Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for IKT og realfag

Sverre Lofthus
Petter Høvik Lintoft
Sigurd Greiff
Daniel André Brunstad Iversen

# Skybasert minidatavarehus og BI-system med API og status-app for Cordel

Bacheloroppgave i Dataingeniør
Veileder: Anniken Karlsen, PhD.

Mai 2021

NTNU
Kunnskap for en bedre verden

Sverre Lofthus
Petter Høvik Lintoft
Sigurd Greiff
Daniel André Brunstad Iversen

# Skybasert minidatavarehus og BI-system med API og status-app for Cordel

Bacheloroppgave i Dataingeniør
Veileder: Anniken Karlsen, PhD.
Mai 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for IKT og realfag

NTNU

Kunnskap for en bedre verden

# EGENERKLÆRING

Alle studentene i prosjektgruppen har satt seg inn i hvilke hjelpemidler som er lovlige og retningslinjene for bruk av kilder. Vi er klar over hvilke konsekvenser fusk vil medføre.

| *Du/dere fyller ut erklæringen ved å klikke i ruten til høyre for den enkelte del 1-6:* | | |
|---|---|---|
| 1. | **Jeg/vi erklærer herved at min/vår besvarelse er mitt/vårt eget arbeid, og at jeg/vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.** | ☒ |
| 2. | **Jeg/vi erklærer videre at denne besvarelsen:**<br>• ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands.<br>• ikke refererer til andres arbeid uten at det er oppgitt.<br>• ikke refererer til eget tidligere arbeid uten at det er oppgitt.<br>• har alle referansene oppgitt i litteraturlisten.<br>• ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse. | ☒ |
| 3. | Jeg/vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§14 og 15. | ☒ |
| 4. | Jeg/vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert i Ephorus, se **Retningslinjer for elektronisk innlevering og publisering av studiepoenggivende studentoppgaver** | ☒ |
| 5. | Jeg/vi er kjent med at høgskolen vil behandle alle saker hvor det forligger mistanke om fusk etter **høgskolens studieforskrift §31** | ☒ |
| 6. | **Jeg/vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider** | ☒ |


**Postadresse**
NRNU i Ålesund
N-6025 Ålesund
Norway

**Besøksadresse**
Larsgårdsvegen 2

**Internett**
www.hials.no

**Telefon**
70 16 12 00

**Epostadresse**
postmottak@hials.no

**Telefax**
70 16 13 00

**Bankkonto**
7694 05 00636

**Foretaksregisteret**
NO 971 572 140

# FORORD

I denne rapporten beskriver vi et prosjekt vi har gjennomført i samarbeid med Cordel. Vi er fire dataingeniører som sammen har utført en bacheloroppgave. Oppgaven har bydd på mange nyttige erfaringer som prosjektplanlegging, tolking av oppgave og fullstack utvikling. Gjennom rapporten beskriver vi det teoretiske grunnlaget, hvilke metoder som er brukt og hvilke resultater vi har oppnådd i prosjektet.

Oppdragsgiver Cordel Norge, et datterselskap av SmartCraft, er en av de ledende leverandørene av programvare til håndverksbedrifter. De har omkring 60 ansatte ved deres tre kontorer Ålesund, Oslo og Hønefoss, hvor Ålesund er hovedkontoret. Cordel brukes av over 1600 norske bedrifter og har over 12000 brukere (1).

Cordel Norge sine kunder har forspurt et BI-system de kan benytte seg av for å få innsyn i spennende bedriftsinformasjon. Siden Cordel allerede håndterer mye relevant informasjon for et slikt system så er dette et behov de kan dekke. Det de trenger for å oppnå et slikt resultat er et datavarehus for innsending og lagring av disse dataene, en nettside med et BI-system, en app for å kontrollere og overvåket datavarehuset og en API for å håndtere kommunikasjonen mellom datavarehuset og brukergrensesnittene i appen og nettsiden.
Dette prosjektet skal fungere som et viktig fundament i videre utvikling for Cordel. Derfor har de forspurt bestemte programvarer vi skal benytte oss av ved utføring av prosjektet. Dette førte til at deler av prosjektet bestod av å bli kjent med de forskjellige programmene vi måtte benytte oss av. Vår løsning for dette var å lage en mindre løsning enn hva den faktiske løsningen vil være for hver del av prosjektet. På denne måten ble vi kjent med de forskjellige programvarene de forspurte, samtidig som vi fikk demonstrert prosjektets gjennomførbarhet.

Vår kontaktperson hos Cordel har vært Ole Johan Larsen. Ole har gjennom hele prosjektet gitt god tilbakemelding, forklart forskjellige problemstillinger og vært med på idémyldring for hvordan vi i samspill med oppdragsgiver skal lage det best mulige produktet. Dette har vært til stor hjelp for å nå målet vårt; Skape et produkt Cordel og deres kunder kan ta i bruk.

Takk til Kristoffer Larsen og Harald Bjørshol hos Cordel Norge. De har bidratt med forberedelsen til Cordels implementering av vårt system ved bruk av Docker og Swagger.

En spesiell takk til Anniken Karlsen ved NTNU Ålesund, som har veiledet prosjektet underveis. Hun har bidratt med sin erfaring og kompetanse både på nett og på møter hos Cordel. Her har vi fått tilbakemeldinger og tips om alt fra møteføring til prosjekt-styring og gjennomføring.

INNHOLD

# SAMMENDRAG

Oppgaven vi har løst for Cordel var å lage en skyløsning for samling av data i et datavarehus. Vi har laget et Bi-system for dataanalyse presentert grafisk, som er klart for bruk av Cordel. I tillegg har vi laget en status-app som skal brukes av Cordel, for å se status på løsningen. I denne rapporten starter vi med forklaring om det teoretiske grunnlaget for utviklingen av systemet, videre en beskrivelse av det ferdige produktet, og til slutt en beskrivelse av utviklingsprosessen.
Bi-systemet vi har laget skal bli brukt av Cordel sine kunder og er ment for mindre bedrifter som ikke behøver dyre fullskala-løsninger. Brukerne av systemet kan benytte seg av datamaskiner, telefoner eller nettbrett for å se disse dataene som ligger i nettsiden. Nettsiden inneholder data som for eksempel fraværsdata og kundefordringer.

Vi har også laget en mobilapplikasjon som har en oversikt over feil ved innsending av data og registrerte tennants. I tillegg har vi implementert muligheten for å registrere nye brukere til Bi-systemet ved hjelp av denne mobilapplikasjonen. Denne appen er crossplatform og mulig å bruke på både telefon og nettbrett.

Vårt system håndterer daglige innsendinger av data fra Cordel sine systemer, for så å lagre dette i datavarehuset.

# TERMINOLOGI

## Begreper

| | |
|---|---|
| **Tennant** | En bedrift som benytter seg av løsningen. |
| **Klient** | En person eller selskap som er en kunde til en tennant |
| **Responsiv** | Tilpasse seg skjermstørrelsen som brukes når designet skal vises. |
| **Base64URL** | Base64 er en gruppe av like binær-til-tekst enkodingskjema som representerer binær data i en ASCII string format. URL-delen av utrykket betyr at strengen er sikker å bruke i en URL (2). |
| **Frontend** | Programvaren som ligger nærmest brukeren og er koden som former det visuelle man ser på skjermen og som bestemmer hva som skjer når man interagerer med elementene (3). |
| **Backend** | Programvaren som ligger nærmest databasen, hvor de tunge kalkuleringsprosessene skjer. Brukeren har ikke nødvendigvis noe forhold til hva som skjer her (4). |
| **Wireframes** | Wireframes er en grafisk beskrivelse eller tegning som gjør det enkelt for å utviklere og designere å kommunisere om strukturen til en programvare eller nettside (5). |
| **Widget** | En widget er et element av et grafisk brukergrensesnitt, som viser informasjon eller en vei for å interagere med en applikasjon (6). |
| **Repository** | Repositories er en plass hvor man lagrer alle filene i et prosjekt, og er en plass hvor man kan endre og diskutere prosjektet (7). |
| **Commit** | En commit er et øyeblikksbilde av et repository. Når man gjør en commit oppdaterer man de lagrede filene ut ifra endringene som er gjort (8). |
| **Branch** | En branch er brukt for å isolere utviklingen i et prosjekt, slik at man ikke påvirker andre brancher. Brancher kan bli slått sammen med andre brancher, for å slå sammen filer (9). |

**SDK** SDK er en forkortelse for «Software development kit». En SDK hjelper utvikleren å lage applikasjoner for en spesifikk platform, system eller programmeringsspråk (10).

