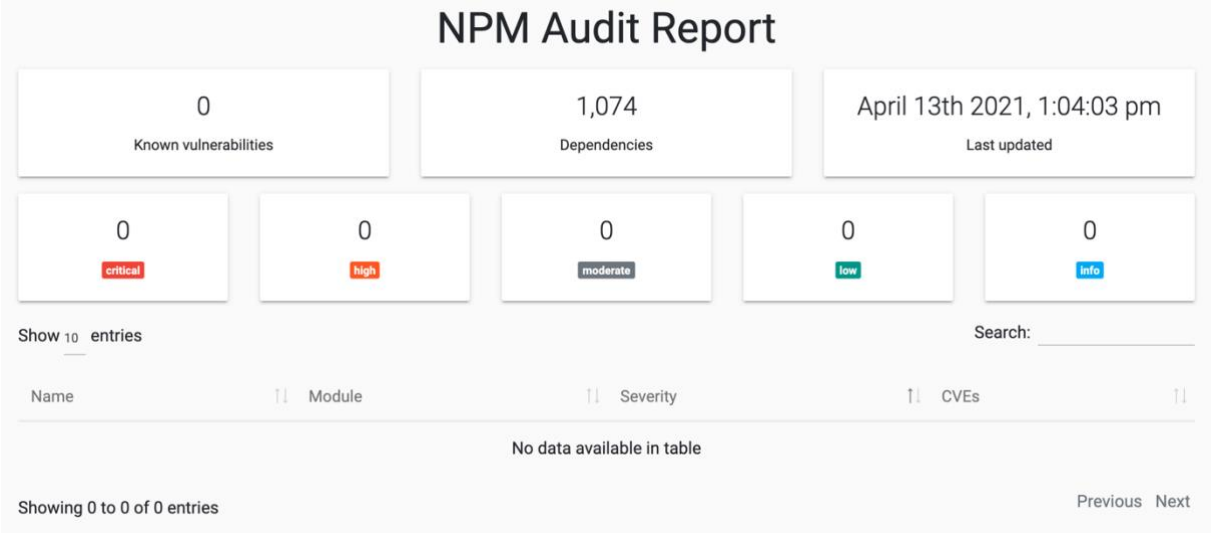
Til tross for at React er betegnet som XSS sikkert er det noen feller man kan går i. Hvis man bruker *dangerouslySetInnerHTML*, må man passe på at strengen som blir sendt inn ikke inneholder Javascript. En annen svakhet er bruk av *a.href* attributtet. Her det også kjøres Javascript-kode ved at strengen starter med Javascript. En annen måte er hvis angriperen kan selv bestemme hvilke attributter som skal være med i HTML elementene. Angriperen kan da spesifisere at for eksempel en div skal ha med *dangerouslySetInnerHTML* attributtet. Disse XSS fellene er ikke i dette systemet, og kan derfor sees bort ifra. En god huskeregel er å alltid være ekstra oppmerksom på steder brukeren kan sende inn data og hvor brukerdata blir vist.

8. Usikker deserialisering

Å utnytte deserialisering er normalt vanskelig for en angriper. Det er bare ett sted i dette systemet hvor deserialiseringsmetoden er spesifisert. Det er deserialiseringen av passord, ettersom det brukes tredjepartsbiblioteket BCrypt.net-Next. Les mer om dette i kapittel 7.10.5 Passord. Andre steder med deserialisering er ikke spesifisert direkte i kildekoden til dette systemet, men ligger i teknologiene som blir brukt. Det er for eksempel serialiseringen av databasen og serialiseringen av dataflyt som bruker HTTPS.

9. Bruke komponenter med kjente svakheter

I dette prosjektet er det prøvd å bruke velkjente pakker, med åpen kildekode som er nylige oppdaterte. Dette er gjort for å sørge for at de har minst mulig svakheter. I tillegg sørger noe som heter Dependabot for at pakker er oppdatert, som betyr at deres svakheter blir automatisk fikset. For å sjekke om pakkene i prosjektet fortsatt har noen kjente svakheter er det blitt kjørt *npm audit --json | npm-audit-html --output report.html*. Kommandoen tar resultatet fra *npm audit* i JSON format og sender dette til pakken *npm-audit-html* som generer en rapport i form av en HTML-side. Resultatet ble som vist under - ingen kjente sårbarheter:



Figur 67: NPM audit rapport med ingen kjente svakheter.

I APIet og i tjenesten for innhenting av data ble det kjørt *dotnet list package –vulnerable* for å se de kjente svakehetene, og det ble heller ikke her funnet noen kjente svakheter.

```
(base) ➜  Scheduler git:(master) ✗ dotnet list package --vulnerable

The following sources were used:
   https://api.nuget.org/v3/index.json

The given project `Scheduler` has no vulnerable packages given the current sources.
```

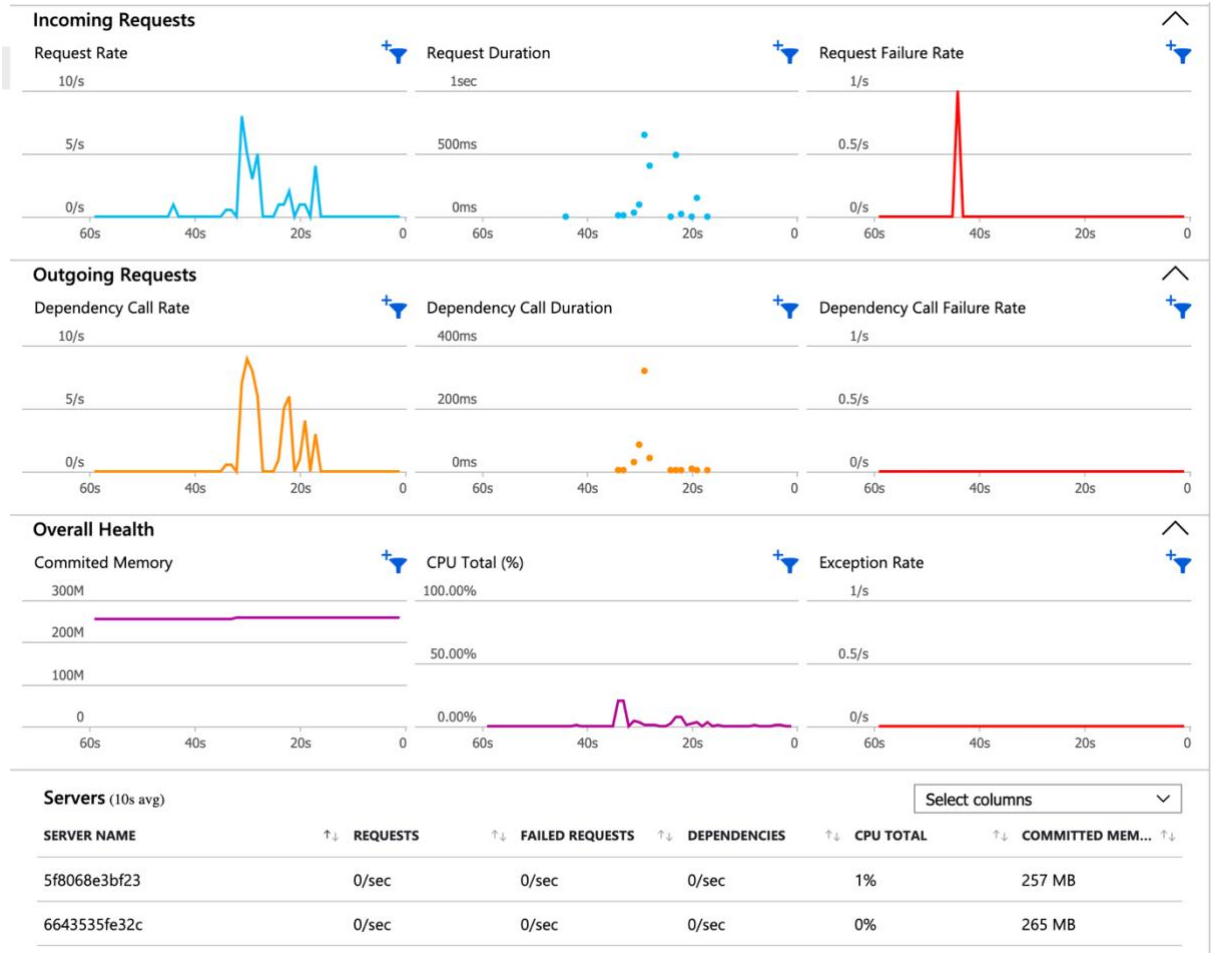
Figur 68: Kjente svakheter i APIet

```
(base) ➜  FishyAPI git:(master)dotnet list package --vulnerable

The following sources were used:
   https://api.nuget.org/v3/index.json

The given project `FishyAPI` has no vulnerable packages given the current sources.
```

Figur 69: Kjente svakheter i tjenesten for innhenting av data

10. Utilstrekkelig logging og monitorering

For at angrep skal bli oppdaget er det satt opp monitorering av systemet. Siden alle tjenestene blir kjørt i Azure følger det med masse ferdiglaget monitorering. I APIiet har vi også eksplisitt satt opp ApplicationInsights som er en tjeneste som Azure leverer. Dette er noe som monitorerer enda mer enn det som følger med som et utgangspunkt i Azure. Her får vi blant annet en oversikt over hvilke kall som feiler og blir vellykket. Her er ett utklipp fra oversikten:



Figur 70: Utklipp fra Azure ApllicatioInsights som viser blant annet kall som er vellykket og kall som feiler

12 Resultat og brukerevaluering

I systemet som er konstruert kan man måle resultatet gjennom flere ulike aspekter, eksempelvis hvordan systemet oppfyller de kravene som ble definert i starten av oppgaven. Et annet aspekt kan være hvilken verdi systemet gir Tytlandsvik Aqua som bruker. Første del av resultatet tar for seg de ulike funksjonelle og ikke-funksjonelle kravene systemet tar utgangspunkt i. Andre del vil ta for seg om systemet samstemmer med brukers forventning og den reelle verdien systemet tillegger Tytlandsvik Aqua.

12.1 Krav

Dette delkapitlet tar for seg de ulike kravene hver for seg og diskuterer hvorfor et krav er innfridd eller ikke. Avsnittene vil linke til hvor i oppgaven hoveddelen av kravet blir innfridd.

12.1.1 Funksjonelle krav

Krav 1: Webapplikasjonen skal visualisere innsamlet data på en måte som gjør at den kan brukes til å sammenligne dataene fra ulike tidsperioder og filtrere på ulike verdier.
Det er mulig å dokumentere at dette kravet er fullført ettersom visualiseringssiden gjør at man kan hente data basert på innsett og hall, samt man kan filtrere på alle parameterne anlegget lagrer i systemene sine. Videre kan man sammenligne innsett mot hverandre. Hoveddelen av kravet er innfridd i 10.1.13 Visualisering av målinger og 10.1.10 Haller og kar.

Krav 2: Systemet skal innhente målinger som blir gjort manuelt på anlegget. Dette skal brukeren enkelt kunne legge inn selv og erstatte nåværende løsning med regneark.
Dette kravet er tilfredsstilt ved modalen på manuelldata-siden som gjør det mulig å legge til, endre og slette manuell data, dette er det også lagt til rette for i APIet. Siden lister også opp dataen på en intuitiv måte. Hoveddelen av kravet er innfridd i 10.1.12 Registrering av manuell data.

Krav 3: Webapplikasjonen skal presentere sanntidsdata som blir hentet fra anlegget i en graf som kontinuerlig oppdateres.
Kravet blir tilfredsstilt ved å benytte sanntidsutsending av data ved hjelp av SignalR. Sanntidsdataen blir vist på dashboardsiden, og blir oppdatert når det legges til nye verdier fra SCADA- og fôringssystemet. Hoveddelen av kravet er innfridd i 10.1.9 Dashbord og 10.2.7 Sanntidsutsending av data.

Krav 4: Systemet skal automatisk formatere og lagre data fra SCADA- og fôringssystemet i en egen database.

Ettersom det var litt problemer med systemene på anlegget ble ikke dette kravet helt fullført, men det er lagt til rette for at det skal være så lite arbeid som mulig å faktisk koble seg opp. Tjenesten for innhenting av data er satt opp, skript for prosessering av logg-data fra både fôrings- og SCADA-systemet er verifisert fungerende ved hjelp av testing på eksisterende data. Det som mangler her, er å koble seg opp mot databasen til foringssystemet og hente loggfiler fra en FTP-server til SCADA-systemet. En midlertidig løsning er lagd for SCADA-loggene ved at man kan laste opp loggfiler fra Webapplikasjonen. Hoveddelen av kravet er innfridd i 10.2.6 Kontrollere, 10.3 Tjeneste for innhenting av informasjon, 10.5 Databehandling og 10.1.14 Automatisk data.

Krav 5: Systemet må gjøre det mulig å spore fisken tilbake til hvilke kar de har vært i.

Kravet er innfridd ved å legge til rette for å spesifisere innsett per kar, samt kunne flytte og splitte kar ut til flere kar. Dette gjør at man kan spore fiskens tilstedeværelse i ulike kar gjennom et innsett. Hoveddelen av kravet er innfridd i 10.1.11 Innsett og tidslinje og 10.1.10 Haller og kar.

Krav 6: Systemet skal eksponere dataene slik at eksterne systemer enkelt kan konsumere dataene.

Dette kravet er blitt løst ved å strukturere loggdataen i tabeller i en SQL-database, på denne måten er strukturen fast samtidig som dataen har blitt gjort enklere å bearbeide. Et godt eksempel på dette er omstruktureringen av loggfilene fra SCADA-systemet som tidligere hadde en rad per sensor mens disse nå er blitt samlet til en rad og videre sortert på tid. Ved å eksponere dataen gjennom et REST API sikrer man at dataen kan gjenbrukes av andre klienter. Hoveddelen av kravet er innfridd i 9.4.2 Endepunkter, 10.5 Databehandling og deler av 9.5 Database ER-Diagram.

Krav 7: Systemet skal kunne brukes av flere brukere med ulike sikkerhetsklareringer, altså skille på lese-, skrive- og administratorrettigheter.

Kravet er i varetatt ved å implementere rollehåndtering i APIet. Ved å implementere disse begrensingene i APIet sikrer man at rollehåndteringen også gjelder eksterne systemer. Hoveddelen av kravet er innfridd i 9.4.1 Roller og 10.2.4 Roller.

Krav 8: En bruker med administratorrettigheter skal kunne administrere alle brukerne i webapplikasjonen.

Ved å eksponere endepunkt for å manipulere brukere dersom man har rollen *Admin* er dette kravet fulgt opp, en side i Webapplikasjonen har også gjort det enkelt for sluttbruker å administrere brukerne. Hoveddelen av kravet er innfridd i 10.1.15 Brukere og 10.2.6 Kontrollere.

Krav 9: Webapplikasjonen må være sikret med innlogging og APIet skal kreve autorisering.

Implementasjonen av JWT autentisering og autorisering samt en side for innlogging i webapplikasjonen imøtekommer dette kravet. Hoveddelen av kravet er innfridd i 10.1.7 Autentisering, 10.2.4 Roller og 10.2.3 JWT.

12.1.2 Ikke-funksjonelle krav

Krav 10: Systemet skal kunne skalere for å håndtere økt brukeraktivitet og større datamengder, samt ha minimale driftskostnader.

Ved å benytte beholdere sammen med Azure AppServices sikrer man at man kan skalere vertikalt og horisontalt. Å skalere vertikalt betyr å øke kapasiteten til en server, når man derimot skalerer horisontalt legger man til nye serverinstanser. Også ved å benytte seg av caching på APIet sikrer man at spørringer til database skjer unødvendig, noe som vil hjelpe på skaleringen ved økt trafikk. Ved å kjøre løsningen som en skytjeneste kan man også minimere driftskostnadene til systemet da man ikke behøver å benytte servere som krever vedlikehold og oppdateringer. Hoveddelen av kravet er argumentert for i 7.3 Mikrotjenester og 7.5 Skytjenester.

Krav 11: Dataen må lagres på en sikker måte, for å unngå uautorisert tilgang samt tap av data ved systemsvikt.