**Superset** Superset betyr I programmeringssammenheng, noe som fungerer som et lag rundt noe. Dette tillater utviklere å bruke flere metoder som igjen krever at du følger en spesiell måte å drive med utvikling, som du ikke behøver uten dette laget (11).

**Object-relational mapper** Orm representerer et sett av teknikker brukt i programmering, som prøver å lage en bro mellom inkompatible systemer, slik at de kan kommunisere, samarbeide og utveksle informasjon. Alt dette samtidig som de forbedrer livet til en utvikler (12).

**Claims** Claims er informasjon om en entitet, som regel en bruker-entitet (13).

**Api-key** En Api-key er en lang string med bokstaver og tall som brukes til å autentisere forespørsler (14).

## Forkortelser

| | |
|---|---|
| PoC | Proof of Concept |
| UML | Unified Modeling Language |
| UP | Unified Process |
| MVP | Minimum viable product (minst brukbare produkt). |
| URL | Unifrom Resource Locator |
| ID | Identiteter |
| HTTP | Hypertext Transfer Protocol |
| API | Application Programming Interface |
| GUID | Globally Unique Identifier |
| DOM | Document Object Model |
| BI | Business Intelligence |

# 1 INNLEDNING

Denne bacheloroppgaven er utført av Petter Høvik Lintoft, Sigurd Brustad Greiff, Daniel Brunstad Iversen og Sverre Lofthus. Oppgaven har til hensikt å utvikle et datavarehus med en frontend, bestående av en administrasjonsapp og et Bi-system, ved NTNU Ålesund etter krav formulert av Cordel Norge.

Dette kapitlet omhandler årsaken til oppgaven, selve problemstillingen og hvilke avgrensinger Cordel har hatt for dette prosjektet. Til slutt skriver vi litt om hvordan rapporten er lagt opp.

## 1.1 Introduksjon

Oppgaven er motivert av sin relevans til vårt datastudium, hvor databaser, systemutvikling og nettverksutvikling har vært kjernefokus i utdanningsløpet. Oppgaven har gitt oss muligheten å tilføre tidligere kunnskap inn i en arena med mange nye teknologier. Ved å jobbe mot en reell oppdragsgiver har vi gode forutsetninger for å jobbe med et prosjekt som byr på virkelighetsnære utfordringer og det dette innebærer.

## 1.2 Problemstilling

Vår oppgave er gitt av Cordel Norges utviklingssjef Ole Johan Larsen. Cordel er et firma som hjelper med administrativt arbeid for håndverkere. De ville derfor ha et automatisert datavarehus som mottar informasjon om sine kunder, for så å analysere bedriftsdataen og visualisere det gjennom et BI-system. I tillegg ønsket de en mobil applikasjon for overvåkning av feilmeldinger og brukere, samt mulighet for oppretting av nye brukere. Cordel har allerede integrasjon mot lignende løsninger for sine kunder, men dette prosjektet skal bli brukt av mindre kunder, hvor en løsning i mindre skala passer bedre.

## 1.3 Avgrensinger

Cordel fremmet tidlig ønske om å enkelt starte med videreutvikling av systemet. Det ble derfor satt noen avgrensinger og ønsker om hvilke teknologier vi skulle ta i bruk ved utvikling av datavarehuset. Cordels ønsker var:

- Databasene skal bruke PostgreSQL.
- Backend med REST-API skal være skrevet i C#.
- Nettsiden utvikles ved bruk av React og Typescript.

- Mobilapplikasjon skrevet i flutter for bruk til både Android og IOS.

## 1.4 Innhold

Vi vil i det følgende dokumentere arbeidet som er gjort i forbindelse med problemstillingen gitt av Cordel Norge.

Først vil det etableres et teoretisk grunnlag som vi har brukt i utredelsen av oppgaven. Når det teoretiske grunnlaget er lagt, vil vi betrakte prinsipper, organiseringsmetoder og annen metodikk som vi brukte i utviklingen av prosjektet. Videre vil vi ta for oss de tekniske verktøyene som vi tok i bruk ved utviklingen av systemet og hvordan disse spiller inn i det endelige resultatet.

# 2 TEORETISK GRUNNLAG

I dette kapittelet skriver vi om teorier vi har brukt for å utvikle vårt system. Det er blant annet teorier om design, datamodellering, objektorientert programmering, arbeidsmetodikk, kommunikasjonsprotokoller og sikkerhet.

## 2.1 Design

Designbegrepet kan tolkes på mange forskjellige måter, men en måte å definere det på er «En plan over noe som skal realiseres» (15). Ved å definere det på denne måten, har vi et ord som kan strekke seg over store deler av dette prosjektet. Fra datamodellering og skissering i planleggingsfasen til hvordan utseendet til nettside og app i prosjektet skal se ut.

### 2.1.1 Don Normans designprinsipper

I Don Normans artikkel «Design Rules Based on Analyses of Human Error» (16) presenterer han flere regler for brukergrensesnittdesign. Disse reglenes hovedfokus handler om å redusere muligheten for brukere å gjøre feil ved bruk av et system.

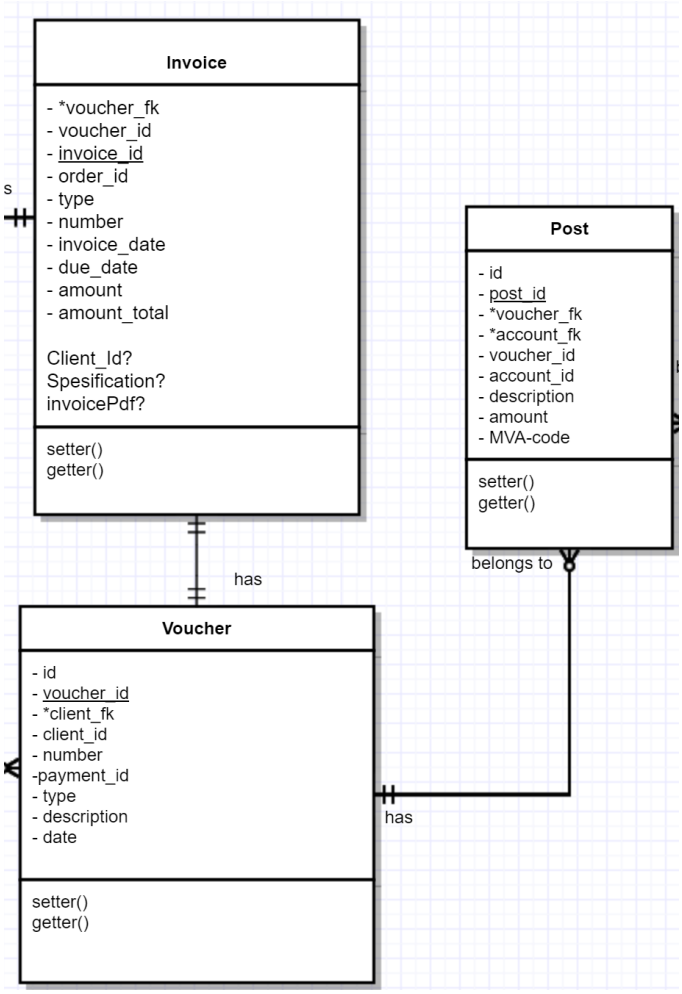
Selv i systemer som er designet og brukt av de beste, vil mennesker gjøre feil. Derfor må man gjøre det så vanskelig som mulig å gjøre brukerfeil, samtidig som man minsker effekten av en mulig bruksfeil (16).

I den samme artikkelen definerer Norman(16) beskrivelsesfeil. Disse feilene oppstår ifølge Norman(16) når man for eksempel har flere knapper som er like, men har ulike utfall. En situasjon som da kan oppstå er at en bruker trykker på feil knapp i forhold til hva brukeren egentlig ønsket å oppnå. En løsning på dette problemet kan da være å vise gode hint og beskrivelser om hva en handling vil medføre (16).

Norman mener (16) at å være konsekvent i utformingen av brukergrensesnittet vil minke faren for at systemet blir brukt på feil måte. Han grunngir dette med at en bruker kan være vant til å gjøre noe på en bestemt måte for å oppnå et bestemt resultat. For eksempel kan en bruker være vant til at en knapp har en spesifikk utforming som er knyttet til en spesifikk handling. Dersom en knapp med utforming lik noe man er vant til, men ikke har samme funksjon, kan dette føre til at brukeren velger å bruke knappen feil. Konsekvent utforming av grensesnitt og bruk av normale designstandarder kan hjelpe å minke faren for at dette oppstår, påpeker Norman (16).

### 2.1.2 Datamodellering

Datamodellering er en måte å beskrive sammenhengen og strukturen mellom objekter på en systematisk måte. Denne teknikken bidrar til en logisk struktur i databasen og har derfor vært et godt hjelpemiddel for oss. I en datamodell har en entitetstype en egen boks som kobles opp mot andre entitetstyper med ulike linjer som har ulik betydning (17). Figur 2.1 demonstrer et en til mange forhold og en til en forhold.




*Figur 2.1 En til en forhold og en til mange forhold, hentet fra vårt UML-diagram*

Figur 2.1 viser en entitetstypene «voucher», «invoice» og «post». En voucher har en invoice og omvendt, som er et en til en forhold. En voucher har også flere posts, men en post har kun en voucher, som er et en til mange forhold. På den måten blir det oversiktlig og enkelt å se hvordan ting henger sammen. Datamodellen blir enda mer tydelig ved å legge til en beskrivende tekst.

For å få en mer informasjonsrik datamodell blir det brukt attributter som opplyser hva en entitet inneholder. De fleste entitetene har ofte et attributt som er unikt. For markeringene av ulike attributter tok vi utgangspunkt i boka Datamodellering – praksis og teori (18) og brukte datamodelleringspraksisen fra denne. Attributtene blir markert med en understrek og brukes som en fremmednøkkel i andre entiteter for å skape en kobling mellom entitetstypene. Fremmednøkler blir ofte markert med en «*» i en datamodell (17).

## 2.2 Datavarehus

Et datavarehus blir brukt for håndtering og analyse av historisk data, og kan komme fra flere forskjellige kilder. Siden datavarehuset og forretningens indre databaser ikke kjører på samme system, vil henting av data fra datavarehuset ikke forstyrre eller ta opp ressurser fra forretningsdatabasen. Dataen kommer som vi vet fra mange forskjellige databaser som vil bli behandlet og renset før det blir sendt videre. Dette er for å sikre integriteten av dataen som blir lagret i datavarehuset (19).

Det er fire hovedtrekk med et datavarehus. Disse er (19):

- **Fagrettet** (Subject-oriented): At datavarehuset er fagrettet betyr at data blir lagret og fremstilt som flere temaer, som for eksempel salg, budsjett og markedsføring.
- **Integrert** (Integrated): Når man sier at et datavarehus er integrert, viser man til at data fra forskjellige databaser og forskjellige bedrifter, som muligens har forskjellige navn og datatyper på felt i datavarehuset, blir slått sammen med felles variabelnavn og datatyper.
- **Tidsvarierende** (Time-Variant): I datavarehuset vil data bli lagret i forskjellige tidsrammer.
- **Permanent** (Non-Volatile): Når data først har blitt lagt inn i et datavarehus vil man ikke ha mulighet til å endre eller slette dataen på et senere tidspunkt.

Å lagre historisk data i et datavarehus gjør det enklere for bedriften å kunne analysere alt fra budsjetter til tidligere oppdrag. Dette gir mulighet for bedre oversikt over bedriften, og mulighet for økt lønnsomhet.

## 2.3 Business intelligence system

Jobben til et business intelligence system (Bi-system) er å presentere informasjonen av dataen som ligger i et datavarehus. Teknikker og verktøy brukes for å transformere rådata til meningsfull informasjon (20).

De fleste bedrifter bygger ikke egne Bi-systemer, men baserer seg på at en tredjepart gir tilgang til informasjonen i datavarehuset. Det er derfor viktig at informasjonen som blir presentert er korrekt, appellerende og lett å tyde (19).

## 2.4 Objektorientert programmering

Objektorientert programmering (OOP) er et programmeringsmønster som blir brukt i flere populære høynivå programmeringsspråk. OOP bruker objekter som er definert av klasser. En klasse inneholder felt som sier noe om hvordan et objekt vil se ut når det blir opprettet. For utviklere er OOP mer oversiktlig og det blir enklere å holde styr og organisere (21,22).

I OOP finnes det to hovedprinsipper som er viktige å følge, kohesjon og kobling. Målet for disse er å skape høy kohesjon og lav kobling i koden (22). I 2.4.1 og 2.4.2 gjør vi kortfattet rede for disse to prinsippene.

### 2.4.1 Kohesjon

Begrepet kohesjon handler om hvordan klasser og metoder er bygget opp. Ved et ideelt design vil klasser og metoder ha ansvar for en spesifikk oppgave og forårsake høy kohesjon. Dersom en klasse eller metode er ansvarlig for kun en ting vil det være enklere å gjenbruke klassen eller metoden i en ny kontekst. Lav kohesjon krever flere metoder og kodeduplisering vil oppstå (22).

### 2.4.2 Kobling

Kobling er et begrep som blir brukt for å beskrive hvor avhengige klasser er i forhold til hverandre. Her skilles det hovedsakelig mellom to typer, høy eller lav kobling. Høy kobling mellom ulike klasser vil gjøre det vanskeligere å endre eller implementere ny funksjonalitet. I en klassestruktur med høy kobling vil en endring i en klasse føre til at flere endringer blir nødvendige i andre klasser. Det er derfor ønskelig å oppnå lav kobling mellom klasser. Lav kobling gir lite avhengighet og gir færre konflikter ved endringer og implementasjoner (22).

## 2.5 Kommunikasjonsprotokoller

Kommunikasjonsprotokoller består av et sett med regler for informasjonsutveksling mellom digitale enheter (23). Det finnes flere protokoller man kan benytte seg av for å overføre informasjon over nett, noen av disse har vi benyttet oss av og skal derfor forklare litt om disse protokollene.

### 2.5.1 Hypertext Transfer Protocol (HTTP)

Hypertext Transfer Protocol (HTTP) er protokollen som blir brukt for sending av informasjon mellom en klient og en server. HTTP fungerer ved at klienten sender en forespørsel til

serveren. Etter at denne forespørselen har blitt behandlet vil serveren sende en respons tilbake til klienten. HTTP tar nytte av flere forskjellige nettverksprotokoller, et eksempel på dette er Transmission Control Protocol (TCP), som sikrer at all data kommer frem til mottaker. Andre nettverksprotokoller som User Datagram Protocol (UDP) kan også bli brukt, men er ikke pålitelig og sikrer ikke at dataen kommer frem (24).

**En HTTP forespørsel er bygd opp av** (24)**:**

- En metode (GET, POST, PUT, HEAD, DELETE, PATCH, OPTIONS, TRACE).
- En path (hyperlink).
- Versjonen av protokollen som blir brukt.
- En header (ekstra informasjon til serveren) eller en body (informasjon som blir sendt sammen med noen spesifikke metoder som for eksempel POST)

**En HTTP respons inneholder** (24)**:**

- Versjonen av protokollen som blir brukt.
- En statuskode som forteller om alt gikk som forventet, eller ikke (2.5.1.2).
- En statusmelding, som beskriver statuskoden kort.
- En header, som holder på mindre informasjon til web-klienten.
- En valgfri body, som inneholder den hentede dataen fra serveren

#### 2.5.1.1 HTTP Metoder

Ønsker man å sende en HTTP-forespørsel til en server, bruker man forskjellige metoder for å fortelle serveren hva vi ønsker å gjøre med denne forespørselen. Det er 8 forskjellige metoder man kan bruke i en http-forespørsel. De forskjellige er GET, POST, PUT, HEAD, DELETE, PATCH, OPTIONS og TRACE, hvor GET og POST er de mest brukte. (25) Nedenfor definerer vi de to forskjellige metodene vi har brukt i vårt prosjekt:

- **GET :** GET metoder blir brukt hvis klienten ikke sender med noe til serveren, men forventer at serveren skal sende noe tilbake. GET metoder kan sende med små mengder data i URL-en, men dette er sterkt begrenset da hver nettleser har ulike maksimale lengder på URL, sammenlignet med POST som kan sende med større mengder data i kroppen (body). Eksempler på når GET blir brukt er når man gjør en forespørsel til en webserver som svarer med alt av informasjon man trenger for å vise

nettsiden. Det man får som svar fra webserveren kan da være et HTML-dokument, et CSS-dokument og et JavaScript-dokument (26).

- **POST :** POST metoder har ingen begrensninger på hvor mye data som kan bli sendt med, og man bruker derfor denne når man ønsker å sende inn data til en server. Bruk av POST skjer når man for eksempel ønsker å opprette en ny bruker i et system. Eksempelvis sender man inn brukernavn og passord uten at det er nødvendig å få tilsendt data tilbake fra serveren. Istedenfor er det bare nødvendig med en respons som sier om det var vellykket eller ikke. (27).