Ved å minimere tilgangen til dataen til å kun skje gjennom APIet sikrer man at det kun er autorisert tilgang. Videre er det satt opp at AzureSQL serveren kun kan snakke med APIet og spesifiserte ip-addresser som er brukt under utviklingen. Sikkerhetsfunksjonaliteten som er spesifisert tidligere i oppgaven som eksempelvis regelmessig sikkerhetskopiering er med på å sikre at tap av data blir så å si umulig. Hoveddelen av dette kravet er innfridd i 7.5.2 Azure, 10.2.3 JWT og 10.2.4 Roller.

Krav 12: Dataflyten i systemet må være sikker slik at ondsinnede aktører ikke kan se dataflyten mellom de ulike tjenestene.

For at flyten skal være sikker så er APIet, Webapplikasjonen, databasen og innhentingstjenesten sikret med blant annet autorisering og logging. Kommunikasjonen mellom alle disse er også begrenset til bare det som er nødvendig i tillegg til at den er kryptert med TLS. Innfrielsen av dette kravet er sterkt knyttet til 11 Sikkerhetsanalyse.

Krav 13: Systemet må legge til rette for videre utvikling.

Dette er et krav som kan være vrient å oppfylle, ettersom det er vanskelig å vite hvilken utvikling man skal legge til rette for videre. Systemet er nå lagd på en slik måte at ny funksjonalitet, tilsvarende det som er lagd til nå, enkelt kan tillegges, dvs. endepunkter for å manipulere og hente ut data. Teknologien som er valgt legger også til rette for videreutvikling av eksterne ressurser ettersom det er valgt å bruke anerkjente teknologier som er mye brukt, samt det er kjente aktører som vedlikeholder valgte rammeverk. Automatisk bygging og utrullingsflyten legger også til rette for videreutvikling. Deler av kravet er innfridd i 10.4 Automatisk bygging og utrulling.

Krav 14: Webapplikasjonen skal være brukervennlig.

Det er valgt å benytte pakker som eksempelvis Material-UI som benytter Google sin designstandard for komponenter og interaksjoner for å sikre at webapplikasjonen blir anvendelig for ikke-tekniske brukere. Noe av oppfyllelsen til dette kravet vil bli sett nærmere på under brukerevalueringen.

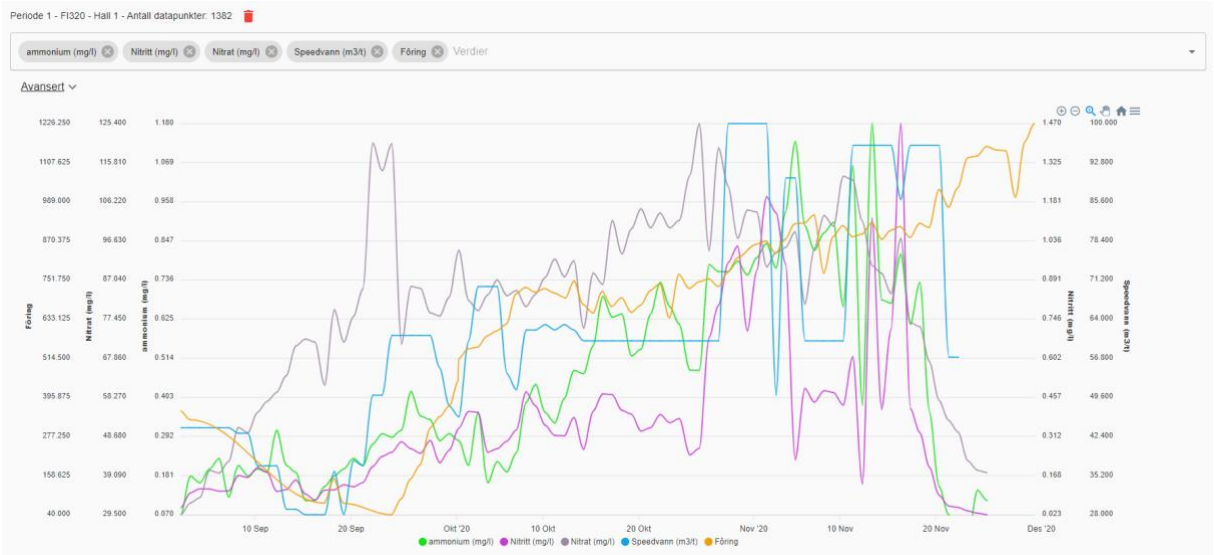
12.2 Brukerevaluering

Målet med oppgaven var også å undersøke om systemet som er utviklet har en bruksverdi i driften av anlegget. Selv om de funksjonelle og ikke-funksjonelle kravene til systemet er oppfylt, er ikke dette synonymt med at systemet har en verdi for anlegget. Dette er vanskelig å si noe om som utviklere av systemet, det krever resultat fra fagansvarlig med erfaring på det spesifikke område på anlegget.

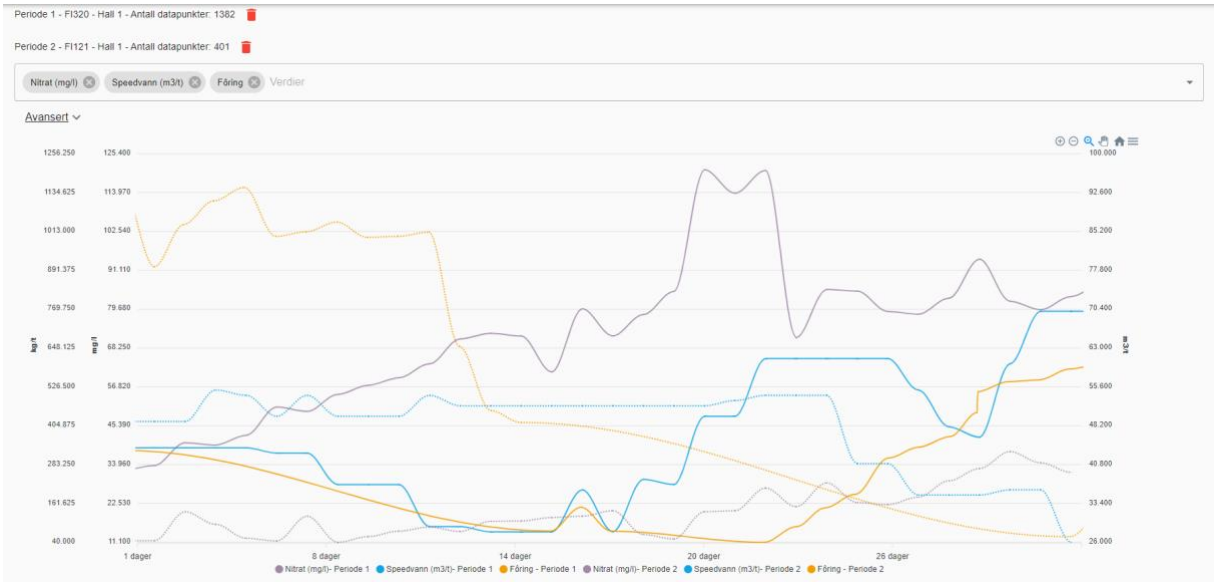
En brukerevaluering gir derfor et grundigere grunnlag til å konkludere om kravene er innfridd. Den biologisk ansvarlige hos Tytlandsvik Aqua fikk i oppgave å komme med et eksempel på hvordan systemet vil bi brukt. Ettersom hun hadde tilgang på testmiljøet, var det

mulig å praktisk gjennomføre dette eksempelet. Eksempelet med forklaring ble presentert som følgende:

> *Et RAS-anlegg er et anlegg med både tekniske og biologiske renseprosesser, som overvåkes med mye automatikk og styring. Ved muligheten til å hente inne alt av tilgjengelig informasjon, sette det i system og sammenligne data, vil det bli enklere å optimalisere driften og utvikle gode rutiner. Både med tanke på de tekniske og biologiske forutsetningene.*
>
> *De biologiske renseprosessene består for eksempel av bakterier til å rense vannet for ammonium (noe av fiskens avfallsstoffer) der nitrat er sluttproduktet, via nitritt. For høye konsentrasjoner vil i verste fall være dødelig for fisken, og man er avhengig av å tynne det ut med nytt vann inn. En økende produksjon gjennom et innsett vil øke presset på renseprosessen. Det å kunne sette disse verdiene opp mot hverandre visuelt og sammenligne det med den tekniske driften av anlegget, vil det være et nyttig verktøy i produksjonen.*



Figur 71: Brukereksempel (1) fra biologisk ansvarlig.