#### 2.5.1.2 Respons-koder

Etter at en server har fått en forespørsel, sender serveren en respons tilbake sammen med en responskode. Denne koden beskriver kort om det har oppstått noen problemer eller om alt har gått som forventet. Her er noen av de mest brukte kodene i vårt prosjekt (28) :

- **200 OK :** Denne koden vil bli returnert hvis ingen feil har oppstått, og alt har gått som forventet. Denne responskoden vil også returnere data tilbake til brukeren hvis dette er forventet.
- **400 Bad Request :** Her har brukeren sendt en forespørsel som serveren ikke har forstått. Dette kan bli forårsaket av at serveren har forventet mer data enn klienten har sendt med.
- **401 Unauthorized :** Feilmeldingen 401 vil oppstå når en klient prøver å gjøre en forespørsel som klienten ikke har tilgang til.
- **404 Not Found :** Feilmeldingen 404 oppstår dersom forespørselen ikke kunne finne frem til den ønskede ressursen. En feil i URL-en vil kunne forårsake dette.
- **405 Method not allowed :** Denne responskoden vil oppstå hvis brukeren prøver å sende en forespørsel til serveren med en feil http metode. Et eksempel på dette kan være hvis klienten sender en GET forespørsel, men serveren forventer en POST forespørsel.
- **500 Internal Server Error :** Denne feilmeldingen kommer når det oppstår en feil under behandlingen av forespørselen som serveren ikke klarer å håndtere.

### 2.5.2 Hypertext Transfer Protocol Secure (HTTPS)

Hypertext Transfer Protocol Secure (HTTPS) kombinerer vanlig HTTP og en TLS- (Transport Layer Security) eller en SSL-kopling (Secure Sockets Layer) som krypterer data som blir sendt. Dette hindrer at uvedkommende får tak i informasjonen som blir sendt med forespørslene og responsene. Under en forespørsel vil ikke domenenavnet (for eksempel www.vg.no) bli kryptert, men resten av URL-en og forespørselen vil bli kryptert. De fleste nettsteder i dag som har stor trafikk eller som behandler sensitiv informasjon bruker derfor HTTPS (29).

### 2.5.3 Rest API (Representational State Tranfer API)

REST API er en arkitektur for programmeringsgrensesnitt som brukes til utveksling av data mellom to forskjellige applikasjoner (6). Denne kjente stilen har vi benyttet oss av i prosjektet for å tillate nettsiden til å hente informasjon fra datavarehuset, for så å visualisere det for brukerne. Mobilapplikasjonen benytter seg også av Rest API-et for å hente ut feilmeldinger og annen informasjon fra serveren. For at API-et skal være RESTful må det følge disse seks veiledende prinsippene (30):

1. **Klient-server-separasjon**
   Ved å separere klient og server vil forbindelsen mellom disse kun være i form av forespørsler, styrt av klienten, som serveren vil respondere på ved å sende en respons. Dermed vil ikke serveren sende masse forskjellig informasjon på egen hånd, men heller ligge å vente på forskjellige forespørsler fra klientene.
2. **Tilstandsløs:**
   Serveren husker ikke noe om kunden som benytter seg av API-et. Hver forespørsel vil derfor inneholde all informasjon serveren trenger for å genere en respons, uavhengig av tidligere forespørsler gjort av samme klient tidligere.
3. **Cacheable:**
   Data som blir sendt fra serveren må inneholde informasjon om det er muligheter for å lagre det i en buffer. På denne måten slipper klienten å forespørre samme data flere ganger, men heller oppdatere det hvis det skulle være en oppdatert versjon av dataen på serveren.
4. **Universelt grensesnitt:**
   Ved bruk av et universelt grensesnitt vil forespørselen fra forskjellige klienter være

like, uavhengig om det er en nettleser, en annen server, et python-script eller lignende som skal benytte seg av en API.

5. **Lagdelt system:**
   Ved bruk av et lagdelt system vil ikke klientene kunne vite nøyaktig hvilket lag av serveren den er knyttet opp mot.
6. **Kjørbar kode (valgfri):**
   Ved implementering av dette vil klienter kunne forespørre kjørbar kode fra serveren, noe som er valgfritt siden ikke alle klienter har slike behov (31).

Et annet viktig bruksområde for vår REST API er separering av data. Ved hjelp av JSON-Web token (2.6.2), kan vi hente ut informasjon om klienten slik at klienten kun får den informasjonen de skal ha mulighet til, men ikke noe mer.

## 2.6 Datastrukturer

For at mottaker skal vite hvordan dataen skal tolkes, har vi tatt i bruk noen kjente datastrukturer for sending av data mellom backend og frontend.

### 2.6.1 JSON

JavaScript Object Notation (JSON) er et dataformat som blir brukt til lagring og transportering av data som skal bli sendt fra en server til en klient. Dataen blir lagret som en tekst (string). Siden JSON er veldig utbredt og tekstbasert har de fleste kodespråk egen metoder for splitting av data fra et JSON objekt (32).

### 2.6.2 JSON Web Token

JSON web token (JWT) er en sikker og enkel måte for overføring av informasjon mellom to parter som et JSON objekt. En JWT har tre hoveddeler: en header, en payload og en signatur. Headeren består som regel av to deler, hvilken type token som er brukt og hvilken algoritme som er brukt for å signere JWT-en. Payload delen inneholder claims, identifiserende informasjon, for eksempel typisk informasjon om en bruker. Signaturen er en hashing av header, payload og en hemmelig nøkkel. Til slutt får vi da en Base64URL kodet JSON-Objekt som er separert med punktum (13).

## 2.7 Datalagring

Store deler av dette prosjektet omhandler lagring av forskjellig data for hver kunde, for så å gjøre spørringer mot disse dataene senere. Siden databasen håndterer flere kunder benytter vi oss av relasjonsdatabaser for å knytte kundedata opp mot kunden.

## 2.8 Smidige metoder

Det finnes flere forskjellige metodikker for programvare-utvikling, men for oss var det naturlig å velge smidige metoder, da det er kjent for oss og passer vårt prosjekt godt.

Den smidige programvaremetodikken er en enkel og effektiv prosess for utvikling av et produkt fra idestadiet. Det smidige manifestet nedenfor summerer de fire kjerneverdiene for smidige metoder (33):

1. Individer og interaksjoner er viktigere enn prosesser og redskap.
2. Fungerende programvare er viktigere enn omfattende dokumentasjon.
3. Samarbeid med kunden er viktigere enn kontraktsforhandlinger.
4. Respondere til endringer er viktigere enn å følge en plan.

Det å jobbe smidig har flere fordeler. Å jobbe målbevisst i iterasjoner kan hjelpe å forme naturlige datoer for møter, hvor vi har mulighet til å konkret vise det vi har oppnådd (33). På den måten har vi skapt en god arena for god kommunikasjon og hyppige tilbakemeldinger.

### 2.8.1 Scrum

Scrum er den smidige metodikken som blir mest brukt av programvareutviklere og er et rammeverk som hjelper arbeidsgrupper å jobbe sammen (34). Smidige metoder og Scrum baserer seg på mye av det samme, nærmere bestemt kontinuerlig forbedring. Det blir derfor ofte forvekslet om hverandre. Forskjellen er at for smidige metoder er dette et kjerneprinsipp, et tankesett, og Scrum er et rammeverk for å få dette gjort. Videre forklarer vi metoder Scrum bruker for å oppnå en smidig arbeidsmetodikk.

#### 2.8.1.1 Sprint

Når man jobber smidig og med bruk av Scrum jobber man i tidsbestemte perioder. En slik tidsperiode kalles en sprint. Varigheten på periodene kan variere, men for å opprettholde det smidige konseptet skal disse holdes innenfor en til fire uker. Når man skal planlegge disse

sprintene kommer man opp med arbeidsoppgaver som blir plassert i en backlog (2.8.1.3). Disse oppgavene får tildelt en poengsum som tilsvarer hvor mye tid som kreves for å fullføre oppgaven. Deretter velger man arbeidsoppgaver fra backloggen og tildeler de til gruppemedlemmer. Oppgavene forblir hovedfokuset til den som har fått oppgaven tildelt. Tidsestimeringen av arbeidsoppgavene kan ofte være vanskelig. Derfor diskuteres det hvordan sprinten har vært i et sprinttilbakeblikk etter fullført sprint, slik at man kan tilpasse neste sprint på en bedre måte (35).

#### 2.8.1.2 Daglig Scrum

Daglig Scrum er små, korte møter som gjerne skjer på starten av en arbeidsdag. Her går man gjennom hva hvert gruppemedlem jobbet med i går, hva de skal jobbe med i dag, og om det er noen problemer man har møtt på som kan hindre progresjonen i utviklingen. Kommunikasjon er viktig i gruppeprosjekter, og daglig Scrum kan hjelpe kvaliteten ved innspill fra andre gruppemedlemmer hvis det er noen problemer som oppstår (36). De daglige møtene har gjort det lettere for oss å vite hvordan progresjonen lå an i forhold til sprintfristen

#### 2.8.1.3 Backlog

Backlog er en samling av alle arbeidsoppgavene som skal gjøres for å forbedre et produkt. Disse oppgavene er kilden til alt arbeidet en scrumgruppe jobber med. Når en sprint skal planlegges blir det hentet og lagt til arbeidsoppgaver fra backloggen. Det er derfor viktig at backloggen inneholder flere oppgaver når en sprint skal utformes (37). For å holde backloggen oppdatert og presis har det vært en pågående aktivitet å endre og legge til oppgaver.