Figur 72: Brukereksempel (2) fra biologisk ansvarlig.

> *Gjennom et godt samarbeid der ønsker og innspill er imøtekommet og godt ivaretatt, tilfredsstiller resultatet våre forventninger. All den dataen som logges kan enkelt linkes til de forskjellige produksjonene, hallene og karene som finnes i anlegget noe som gjør det brukervennlig. Det er ikke tvil om at dette er et godt utgangspunkt, med mye fremtidig potensial.* (Jenssen, 2021)

Man kan ut ifra dette kunne si at systemet fungerer etter sin hensikt og kan anvendes av det personell det er ment for.

13 Diskusjon og videre utvikling

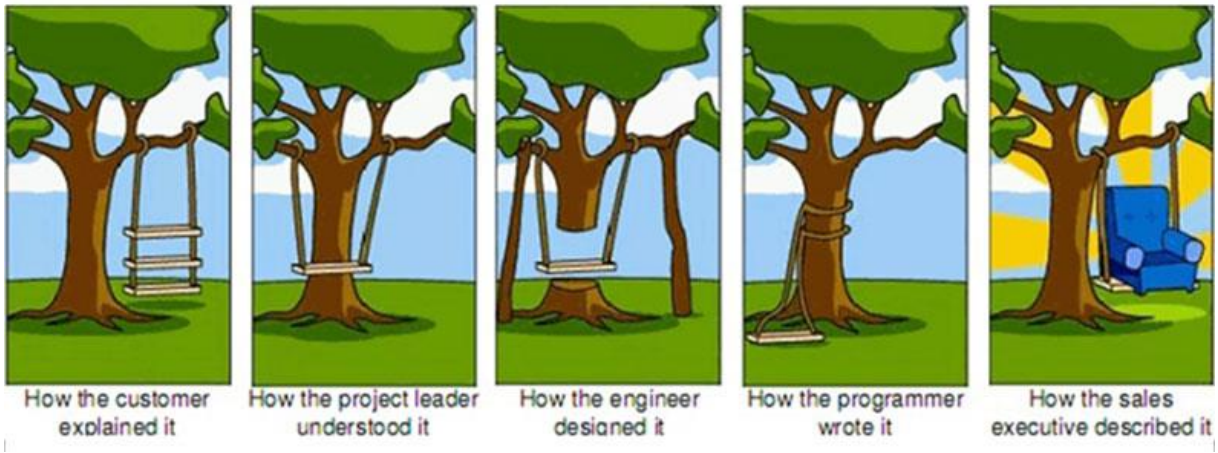
Dette kapitlet tar for seg hva som har fungert bra og ikke, samt hvilke endringer vi hadde ønsket å gjøre dersom vi skulle gjort prosjektet om igjen. Det vil også bli gjennomgått hva som må til for å koble tjenesten for innhenting av informasjon til SCADA- og fôringssystemet, ettersom dette ikke kunne ferdigstilles i løpet av prosjektets tidsperiode.

13.1 Prosjektevaluering

I denne delen blir det diskutert rundt hva som kunne vært gjort annerledes og om noe fungerte bra eller dårlig, samt hva vi har lært.

Arbeidsmetodikk

Grunnet koronasituasjonen og den lange avstanden til anlegget foregikk kommunikasjonen i hovedsak over e-post eller i nettmøter. Det ble gjennomført tre møter på anlegget, som var veldig givende for forståelsen av hva systemet skulle løse. I en best mulig situasjon hadde det vært ønskelig med flere slike møter for blant annet å ha en enda klarere kommunikasjon om kravene. Kravinnhentingen var noe som har foregått dynamisk over hele utviklingsperioden av systemet. Dette er noe som har fungert godt i dette prosjektet, ettersom vi har hatt tett kommunikasjon gjennom hele prosjektløpet samt hovedproblemet som skulle løses ble fastsatt tidlig. Hadde vi ikke hatt dette gode samarbeidet ville nok en fullstendig kravliste på forhånd vært mer fordelaktig.



Figur 73: Dette bildet illustrerer hvor viktig det er med god kommunikasjon i en kravinnhentingsprosess og videreformidling av disse kravene.

Hentet fra (Fleming, 2014)

Fremdriftsplanen, som beskrevet i 8.1 Fremdriftsplan, var behjelpelig for å få en oversikt over hvor lang tid vi antok det ville ta å utvikle de ulike delene av systemet. I selve utviklingsprosessen var fasene mer oppstykket og overlappende. Eksempelvis måtte vi gjøre små ender på visualiseringen etter at vi hadde fått hentet ut produksjonsdataene.

Som beskrevet i 8 Arbeidsmetodikk ble det fulgt en scrum arbeidsmetodikk. Utviklingen ble derfor fordelt i små oppgaver som skulle løses i sprinter. Optimalt skal disse oppgaven tydelig merkes med hvem som sitter med oppgaven og i hvilken tilstand den er i. Dette ble fulgt i starten av utviklingen, men ble mindre fulgt etter hvert. Det var fordi vi satt mye sammen og vi bare er en gruppe på tre, og alle hadde god oversikt over statusene på de forskjellige oppgavene til enhver tid. Det ble derfor overfladisk å dokumentere det.

Arkitektur

Tjenestene har hvert sitt overordnede ansvarsområde, noe som har gjort systemet fleksibelt for å imøtekomme diverse krav. Strukturen av databasemodellene oppleves oversiktlig og intuitivt å jobbe med, da flere av databasemodellene er hentet rett ut fra domenet i tillegg til å følge normaliseringsprinsippene. Modellene er altså konstruert direkte ut ifra hvordan ting er i virkeligheten. De fleste arkitekturvalg er basert på standarder vi har plukket opp gjennom studie og arbeidsliv, noe som kan være grunnen til at det har fungert godt.

Det opplevdes også fordelaktig å få en god oversikt over dataflyten ved å lage noen enkle skisser. Vi kunne da diskutere mye om hvordan løsningen skulle ende opp før vi startet på selve implementasjonen, noe som gjorde at vi ikke behøvde å implementere samme funksjonalitet i flere omganger. Selve designet på sidene ble gjort under utvikling, da det ga mer frihet, mens ideelt sett kunne dette vært mer spesifisert på forhånd. Dette ville da gitt et enda mer gjennomtenkt brukergrensesnitt med tanke på bedre brukervennlighet og særpreg.

Dersom prosjektet skulle gjennomføres på nytt hadde det vært spennende å undersøke muligheten for å benytte en løsning der enda mer funksjonalitet som kommer rett ut av boksen. Eksempel på dette kunne vært å benytte en såkalt serverløs arkitektur, hvor all funksjonalitet er skrevet som separate funksjoner og kan gjør løsningen mer hendelsesbasert. En fordel med dette er at skalering og hosting kommer ut av boksen. Et verktøy som hadde gitt mye god serverløs funksjonalitet er eksempelvis Amazone Web Services (AWS) sitt rammeverk Amplify.

Teknologivalg

Det ble benyttet teknologi som var noe kjent for de fleste i gruppen. Ettersom prosjektet var omfattende, ble dette viktig for å kunne ferdigstille systemet innenfor det gitte tidsrommet.

Når det kommer til API hadde det nok vært lurt, med tanke på videre utvikling, å undersøke GraphQL ettersom dette ville effektivisert APIet ved at man da kun henter den dataen man trenger der og da. Det finnes også mange rammeverk her som automatisk generer spørringer og typer som kan benyttes i klientapplikasjonen slik at man får mindre implementasjonsfeil og benytter mindre tid på kode som i stor grad vil være lik.

Det hadde også vært en idé å bytte ut SQL databasen med en dokumentdatabase for å tilrettelegge for en mer dynamisk datastruktur. Ettersom Tytlandsvik Aqua til tider får nye sensorer krever det at SQL databasen manuelt må vedlikeholdes. Ved bruk av dokumentdatabase vil datastrukturen bli mer dynamiske, men det kan medføre andre problemer som er årsaken til at dette ikke ble valgt i denne omgangen. Relasjoner vil for eksempel bli mer avansert å jobbe med, i tillegg har ikke Entity Framework støtte for å migrere dokumentdatabaser.