#### 2.8.1.4 Roller

En scrumgruppe består av tre forskjellige roller. Rollene består av en produkteier, Scrum Master og utviklerteam. En produkteiers rolle er å forsikre at oppgavene som blir gjort er hensiktsmessige og gir mest verdi. Ofte er bedriften representert som produkteieren, og har som ansvar å forteller utviklerteamet hva som er viktig å levere (38). En Scrum Master har ansvar for å etablere og tilse at arbeidsmetodikken følger de rette retningslinjene. Utviklerteamet er de som skal utvikle produktet.

### 2.8.2 Parprogrammering

I smidige metoder er det utviklet en teknikk kalt parprogrammering. Parprogrammering består av to utviklere som benytter samme arbeidsstasjon hvor en skriver koden mens den andre aktivt følger med og kommer med innslag. Noen av de fordelene vi forventet ved å benytte oss av en slik arbeidsmetodikk var følgende (39):

1. **Økt kodekvalitet**. Når vi programmerer sammen, øker nødvendigheten for å forklare hva vi gjør og hvordan vi tenker å utføre det. På denne måten unngår vi mange småfeil og får et presist syn på hva vi skal oppnå.
2. **Fordelt kunnskap**. Siden mange av konseptene og utviklerspråkene er nye så er det større sjans for at en av utviklerne ikke er kjent med dem. Da kan vi lettere øke kompetansen innad i gruppen for videre utvikling.
3. **Bedre kommunikasjon**. Når kommunikasjonen innad i gruppen forbedres vil også informasjonsflyten øke.
4. **Effektivitet**. Når visse deler av utviklingen kan bli abstrakte og vanskelig er det lett for en utvikler å sette seg fast på problemet. Når slike scenario oppstår, kan det overkommes ved bruk av denne arbeidsmetodikken.

## 2.9 Sikkerhet

Vårt datavarehus inneholder mye forskjellig informasjon, noe av det som kan kategoriseres som sensitiv informasjon. Det er derfor viktig å implementere forskjellige metoder for å unngå at sensitiv informasjon ender opp hos feil eller uønskede brukere.

### 2.9.1 Hashing

Hashing er prosessen som gjør at en verdi konverteres til en annen verdi. De fleste hash funksjonene er ikke reversible, noe som betyr at den ikke kan gå tilbake til sin originale verdi. Da det ikke er reversibelt er dette en god måte å kryptere sensitiv data på, som for eksempel passord. Dersom noen skulle få tilgang til databasen vil ikke passordene stå i plaintekst, men heller som en lang, uforståelig tekst. (40) Samme tekst vil alltid gi samme hash, noe som gjør det enklere for angripere å hente ut hashet informasjon med mindre vi legger til en salt (2.9.2).

### 2.9.2 Salt

Salt blir ofte brukt for å gjøre passord enda vanskeligere å tyde. Det legger til en tilfeldig streng med tekst før passordet blir hashet slik at alle passord blir unike selv om det er samme

tekst-input. Ved å legge til denne tilfeldige strengen med tekst vil dataen være mer sikker og det blir vanskeligere for angripere å hente ut informasjon (41,42). En salt kan være unik eller lik for hvert passord. Unike salt verdier er det mest effektive, da like passord vil ha ulik hash som vist i figur 2.2.


*Figur 2.2 Salt ved hashing (43)*

Fra figuren ser vi tydelig at samme passord uten salt vil generere samme hash. Ved å legge til en unik salt for hvert passord blir hver hash unik.

### 2.9.3 SQL injeksjon

SQL injeksjoner er en angrepsmetode hvor angriperen sender deler av en SQL-spørring som input til applikasjonen(44). Slik kan en angriper få tak i sensitiv data fra databasen eller annen data brukeren ikke skal ha tilgang til. Angriperen kan også gjøre endringer i databasen som for eksempel slette deler av databasen eller bare legge til nye ting som ikke skal ligge der (44).

## 2.9.4 Brute Force

Brute force er en vanlig måte angripere bruker for å få tilgang til sensitiv informasjon, da ofte passord. Det er en «prøv og feil» metode hvor angriperne sender flere forespørsler til en server i håp om å få noen verdifulle resultater. Ved å bruke en brute force metode kan det ta alt fra sekunder til år for å få et verdifult resultat (45). For å gjøre det vanskeligere for angriperne å få ønskede resultater kan krav til passord settes, som for eksempel spesialtegn, tall eller lengde på passordet (46).

## 2.10 Algoritmer

Informasjonen som skal visualiseres på nettsiden blir hentet fra datavarehuset. Siden datavarehuset lagrer mer informasjon enn nettsiden eller appen har bruk for, trengs det ofte algoritmer for å konvertere og/eller kalkulere dataen fra datavarehuset. Algoritmene er bestemte fremgangsmåter for hvordan informasjonen må behandles for å løse forskjellige oppgaver (47). Dette gjør at nettsiden og appen får ferdigstilt informasjon som de kan benytte seg av direkte og slipper å gjøre kalkuleringer og sjekker i frontend. Kjøretiden til en algoritme forklarer det asymptotiske forholdet mellom størrelsen på problemet og hvor lang tid det tar å løse det (48).

## 2.11 Testing

I programvareutvikling finnes det mange forskjellige metoder for å teste kode (49). For å teste deler av koden vi har skrevet benytter vi oss av to kjente metoder, unit testing og widget testing. Testing er viktig for bl.a. å unngå feil ved lansering av produktet, samtidig kan det høyne kvaliteten av koden (50).

### 2.11.1 Unit testing

En unit test er en type test som tester spesifikke deler av en applikasjon. Det kan være flere klasser, en klasse eller en metode. Alle metoder som har kompilert burde testes. Unit tester kan deles inn i positive og negative tester. En positiv test forventer et riktig resultat, mens en negativ test forventes å feile (22). En unit test bør testes i isolerte miljøer for at det ikke skal være avhengig av andre systemer som for eksempel en database (51).

### 2.11.2 Widget testing

Målet med en widget test er å sjekke at hver komponent i en widget utfører de oppgavene den er ment å gjøre (52). En widget test bør være så kort som mulig, og bør kun inneholde absolutt nødvendige elementer, ideelt kun en komponent (53).

# 3 MATERIALER OG METODE

I denne delen av rapporten beskrives fremgangsmåten for hvordan prosjektoppgaven ble løst. Dette innebærer blant annet hvordan prosjektet ble organisert og utført, hvilke verktøy og teknologier som er bruk, og hvilken dokumentasjon som har blitt utarbeidet.

## 3.1 Organisering

Oppgaven er delt inn i tre forskjellige organisatoriske parter. Vi vil her forklare hvilke roller de forskjellige partene har i oppgaven.

### 3.1.1 Prosjektgruppe

Vi er fire studenter fra NTNU Ålesund som sammen har utviklet en løsning for Cordel. Studentene er som følger: Sverre Lofthus, Sigurd Brustad Greiff, Daniel Andrè Brunstad Iversen og Petter Høvik Lintoft. Gjennom prosjektet hadde vi to ulike roller, prosjektleder og litteraturansvarlig. Prosjektlederen har som oppgave å passe på at sprint og prosjektarbeid går som planlagt, mens litteraturansvarlig har ansvar for at innleveringsfrister blir opprettholdt og dokumenter for prosjektet blir holdt kontroll på. Disse rollene ble rullert på gjennom hele prosjektet.

### 3.1.2 Oppdragsgiver

Oppdragsgiveren for dette prosjektet er Cordel Norge. De leverer et system tilpasset håndverkere, entreprenører og faghandel. Cordel holder til i Ålesund, og blir i dag brukt av over 1600 bedrifter (1). Gjennom Cordel fikk vi en kontaktperson som er bindeleddet mellom oss og bedriften. Kontaktpersonen for Cordel er Ole Johan.

Ole har vært lett tilgjengelig gjennom hele prosjektet, og har stilt opp på kort varsel. I tillegg har han hentet inn andre ansatte i bedriften som har bidratt med ulike tips om hvordan oppgaven kan forbedres og andre ting som bør gjøres.

Under prosjektets fremgang planla vi flere møter sammen med Ole i virksomheten. Vi satt opp tider og inviterte til møter, hvor vi informerte om hva som var gjort og videre diskuterte vi hva som skulle gjøres videre.

### 3.1.3 Veileder

Vi har i tillegg til en oppdragsgiver, fått en veileder fra NTNU. For oss er dette Anniken T. Susanne Karlsen. Hun har bidratt til at prosjektet har hatt fremgang og kommet med råd og tilbakemeldinger underveis i prosjektet. Anniken har også vært med på flere møter sammen med arbeidsgiver.

## 3.2 Prosjektplanlegging

I starten av prosjektperioden utviklet vi en forprosjektrapport (se Vedlegg 2)**.** Vi baserte rapporten på en mal utlevert av foreleser i forbindelse med den obligatoriske innleveringen gjennom et annet emne (IF300114) ved NTNU. Vi startet rapporten med informasjonsinnhenting om teknologier og teorier som vi brukte under programvareutviklingen. Videre definerte vi selve prosjektorganiseringen, deretter definerte vi hvilke avtaler som ble gjort med oppdragsgiver. Disse avtalene inneholder sikring for at vi jobber med riktig arbeidsmetodikk, arbeider konsistent og gruppenormer for arbeidsgruppen.

I det neste kapitelet i rapporten forklarer vi oppgaven i sin helhet. Da vi fikk utdelt oppgaven fikk vi tildelt en kort beskrivelse av problemstillingen og hva oppdragsgiver ønsket å få ut av oppgaven. Tidlig i prosessen brukte vi tid på å drøfte våre tanker om hvilke muligheter vi har i forhold til tidsomfang og kunnskap innenfor fagfeltet. I første møte med oppdragsgiveren diskuterte vi kravene til oppgaven. Ut ifra dette møte utredet vi en mer spesifikk og detaljert prosjektbeskrivelse, der vi beskrev hva oppgaven skulle gå ut på. Denne prosjektbeskrivelsen ble igjen diskutert med oppdragsgiver for å forsikre oss om at oppgaven var forstått riktig. I tillegg til beskrivelsen supplerte vi også med prosjektforklarende diagrammer som gjorde det lettere for oss som utviklere å jobbe med prosjektet, samt å forklare logikken bak prosjektet til andre.

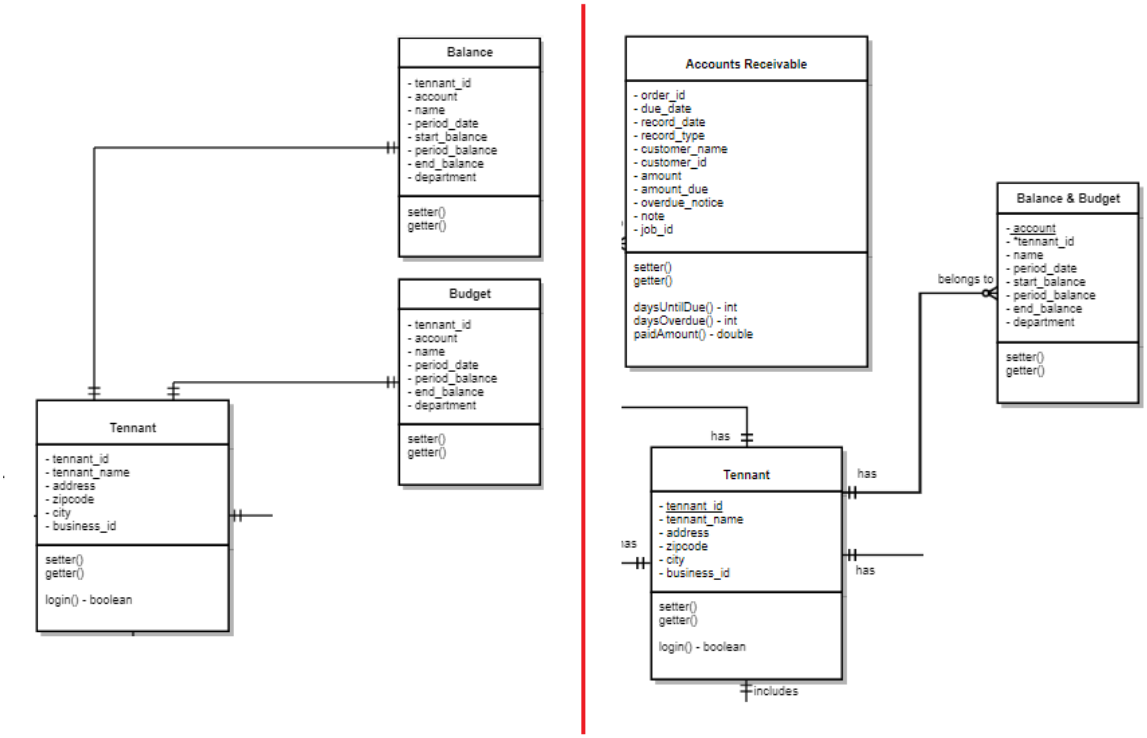
Vi laget så et aktivitetsdiagram som tar deg gjennom livet til systemet gjennom øynene til brukeren, fra du logger inn til du logger ut. Vi utformet også et UML-diagram viser programvare komponenter i et diagram, slik at du får vist det på en grafisk måte. Dette gjør at man enklere kan forstå sammenhenger i systemet og lettere finne feil.

Vi benyttet oss av wireframes for planlegging av design for både nettsiden og appen. Wireframes tillater oss enkelt å lage grafiske skisser for utseendet til app og nettside. Disse ble presentert og evaluert av arbeidsgiver og veileder før vi startet selve utformingen av nettsiden.

Vi fikk tildelt et dokument med vitale data som støpte grunnmuren for UML-diagrammet. Første utkast ble delt med både veileder og oppdragsgiver for tilbakemelding, og endringer ble gjort, klasser ble slått sammen og felt ble lagt til og fjernet. Etter å ha endret ble de igjen

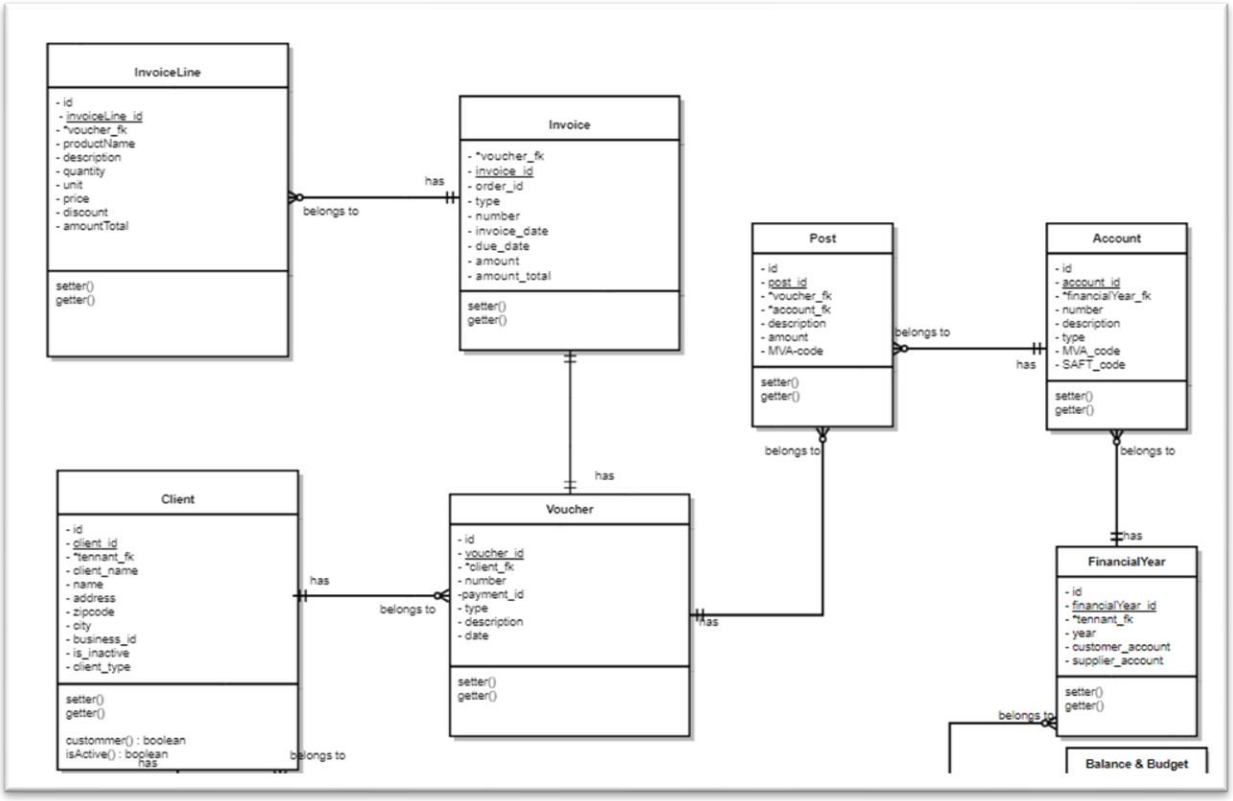
fremvist til veileder og oppdragsgiver, hvor ytterlige tilbakemeldinger ble gitt og endringer ble gjort. Disse endringene innebar blant annet rollebeskrivelser og markering av nøkler.