Rammeverkene som har blitt benyttet har god funksjonalitet og dokumentasjon. Frontend-rammeverket NEXT er vært enkelt å sette opp og baserer seg på React som gjør det mulig å benytte komponent-biblioteket Material-UI, noe som viste seg å være lurt i dette prosjektet slik at det var mulig å fokusere mer funksjonalitet fremfor å bruke tid på å implementere designet fra bunn av. .NET var litt knotete å komme i gang med ettersom det finnes mange ulike versjoner som til tider gjorde det komplisert å finne riktig dokumentasjon, men når man kom inn i dette universet var dette enklere. Det kan sees på som litt omfattende å benytte to ulike programmeringsspråk, da man heller kunne valgt enten .NET eller JavaScript baserte rammeverk til alle tjenestene uten at man hadde fått et annerledes produkt.

Implementasjon

Det kunne vært fordelaktig å skrive enhets- og integrasjonstester slik at man hadde fått en mer robust kodebase som gir beskjed dersom endringer ødelegger funksjonaliteten. Man kunne eksempelvis benyttet testdreven utvikling der man først skriver testene og så skriver kode som tilfredsstiller testene. Dette ble valgt bort til fordel for utvidet funksjonalitet, ettersom det å skrive tester er tidkrevende.

Læringsutbytte

Noe av formålet med teknologivalgene var blant annet å ha fokus på læringsutbytte. Alle hadde kjennskap til teknologiene som ble brukt, mens kunnskapsnivået på dem var spredt i gruppen. Dette gjorde at den som hadde god kontroll på en teknologi kunne sette andre i riktig retning, noe som har fungert godt. Ved å sette oss inn i nye teknologier har vi også fått anvendt allerede kjente praksiser på en ny måte, noe som har gitt en fyldigere forståelse for faget.

Det har også vært lærerikt å jobbe med en reell bedrift, der vi selv sto for all kommunikasjonen og hadde hovedansvaret for løsningen. Dette har krevd at vi spesifiserer hele løsningen selv, som arkitektur- og teknologivalg, noe som har vært nytt for oss. Det har gjort at vi har blitt mer trygge på våre tekniske beslutninger, da vi mener dem har fungert bra i dette prosjektet.

13.2 Produksjonssetting

Grunnet forsinkelser og tekniske utfordringer på anlegget, samt endringer som hadde prioritet over implementasjonen av dette systemet, ble ikke systemet implementert med automatisk innhenting av data. For å fullføre integrasjonen med automatisk innhenting av data må det lokale nettverket åpne adresse og port til FTP-server for SCADA-logger og fôringssystemets database. Videre må metodene i tjenesten for automatisk innhenting av data skrives litt om. SCADA-systemets metode må hente de siste filene og laste opp til APIet. Fôringsystemets metode vil laste opp innholdet fra fôringsdatabasen nærmes uten endringer. Dette er to steg som ikke skal være omfattende å gjennomføre da Microsoft eksempelvis har et eget bibliotek for FTP-klienter i .NET. Videre er samtlige miljøer satt opp i Azure for videre bruk.

På grunn av at denne delen ikke er blitt implementert har heller ikke Tytlandsvik Aqua fått gjennomført utvidet testing. De har altså ikke fått testet det nye systemet som en erstatning for det systemet de har i dag. Dette har ført til en forsinkelse på konkrete tilbakemeldinger om hva som fungerer bra og ikke. Disse konkrete tilbakemeldingene er nødvendig for å lage en skreddersydd brukeropplevelse av analysen og løsningen i sin helhet. Det er derfor forventet at det kan oppstå et ønske om endringer etter hvert som Tytlandsvik Aqua får testet mer.

13.3 Videre utvikling og forbedringer

Dette delkapittelet tar for seg forbedringsområder og mulige områder for videreutvikling. Et område som har forbedringspotensialer, er optimalisering av spørringer. Det er valgt å ikke fokusere på dette i denne omgang ettersom det ikke gir verdifull funksjonalitet for sluttbrukeren annen enn å spare noen sekunder på lasting av data. Det kunne også vært fordelaktig å benytte mer tid på designet slik at brukerflyten blir mer brukervennlig, dette er et område hvor man mangler kunnskap og som med fordel kunne vært gjort under veiledning med en fagperson på området.

Når det kommer til videre utvikling vil det helt åpenbare være å koble seg skikkelig på SCADA- og fôringssystemet når dette blir åpnet for. Det hadde også vært interessant å sett på filtrering av ekstremverdier, ved at f.eks. brukerne selv kan sette verdiene som skal filtreres bort på de ulike nøklene. Det nåværende systemet tar vare på all dataen, men det kunne også vært nyttig å undersøke hvor mye av denne dataen som trengs for at dataen skal være nyttig for brukeren, en måling i timen, to eller 50? Videre er det også nevnt av Tytlandsvik Aqua at det kunne vært interessert å koble seg på enda flere systemer for å få et enda større datagrunnlag. Det er også vist interesse for å legge til muligheter for statistiske analyser, prediksjon og å generere rapporter, noe som kunne økt bruksområdet ytterligere. Det er altså svært mange muligheter for videre utvikling for systemet.

14 Figurliste

15 Referanser

15.1 Personer

Eli Birghite Jenssen – Biologisk Ansvarlig – Tytlandsvik Aqua

Jone Sedberg – Teknisk Leder – Tytlandsvik Aqua

15.2 Skriftlig

Kildekritikk

Det er valgt å benytte seg av dokumentasjonen til de ulike teknologien så godt det lar seg gjøre, dette er fordi disse ofte innehar den mest oppdaterte dokumentasjonen og bruk i forhold til beste praksis.

(u.d.). Hentet fra Fusion Charts Docs: https://www.fusioncharts.com/dev/fusioncharts

A5:2017-Broken Access Control. (u.d.). Hentet fra OWASP: https://owasp.org/www-project-top-ten/2017/A5_2017-Broken_Access_Control

Ali, M. F. (2017, Desember 2). *Data Annotations*. Hentet fra Medium: https://medium.com/@mirzafarrukh13/data-annotations-36acbee715a8

Altexsoft. (2018, Mai 29). *Functional and Nonfunctional Requirements: Specification and Types*. Hentet fra Altexsoft: https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/

Anderson, R., scottaddie, intrepion, guardrex, tdykstra, v-thepet, . . . danroth27. (2020, 11 13). *Tutorial: Part 5, apply migrations to the Contoso University sample*. Hentet fra Microsoft docs: https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/migrations?view=aspnetcore-5.0

Appendix Q: SDL-Agile Bucket Requirements. (2012, Mai 22). Hentet fra Microsoft Docs: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ee790611(v=msdn.10)

Appendix R: SDL-Agile One-Time Requirements . (2012, Mai 22). Hentet fra Microsoft Docs: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ee790612(v=msdn.10)

Application Threat Modeling. (u.d.). Hentet fra OWASP: https://owasp.org/www-community/Application_Threat_Modeling

Authentication. (2018, Juli 13). Hentet fra TechTerms: https://techterms.com/definition/authentication

AWS. (2021, mars 3). *What Is a Document Database?* . Hentet fra AWS: https://aws.amazon.com/nosql/document/

A2:2017-Broken Authentication. (u.d.). Hentet fra OWASP: https://owasp.org/www-project-top-ten/2017/A2_2017-Broken_Authentication

A4:2017-XML External Entities (XXE). (u.d.). Hentet fra OWASP: https://owasp.org/www-project-top-ten/2017/A4_2017-XML_External_Entities_(XXE)

A6:2017-Security Misconfiguration. (u.d.). Hentet fra OWASP: https://owasp.org/www-project-top-ten/2017/A6_2017-Security_Misconfiguration

A7:2017-Cross-Site Scripting (XSS). (u.d.). Hentet fra OWASP: https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS)