Nedenfor er to figurer som gir et lite innblikk i hvilke endringer som ble gjort i vårt UML-diagram. Alle disse endringene ble gjort i samsvar med arbeidsgiver sine preferanser. Begge bildene er bare et lite utklipp av UML-diagrammet, for å belyse endringene som er blitt gjort.




*Figur 3.1 Endringer gjort i UML-diagram. Bildet til venstre er fra 04.02.2021, bildet til høyre er fra 24.02.2021.*

På bildet over ser vi at Balance and Budget har blitt slått sammen, Account Receivable har blitt lagt til og relasjoner og modellbeskrivende ord er endret og lagt til.




*Figur 3.2, siste utkast, utklipp, UML-diagram (23.04.2021).*

Figur 3.2 viser siste utkast av UML-diagrammet. Her har det blitt lagt til flere nye modeller, InvoiceLine, Client, Voucher, Post, Account og FinancialYear. Arbeidsgiver har her kommet med ny informasjon om ønsket funksjon i applikasjonen, noe som har krevd endringer i datamodellen.

I forprosjektrapporten beskrev vi en fremdriftsplan med hovedaktiviteter og milepæler og vi fastsatte hvilke styrings- og utviklingshjelpemidler vi skal benytte oss av og hvordan vi skal bruke disse. Videre planla vi hvordan møtevirksomhet skal foregå med både veileder og oppdragsgiver, men også innad i gruppen. Som ledd i god praksis utviklet vi også en planlagt avviksbehandling. Avslutningsvis gikk vi gjennom utstyrsbehovet vi hadde for gjennomføringen av prosjektet.

## 3.3 Utviklingsmetodikk

Vi bestemte oss for å utvikle prosjektet med smidige metoder i form av Scrum (2.8.1). Dette innebærer at arbeidsmetodikken vår baserer seg på at vi jobber iterativt og inkrementelt. Utviklingen kan beskrives som iterativ da vi jobbet i sprinter og inkrementell fordi løsningen ble oppdatert til en ny versjon etter hver sprint.

Daglig scrum ble gjennomført hver arbeidsdag og sprintenes varighet ble satt til to uker. Sprintplanlegging, sprintgjennomgang og sprinttilbakeblikk har blitt gjort i tidsrommet mellom avsluttet sprint og opprettelse av ny sprint. Rollen som Scrum Master ble rullert på annenhver sprint.

### 3.3.1 Proof of concept

Proof of concept (PoC) er en metode for å visualisere om et produkt er gjennomførbart og levedyktig. Er dette noe en bruker faktisk er villig til å bruke penger på (54)? Vi brukte dette konseptet som en start av utviklingen, for å kunne vise arbeidsgiver hvordan vi så for oss en løsning, i tillegg til å lære oss nye teknologier. Tre reposotories i Github ble opprettet for utvikling av PoC, et for backenden, et for nettsiden og et for mobilapplikasjonen. Når vi var ferdig med PoC, viste vi frem våre resultat for arbeidsgiver for å få tilbakemelding og deretter opprette nye repositiories i Github. Her ble noe av koden fra PoC-en hentet for gjenbruk i den faktiske løsningen.

### 3.3.2 Minimal viable product

Minimal viable product (MVP), beskrives som nøkkelfunksjonaliteten som muliggjør at produktet kan brukes, uten ekstra funksjonalitet. MVP er en glimrende måte å teste en ide mot brukere uten å besitte store ressurser. Dette reduserer risiko og er derfor en fornuftig tilnærming til produktutvikling. Likevel er ikke MVP et halvferdig prosjekt, men heller et prosjekt som er akkurat nok for at kunden skal få nytte ut av det (55).

Vårt prosjekt kan ansees å være en MVP. Vi har fokusert på å lage en god grunnmur for videre utvikling, uten store kostnader for Cordel. Dette har under hele prosjektet vært vårt og Cordel sitt mål.

## 3.4 Teknologier

Gjennom dette prosjektet har vi tatt i bruk flere ulike teknologier for å hjelpe oss med å løse oppgaven. Mange av disse teknologiene er tatt i bruk etter krav fra oppdragsgiver.

### 3.4.1 Git og GitHub

Ettersom vi er flere som har jobbe på samme prosjekt valgte vi å bruke Git. Gjennom studietiden ved NTNU er det dette versjonskontrollverktøyet vi har brukt, noe vi har fått god erfaring med.

Git er et versjonskontrollverktøy som blir mye brukt i sammenheng med systemutvikling (56). Ved å bruke Git ble det enkelt for oss å dele, slå sammen eller hente kode fra oss som jobber sammen. Endringer som blir gjort underveis blir publisert med en commit som inneholder en beskrivende tekst alt etter hvilke endringer som har blitt gjort. Det blir da enkelt for de andre gruppemedlemmene å se hva som er gjort, og en kan enkelt gå tilbake til tidligere løsninger dersom det skulle oppstå problemer underveis. Hvert medlem har ofte jobbet i en egen branch hvor en spesifikk oppgave har blitt løst. Det har gjort det mulig for oss å jobbe parallelt med prosjektet samtidig som det er enkelt å gå fra en versjon til en annen (57).

GitHub er en skytjeneste for versjonskontroll som vi har tatt i bruk under utvikling av prosjektet (58). Dette verktøyet har gjort det enklere for prosjektgruppen å kunne utvikle sammen. I tillegg til at det blir enklere å utvikle sammen, kan prosjektet lastet opp på GitHub hentes av andre brukere. Koden er derfor enkel å få tak i for både oppdragsgiver og veileder.

### 3.4.2 PostgreSQL

PostgreSQL er et databasesystem for relasjonsdatabaser og har en åpen kildekode. Databasesystemet er en utvidet versjon av SQL-språket. (59). PostgreSQL er noe hele prosjektgruppen har benyttet seg av tidligere og er derfor godt kjent med. Produktet vi har laget består av to forskjellige databaser som også snakker sammen. En egen database blir brukt for innlogging og en annen for selve datavarehuset.

### 3.4.3 Docker

Dockercontainere gjorde det enkelt for oss å distribuere elementære deler av prosjektet innad i gruppen. For å konfigurere containerne satte vi opp en docker-compose.yml fil. Der opprettes to containere av samme bilde med ulike konfigurasjoner for våre databaser.

Docker består av bilder og containere. Bilder er instruksjoner til hvordan en dockercontainer blir laget. En container blir definert av et bilde og er et kjørbart miljø for et bilde. Hver container er isolert fra omgivelsene rundt seg og kjører derfor i sitt eget miljø. Containere kan

bli flyttet til andre klientmaskiner og kjøres uten at innholdet endrer seg (60). PostgreSQL kjører vanligvis på port 5432. Vi har brukt en ekstra port, og har derfor vært nødt til å distribuere en ekstra port for den andre databasen.

### 3.4.4 Postman

For testing av API-kallene våre brukte vi Postman. Postman er ett av flere verktøy for å teste forskjellige endepunkt for en API (2.5.3). Ved hjelp av dette verktøyet kan vi definere hvilken type forespørsel vi sender, og til hvilket endepunkt (61). Grunnen til at dette er viktig er fordi den samme URL-en kan inneholde flere endepunkt, for eksempel en for å legge til data og en for å hente data. Samtidig som at Postman løser problemer for oss, har Postman fordeler som lagring av forskjellige kall, gode debugging-funksjoner og muligheten for å lage tester.

### 3.4.5 Microsoft Teams og Discord

På grunn av koronasituasjonen har det vært noe problematisk å møtes fysisk. Vi har derfor vært nødt til å bruke noen alternative metoder. Møter sammen med arbeidsgiver og veileder har blitt gjort med bruk av Microsoft Teams, med unntak av et par fysiske møter, når smittesituasjonen tillot det. Microsoft Teams har vært enkelt å bruke da det er noe både prosjektgruppen, arbeidsgiver og veileder er vant med å bruke fra før.