BillWanger. (2018, 26 4). *Attributes (C#)*. Hentet fra docs.microsoft: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/
Broecklmann, R. (2017, Julay 19). *medium*. Hentet fra SAML2 vs JWT: A Comparison: https://medium.com/@robert.broeckelmann/saml2-vs-jwt-a-comparison-254bafd98e6
CAPTCHA. (2019, Juni 30). Hentet fra Wikipedia: https://no.wikipedia.org/wiki/CAPTCHA
ChrisMcKee. (2020, 12). *BCrypt.Net-Next*. Hentet fra Nuget.org: https://www.nuget.org/packages/BCrypt.Net-Next/
Cisco Networking Academy. (2013). *Switching, Routing, and Wireless Essentials Companion Guide (CCNAv7).* Cisco Press.
CodeCademy. (2021, mars 12). *What is a Relational Database Management System?* Hentet fra CodeCademy: https://www.codecademy.com/articles/what-is-rdbms-sql
Damtoft, E. (2019, 3 29). *Storing Passwords in .NET Core*. Hentet fra medium: https://medium.com/dealeron-dev/storing-passwords-in-net-core-3de29a3da4d2
Database Model. (2017, Januar 19). Hentet fra Techopedia: https://www.techopedia.com/definition/6762/database-model
Datatilsynet. (2018, 06 23). *Datatilsynet - Skytjenester*. Hentet fra https://www.datatilsynet.no/personvern-pa-ulike-omrader/internett-og-apper/skytjenester/
Debugger. (u.d.). Hentet fra JWT.
Docker Inc. (2021, 02 09). *Docker Documentation*. Hentet fra What is a cointainer?: https://www.docker.com/resources/what-container
Docker Inc. (2021, 02 09). *Dockerfile Reference*. Hentet fra Docker: https://docs.docker.com/engine/reference/builder/
Docker Inc. (u.d.). *Docker Docs - Engine*. Hentet fra https://docs.docker.com/engine/
Exposure, A.-S. D. (u.d.). Hentet fra OWASP: https://owasp.org/www-project-top-ten/2017/A3_2017-Sensitive_Data_Exposure
Fleming, P. (2014, August 19). *Mind the Gap: Addressing Ambiguity in Requirements* . Hentet fra Lansa: https://lansa.com/blog/rapid-application-development/ambiguity-in-requirements/
Fourth Normal Form (4NF). (u.d.). Hentet fra studytonight: https://www.studytonight.com/dbms/fourth-normal-form.php
Galloway, J. (2011, 4 21). *Part 6: Using Data Annotations for Model Validation*. Hentet fra Microsoft docs: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/mvc-music-store/mvc-music-store-part-6
Getting Started. (u.d.). Hentet fra ZAProxy: https://www.zaproxy.org/getting-started/
Git. (2021, 02 09). *Git Documentations*. Hentet fra Git: https://git-scm.com/doc
Git. (21, 2 08). *https://git-scm.com*. Hentet fra https://git-scm.com/docs: https://git-scm.com/docs/githooks
Github Inc. (2021, 02 09). *About GitHub Container Registry* . Hentet fra Github : https://docs.github.com/en/packages/guides/about-github-container-registry
GitHub Inc. (2021, 02 09). *GitHub*. Hentet fra Github Documentation: https://docs.github.com/en
Github Inc. (2021, 02 09). *GitHub Documentation*. Hentet fra Github: https://docs.github.com/en
Hangfire. (2021, 02 09). *Hangfire*. Hentet fra Hangfire Docs: https://www.hangfire.io/
Howard, Michael; Steve Lipner. (2006). *The Security Development Lifecycle.* Washington: Microsoft Press.
HTTP headers. (u.d.). Hentet fra Developer Mozilla: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

Hva er Kanban? (2019, Juni 6). Hentet fra Prosjektbloggen: https://www.prosjektbloggen.no/hva-er-kanban
Jenssen, E. B. (2021, Januar). Samtale med Eli. Tøtlandsvik.
json. (u.d.). *https://www.json.org.* Hentet fra https://www.json.org: https://www.json.org/json-en.html
julieMSFT. (2017, Mars 16). *NTILE (Transact-SQL)*. Hentet fra Microsoft docs: https://docs.microsoft.com/en-us/sql/t-sql/functions/ntile-transact-sql?view=sql-server-ver15
Lahar, S. (12, 16 2020). *What Is a CSV File? Guide to Uses and Benefits*. Hentet fra FlatFile: https://flatfile.io/blog/what-is-a-csv-file-guide-to-uses-and-benefits
Liew, Z. (2018, Januar 17). *Understanding And Using REST APIs*. Hentet fra Smashingmagazine: https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/
LINQ Prevent SQL Injection. (u.d.). Hentet fra Entity Framework: https://entityframework.net/linq-prevent-sql-injection
Material UI. (2021, Feb 15). *https://material-ui.com*. Hentet fra https://material-ui.com: https://material-ui.com/getting-started/installation/
Microsoft Azure. (2021, 02 09). *Microsoft Azure Appservices*. Hentet fra Microsoft Azure: https://azure.microsoft.com/nb-no/services/app-service/
Microsoft Azure. (2021, 02 09). *Microsoft Azure SQL Database*. Hentet fra Microsoft Azure: https://azure.microsoft.com/nb-no/services/sql-database
Microsoft Azure. (2021, 02 09). *Mircrosoft Azure SignalR*. Hentet fra Microsoft Azure: https://dotnet.microsoft.com/apps/aspnet/signalr/service
Microsoft. (2011, 2 21). *How to: Validate Model Data Using DataAnnotations Attributes*. Hentet fra Microsoft docs: https://docs.microsoft.com/en-us/previous-versions/aspnet/ee256141(v=vs.100)?redirectedfrom=MSDN
Microsoft. (2011, 2 21). *How to: Validate Model Data Using DataAnnotations Attributes*. Hentet fra Microsoft docs: https://docs.microsoft.com/en-us/previous-versions/aspnet/ee256141(v=vs.100)?redirectedfrom=MSDN
microsoft. (2012, Mai 22). *Appendix P - SDL-Agile Every-Sprint Requirements*. Hentet fra Microsoft Docs: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ee790610(v=msdn.10)
Microsoft. (2017, 5 24). *Microsoft docs*. Hentet fra Caching: https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching
microsoft. (2021, 03 23). *https://docs.microsoft.com/en-us/azure/azure-sql/database.* Hentet fra https://docs.microsoft.com: https://docs.microsoft.com/en-us/azure/azure-sql/database/automatic-tuning-overview
Microsoft. (2021, 04 05). *Service tiers in the DU-based purchase model*. Hentet fra Microsoft Docs: https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tiers-dtu
Moor, T. D. (u.d.). *https://x-team.com/*. Hentet fra https://x-team.com/blog/8-best-and-most-popular-react-libraries-in-2019/: https://x-team.com/blog/8-best-and-most-popular-react-libraries-in-2019/
Multi-factor authentication. (u.d.). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Multi-factor_authentication
neo4j. (2021, Mars 12). *What is a Graph Database?* Hentet fra Neo4j: https://neo4j.com/developer/graph-database/
Nes, S. M. (2019, mai 16). *Visma*. Hentet fra Visma.no/blogg: https://www.visma.no/blogg/en-kort-introduksjon-til-scrum/
Normalization of Database. (u.d.). Hentet fra studytonight: https://www.studytonight.com/dbms/first-normal-form.php

Norsk Røyeforum. (2021, Mars 4). *RAS*. Hentet fra Norsk Røyeforum: https://royeforum.no/oppdrett/ras/
Osman, M. (2020, 12 6). *10 Ways Website Structure Can Affect SEO*. Hentet fra searchenginejournal: https://www.searchenginejournal.com/how-website-structure-affects-seo/387034/#close
OWASP Foundation, I. (u.d.). *OWASP Top Ten*. Hentet fra OWASP.org: https://owasp.org/www-project-top-ten/
Patel, P. (2018, 04 18). *freecodecamp.* Hentet fra https://www.freecodecamp.org: https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/
Paul A. Grassi, J. L.-Y. (2021, Mai 11). *NIST*. Hentet fra NIST Special Publication 800-63B: https://pages.nist.gov/800-63-3/sp800-63b.html#memsecret
Peyrott, S. E. (Version 0.14.1, 2016-2018). The JWT Handbook - Security Considerations. I S. E. Peyrott, *The JWT Handbook* (ss. 9-13). Auth0 Inc.
React.js. (u.d.). *https://reactjs.org/.* Hentet fra https://reactjs.org/docs/: https://reactjs.org/docs/getting-started.html
React.js. (u.d.). *reactjs.* Hentet fra ntroducing-jsx: https://reactjs.org/docs/introducing-jsx.html
React.js. (u.d.). *reactjs.* Hentet fra https://reactjs.org/docs/: https://reactjs.org/docs/components-and-props.html
React.js. (u.d.). *reactjs.* Hentet fra https://reactjs.org/docs/: https://reactjs.org/docs/hooks-intro.html
Sedberg, J. (2020, Januar). Samtale med Jone. Tøtlandsvik.
Semantic Release. (2021, 02 09). *Github Semantic Release Docmentation*. Hentet fra Semantic Release: https://github.com/semantic-release/semantic-release
Sullivan, B. (2019, 10 9). *Microsoft Docs*. Hentet fra Streamline Security Practices For Agile Development: https://docs.microsoft.com/en-us/archive/msdn-magazine/2008/november/agile-sdl-streamline-security-practices-for-agile-development
TechStacker. (2020, mai 19). *https://techstacker.com/*. Hentet fra https://techstacker.com/: https://techstacker.com/server-side-rendering-ssr-pros-and-cons/
TEK-Tools. (2020, Oktober 20). Hentet fra React Logging and Error Handling Best Practices: https://www.tek-tools.com/cloud/react-logging-and-error-handling-best-practices
Tengesdal, M. (2018). Frå transistor til datamaskin. I M. Tengesdal, *Frå transistor til datamaskin* (s. 9). Stavanger: Universitetet i Stavanger.
Third Normal Form (3NF). (u.d.). Hentet fra studytonight: https://www.studytonight.com/dbms/third-normal-form.php
typescriptlang. (u.d.). *https://www.typescriptlang.org/.* Hentet fra https://www.typescriptlang.org/: https://www.typescriptlang.org/
Universitetet i Stavanger. (2019, 8 18). *forelesning05-securebydesign-1t.pdf.* Hentet fra Canvas: https://stavanger.instructure.com/courses/4560/files/folder/Forelesninger/Samling%201%20-%2022_8?preview=537280
verma_anushka. (2020, August 31). *https://www.geeksforgeeks.org*. Hentet fra https://www.geeksforgeeks.org: https://www.geeksforgeeks.org/difference-between-fetch-and-axios-js-for-making-http-requests/
Visual Paradigm | ER-Diagram. (2021, 03 16). Hentet fra What is entity relationship diagram? : https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/
W3C. (2016). *https://www.w3.org/standards/webdesign/script.* Hentet fra https://www.w3.org: https://www.w3.org/standards/webdesign/script

W3C. (2021, 05 07). *https://www.w3.org/Style/CSS/.* Hentet fra https://www.w3.org: https://www.w3.org/Style/CSS/Overview.en.html

w3schools. (u.d.). *w3schools.* Hentet fra https://www.w3schools.com/: https://www.w3schools.com/whatis/whatis_npm.asp

What is a REST API? (u.d.). Hentet fra Red Hat: https://www.redhat.com/en/topics/api/what-is-a-rest-api

What is Entity Framework? (u.d.). Hentet fra Entity Framwork Tutorial: https://www.entityframeworktutorial.net/what-is-entityframework.aspx

What is Second Normal Form? (u.d.). Hentet fra studytonight: https://www.studytonight.com/dbms/second-normal-form.php

WHATWG. (u.d.). *html.spec.whatwg.org/multipage/.* Hentet fra html.spec.whatwg.org/multipage/: https://html.spec.whatwg.org/multipage/

Yosef, N. (2017, Desember 13). *https://codeburst.io*. Hentet fra https://codeburst.io: https://codeburst.io/the-ugly-side-of-redux-6591fde68200

16 Vedlegg

Kildedata som er benyttet til testing av applikasjonen kan ikke tilgjengelig gjøres da denne er konfidensiell.

16.1 Kildekode

Kildekode til alle tjenestene og skript: https://github.com/fishy-bachelor/taqua-data-system, se README.md for mer informasjon knyttet til strukturen. Merk at alle nøkler er fjernet.

16.2 Systemets Endepunkter

Dette vedlegget har som formål å dokumentere endepunktene til API-et på en god måte, som videre kan brukes når APIet skal implementeres. * brukes for å markere at alle brukere skal ha tilgang til dette endepunktet.

Kontroller	Endepunt	Metode	Rolle	Funksjon
Auth	/Auth/SignIn	POST	*	Autentisere seg for å få JWT token.
	/Auth/SignUp	POST	Admin	Registrere/endre brukere
DataKey	/DataKey	GET	Admin,Leser, Skriver	Hente alle datanøkler
	/DataKey	POST	Admin	Opprette nye datanøkler
	/DataKey/{id}	GET	Admin, Leser, Skriver	Hente datanøkkel på id
	/DataKey/{id}	PUT	Admin	Endre datanøkkel
	/DataKey/{id}	DELETE	Admin	Slette datanøkkel
FeedingLog	/FeedingLog	GET	Admin, Leser, Skriver	Hente all data fra foringslogg.
	/FeedingLog	POST	Data System	Legge inn ny data fra foringslogg.
	/FeedingLog/filer	GET	Admin, Leser, Skriver	Hente data fra foringslogg på periode og hall.
	/FeedingLog/{id}	GET	Admin, Leser, Skriver	Hente rad fra foringslogg basert på id.
Hall	/Hall	GET	Admin, Leser, Skriver	Hente alle haller

	/Hall	POST	Admin, Skriver	Legge til haller
	/Hall/{id}	GET	Admin, Leser, Skriver	Hante hall basert på hall-id
	/Hall/{id}	PUT	Admin, Skriver	Endre hall
	/Hall/{id}	DELETE	Admin, Skriver	Slette hall
ManualInput	/ManualInput	GET	Admin, Skriver, Leser	Hente all manuell data.
	/ManualInput	POST	Admin, Skriver	Legge til manuell data
	/ManualInput/filter	GET	Admin, Skriver, Leser	Hente manuell data på periode og hall.
	/ManualInput/{id}	GET	Admin, Skriver. Leser	Hente manuell data på id.
	/ManualInput/{id}	PUT	Admin, Skriver	Endre manuell data på id
	/ManualInput/{id}	DELETE	Admin, Skriver	Slette manuell data på id.
Period	/Period	GET	Admin, Skriver, Leser	Hente alle perioder
	/Period	POST	Admin, Skriver	Opprette ny periode
	/Period/{id}	GET	Admin, Skriver, Leser	Hent periode på id
	/Period/{id}	PUT	Admin, Skriver	Endre periode
	/Period/{id}	DELETE	Admin, Skriver	Slette periode
Pool	/Pool	GET	Admin, Skriver, Leser	Hente alle kar
	/Pool	POST	Admin, Skriver	Legge til kar på en hall
	/Pool/{id}	GET	Admin, Skriver, Leser	Hente kar på kar-id.
	/Pool/{id}	PUT	Admin, Skriver	Endre kar
	/Pool/{id}	DELETE	Admin, Skriver	Slette et kar
PoolPeriod	/PoolPeriod	POST	Admin, Skriver	Slette kar-periode.
	/PoolPeriod/{id}	PUT	Admin, Skriver	Endre kar-periode
	/PoolPeriod/{id}	DELETE	Admin, Skriver	Slett kar-periode

RasLog	/RasLog	GET	Admin, Skriver, Leser	Hente all dataene fra rasloggen
	/RasLog	POST	Datasystem	Opprette nye loggverdier
	/RasLog/filer	GET	Admin, Skriver, Leser	Hente raslogg på hall-id og periode.
	/RasLog/{id}	GET	Admin, Skriver, Leser	Hente raslogg på raslogg id.
	/RasLog/ProcessLogfile	POST	Datasystem, Admin	Endepunkt for å laste opp og prosessere loggfiler.
User	/User/listUsers	GET	Admin	Hente alle brukere
	/User/changeUserRole	PATCH	Admin	Endre rollen til en bruker
	/User/changeUser	PATCH	Admin, brukeren selv.	Endre informasjon på bruker.
	/User/deleteUser	DELETE	Admin	Slette bruker
	/User	GET	*	Hente informasjon om den innloggede brukeren.

16.3 Sikkerhetskrav fra Microsoft

16.3.1 Flerfaktorkrav

- Apply input validation (LOB)
- Avoid Exec in stored procedures
- Compile all code with the /GS compiler option
- Comply with SDL firewall requirements
- Conduct internal security design review (LOB)
- Do not use banned APIs in new code
- Encrypt all secrets, such as credentials, keys, and passwords (LOB)
- Ensure all database access is performed through parameterized queries to stored procedures
- Ensure all team members have had security education within the past year
- Ensure the application domain group is granted only execute permissions on the database stored procedures
- Fix all issues identified by code analysis tools for unmanaged code

- Fix all security issues identified by CAT.NET and FxCop static analysis
- Follow input validation and output encoding guidelines to defend against cross-site scripting attacks
- Harden or disable XML entity resolution
- Host security deployment review (LOB)
- Link all code with the /dynamicbase linker option (Address Space Layout Randomization)
- Mitigate against cross-site request forgery (CSRF)
- Mitigate against cross-site scripting (XSS)
- Secure sensitive data-at-rest (LOB)
- Secure sensitive data-in-transit (LOB)
- Update threat models for new features
- Use safe redirect
- Use secure cookie over HTTPS
- Use standard annotation language (SAL) to annotate all functions
- Utilize LOB Secure Code Review (LOB)

(microsoft, 2012)

16.3.2 Bøttekrav

Sikkerhet verifisering:

- Debug the application with the Application Verifier enabled
- Ensure regular expressions must not execute in exponential time (O(2^n))
- Ensure sample code complies with appropriate SDL development practices
- Employ network fuzzing
- Perform ActiveX control fuzzing
- Perform attack surface analysis
- Perform binary analysis (BinScope), FxCOP??
- Perform cross-domain scripting testing
- Perform file fuzz testing
- Perform RPC fuzz testing

(microsoft, 2012)

Design gjennomgang:

- Avoid cross-domain access to authenticated sites
- Comply with User Account Control (UAC) best practices to ensure all code runs as a non-administrator
- Conduct a privacy review
- Ensure all code is compliant with the SDL Cryptographic Standards
- Ensure all code is compliant with the SDL Privacy Guidelines document
- Incorporate third-party component licensing security requirements in all new contracts
- Opt out of automatic MIME sniffing
- Use strongly named assemblies, and request minimal permissions

(microsoft, 2012)

Respons:

- Add or update privacy scenarios in the test plan
- Create or update the list of response contacts
- Define or update the privacy bug bar
- Define or update the security bug bar
- Ensure symbols are available internally for all public releases

(microsoft, 2012)

16.3.3 Engangs-/ombordstigningskrav

- Avoid writable PE segments
- Create a baseline threat model
- Determine security response standards
- Do not use Visual Basic 6 to build products
- Establish a security response plan
- Identify primary security and privacy contacts
- Identify your team's privacy expert
- Identify your team's security expert
- Threat model your product, its attack surface, and its new features
- Use approved XML parsers

- Use latest compiler versions
- Use minimum code generation suite and libraries

(microsoft, 2012)

16.4 DBF field types and specifications

Show

α λτ **DBF Field Types and Specifications**

Advantage Concepts

Standard DBF Table Data Types

The following are standard field data types available in DBF tables. The length indicates the amount of data stored in the record image. For fields that include memo file storage, see the description for the amount of data that can be stored.

Type	Length	Decimals	Description
Character	1 to 65534	N/A	Fixed-length character field that is stored entirely in the table. Note that if you want the table to be compatible with Visual FoxPro, you should limit the field length to 254 or less.
Numeric	1 to 32	0 to Length-2	Fixed-length (exact ASCII representation) numeric.
Date	8	N/A	8-byte date field in the form of CCYYMMDD.
Logical	1	N/A	1-byte logical (boolean) field. Recognized values for True are '1', 'T', 't', 'Y', and 'y'.
Memo	10 or 4	N/A	Variable-length memo field. The size of each field is limited to 4 GB. Visual FoxPro memo fields require 4 bytes in the record image, while standard DBF memo fields require 10 bytes. The memo data is actually stored in a separate file, called a memo file, to reduce table bloat.

Extended DBF Table Data Types

The following are extended field data types available in DBF tables. The extended data types are non-standard DBF extensions. Non-Advantage applications that read DBF tables may not be able to open and read tables that have extended data types. Visual FoxPro will recognize double, integer, general, and picture fields.

Type	Length	Decimals	Description
Double	8	0-20	8-byte IEEE floating point value in the range 1.7E +/-308 (15 digits of precision). The decimal value affects NTX indexes and the use of the field in expressions. It does not affect the precision of the stored data. If the length is given, it will be ignored. For example, "salary, double, 10, 2" and "salary, double, 2" produce the same field.
Integer	4	N/A	4-byte long integer values from -2,147,483,648 to 2,147,483,647.
ShortDate	3	N/A	3-byte date field. This type supports the same range of dates as a standard date field.
Image	10	N/A	Variable-length memo field containing binary image data. The size of each field is limited to 4 GB. The binary image data is actually stored in a separate file, called a memo file, to

			reduce table bloat. If using the Advantage CA-Visual Objects RDDs, Advantage Client Engine APIs must be used to set and retrieve the image data. Most non-Advantage applications will interpret this data as a text memo field with a short text identifier.
α λτ			
Binary	10	N/A	Variable-length memo field containing binary data. The size of each field is limited to 4 GB. The binary data is actually stored in a separate file, called a memo file, to reduce table bloat. If using the Advantage CA-Visual Objects RDDs, Advantage Client Engine APIs must be used to set and retrieve the binary data. Most non-Advantage applications will interpret this data as a text memo field with a short text identifier.

Visual FoxPro DBF Table Data Types

In addition to the standard DBF types, the following types can be used when using the ADS_VFP table type (Visual FoxPro driver). These are fully compatible with Visual FoxPro.

Type	Length	Decimals	Description
Double	8	0-20	8-byte IEEE floating point value in the range 1.7E +/-308 (15 digits of precision). The decimal value affects NTX indexes and the use of the field in expressions. It does not affect the precision of the stored data. If the length is given, it will be ignored. For example, "salary, double, 10, 2" and "salary, double, 2" produce the same field.
Integer	4	N/A	4-byte long integer values from -2,147,483,648 to 2,147,483,647.
Binary	4	N/A	Variable-length field containing binary (Blob) data. The size of each Blob is limited to 4 GB. The binary data is stored in the FPT memo file associated with the table.
Money	8	4 implied	Currency data stored internally as a 64-bit integer, with 4 implied decimal digits from -922,337,203,685,477.5807 to +922,337,203,685,477.5807. The Money data type will not lose precision.
TimeStamp	8	N/A	DateTime value containing year, month, day, hour, minute, and second. Note that the timestamp value can contain milliseconds, but Visual FoxPro always rounds to the nearest second. Unlike ADI index keys, VFP TimesStamp index keys do not contain the millisecond portion of the value.
Character NoCPTrans	1 to 65534	N/A	Fixed-length character field that is stored entirely in the table. The NoCPTrans (binary) option indicates that ANSI/OEM translations will not be performed on the data. Note that if you want the table to be compatible with Visual FoxPro, you should limit the field length to 254 or less.
Autoinc	4	N/A	4-byte read-only positive integer value from 0 to 4,294,967,296 that is unique for each record in the table. A starting value and the

				increment value can be supplied when creating the table.
α λτ	**Memo NoCPTrans**	4	N/A	Variable-length memo field containing character data. The size of each field is limited to 4 GB. The NoCPTrans (binary) option indicates that the ANSI/OEM translations will not be performed on the data.
	Varchar	1 to 254	N/A	This field type allows variable length character data to be stored up to the maximum field length, which is specified when the table is created. It is similar to a character field except that the exact same data will be returned when it is read without extra blank padding on the end.
	Varchar NoCPTrans	1 to 254	N/A	This is the same as a VarChar except that no ANSI/OEM translations will be performed on the data.
	Varbinary	1 to 254	N/A	Variable length binary data. The maximum length of data that can be stored in the field is specified when the table is created.

- When creating Visual FoxPro (VFP) tables, you can specify that a specific field allows NULL values. To do this, include "NULL" in the table creation string. If it is for an Advantage Client Engine API such as AdsCreateTable, include it as an additional comma delimited item. The same can be done with the NoCPTrans option. For example, the table creation string "c,char,10,null,nocptrans; i,integer,null;" specifies a table with two fields that can hold NULL values and with no codepage translations performed on the character field. With SQL, the options are provided after the field type. The equivalent SQL field definition would be "(c char(10) null nocptrans, i integer null)".

Kilde:http://devzone.advantagedatabase.com/dz/webhelp/advantage9.0/server1/dbf_field_types_and_specifications.htm , lastet ned 13.04.2021