Gjennom hele prosjektet brukte vi kommunikasjonsverktøyet Discord. Der opprettet vi en egen server hvor vi enkelt kunne dele informasjon i ulike tekstkanaler, snakke sammen og dele skjerm. Med muligheten for skjermdeling på en rask og enkel måte, ble parprogrammering (2.8.2) enkelt og effektivt å bruke under utvikling.

## 3.5 Utviklingsverktøy

Ved valg av utviklingsverktøy hadde oppdragsgiver noen overordnede forutsetninger som påvirket valget av hvilke utviklingsverktøy vi skulle ta u bruk. Vi vil i dette kapitelet gå gjennom de forskjellige verktøyene vi valgte, og forklare litt om disse

### 3.5.1 Visual Studio Code

For utvikling av både BI-systemet (2.3) og API-et (2.5.3) har vi brukt Visual Studio Code (VSC). VSC er en editor med innebygd støtte for forskjellige språk, inklusiv TypeScript (3.6.2), men også utvidelser for andre språk som for eksempel C# (3.6.4) (62).

### 3.5.2 Android Studio

For utvikling av mobilapplikasjonen brukte vi Android Studio. Android Studio er en IDE brukt for applikasjonsutvikling (63). Alle i prosjektgruppen har tidligere erfaring med Android Studio, noe som førte til at det ble et naturlig valg for oss.

### 3.5.3 DataGrip

For å enkelt kunne endre og visualisere våre databaser brukte vi DataGrip. DataGrip er et databasestyringsverktøy for utviklere. Det er laget for å gi brukeren muligheten til å søke, opprette og administrere databaser og tabeller. Datagrip støtter flere typer databaser, inkludert PostgreSQL (64). Alle på gruppen har brukt denne programvaren tidligere, og er noe vi er komfortable med.

## 3.6 Programmeringsspråk

Etter samtaler med Cordel kom det tydelig frem hvilke programmeringsspråk de ønsket at vi skulle benytte oss av. Cordel skal videreutvikle prosjektet når vi er ferdige, og ved å benytte oss av deres ønskede programmeringsspråk, vil overgangen fra oss til dem være enklere.

### 3.6.1 Flutter

Flutter er et programvareutviklingssett (SDK) som blir brukt til å utvikle både mobile applikasjoner for iOS og Android, og web applikasjoner for Windows, Mac og Linux. Programmeringsspråket som blir brukt for flutter heter Dart (65).

### 3.6.2 TypeScript

For å innføre funksjonaliteten nettsiden trenger for å lage et BI-System til Cordels brukere, brukte vi TypeScript. Typescript er et superset av Javascript, som kompileres til Javascript når det kjøres. Typescript bruker typesterkt system som kan definere typene for å gjøre det lettere å dokumentere, validere og finne feilkode. Typescript hjelper også til med å gjøre prosjektet skalerbart, noe som er en fordel når prosjektet skal videreutvikles (66).

### 3.6.3 CSS

For å få nettsidene til å være responsiv og se bedre ut har vi tatt i bruk Cascading Style Sheets (CSS). Ved bruk av CSS kan vi definere flere sett med regler for hvordan forskjellige elementer på nettsiden skal se ut (67). På denne måten kan man oppnå ønsket plassering, størrelse, farge, bakgrunn o.l. til alle filer som peker til CSS-filen(e). Flere av nettsidens

endepunkt kan benytte seg av samme CSS-fil, slik at vi slipper å definere designet på nytt for hvert endepunkt.

### 3.6.4 C# og .NET

Etter ønske fra arbeidsgiver tok vi i bruk programmeringsspråket C# for vår backend API. C# er et objektorientert språk som har likheter med andre språk som C, C++ og Java (68). Gjennom studiet har vi jobbet mye med Java, noe som har gjort overgangen til C# enklere. I startfasen av prosjektet var språket noe uvandt, men med tiden ble det lettere å knytte C# opp mot våre tidligere erfaringer med Java. For store deler av backenden har vi benyttet oss av C# og deres biblioteker og rammeverk, som blir utdypet i kapittel 3.7.

## 3.7 Eksterne biblioteker og rammeverk

Under får man en oversikt over ulike biblioteker og rammeverk som vi har brukt utover prosjektet. For å skape en bedre oversikt uten å forlenge delpunktene har vi satt de første delpunktene fram til 3.7.10 for Backend og de resterende delpunktene henviser til frontend.

### 3.7.1 Entity Framework Core

Entity Framework Core (EFCore) har hatt en sentral rolle under utviklingen av backenden vår, da vi hele tiden har hatt behov for å snakke med databasene. EFCore er et rammeverk utviklet av Microsoft og er en object-relational mapper (ORM) og et open-source rammeverk som gjør det enkelt å jobbe med databaser med bruk av ulike .NET objekter og migrasjoner. EFCore bruker Language Integrated Query (3.7.2), som gjør at man slipper vanlige SQL-spørringer (69).

### 3.7.2 LINQ

Når vi bruker EFCore tar man også i bruk Language Integrated Query (LINQ). LINQ er SQL-spørringer som er integrert i språket C#. Vanlige SQL spørring består av strenger med vanlig tekst for å kunne kommunisere med en database og kan enkelt skrives feil. Ved å ta i bruk LINQ slipper man disse spørringene, og det blir istedenfor utført som vanlig kode med nøkkelord som er opplysende og nøyaktige i forhold til hva den gjør. LINQ er veldig lett å lese og har en syntaks som gjør det mulig å utføre flere operasjoner på samme objekt med et minimum av kode (70). LINQ har også innebygd sikkerhet ved å overføre data til databasen som SQL parametere (71,72).

### 3.7.3 ASP.NET Core MVC

For å bygge API-et ved Model-View-Controller designet tok vi i bruk ASP.NET Core MVC. Rammeverket inkluderer flere nyttige løsninger som vi brukte under utvikling. Routing gjør det mulig for en kontroller å ha flere endepunkter med en definert attributt (73). På den måten var det mulig for oss å spesifisere endepunktene på en god og oversiktlig måte slik at det ble enkelt å bruke for spørringene i frontend. For å ta enda mer nytte av rammeverket tok vi også i bruk modellvalidering og filtrering. Valideringen er dataannoteringer som definerer krav til felt, for eksempel nødvendige felt i en modell. Filtrering blir brukt for å blant annet begrense tilgang til metoder og endepunkt, men mer om det blir nevnt i 3.7.4 (73).

### 3.7.4 ASP.NET Core Authorization

Som nevnt delkapittel 3.7.3 har vi tatt i bruk autorisering for å begrense tilgangen til enkelte brukere ved å ha to typer brukere. ASP.Net Core Authorization gjør det mulig å deklarere spesifikke roller («User» og «Admin») og/eller legge til spesifikke regler for bruk av metoder eller modeller. (74) Ved å bruke autoriserings attributtet til ASP.Net Core MVC er det mulig å definere hvilke roller som har tilgang til en metode. Metoder som har definert en autoriserings-rolle vil da kun være tilgjengelig for den rollen. (75)

### 3.7.5 Newtonsoft

For oss var det nødvendig å kunne konvertere .NET objekter og JSON objekter om hverandre. Det er fordi systemet får tilsendt data som JSON, som videre blir behandlet med EFCore, før det blir lagt til i databasen. For å behandle dette, konverteres JSON dataen om til .NET objekter, og da tok vi i bruk rammeverket Newtonsoft. Newtonsoft er et JSON rammeverk brukt i .NET applikasjoner for å konvertere mellom JSON og .NET objekter (76).

### 3.7.6 Xunit

For unit testing (2.11.1) valgte vi å ta i bruk rammeverket Xunit. Vi så vi på to forskjellige rammeverk for unit testing, Nunit og Xunit. Valget endte på Xunit da dette føltes noe mer moderne og mer optimalisert (77). For å teste våre metoder på best mulig måte, kjørte vi isolerte tester med å ta bruk blant annet Moq (3.7.7).

### 3.7.7 Moq

For at testene våre skulle bli testet i et isolert miljø tok vi i bruk rammeverket Moq. Ved å bruke dette rammeverket ble testene isolert og var derfor ikke avhengig av databaser eller

ekte resultater. Moq tillater mocking av data slik at en kan simulere ekte objekter og metoder. Dette gjøres ved bruk av en Setup metode Moq tar i bruk. Det blir deretter gjort et metodekall som da returnerer en gitt verdi i testklassen. På den måten defineres en metode definert i testmetoden uten at den må ha direkte kontakt med selve applikasjonen (78).

### 3.7.8 BCrypt

Å behandle brukere med brukernavn og passord krever at en tenker på sikkerhet. Som nevnt i 2.9.1 og 2.9.2 så kan man hashe passord sammen med en salt for å gjøre en hash unik. BCrypt løser dette ved å legge en salt til input verdien i en metode. Når en bruker skal verifiseres i vårt system vil den sjekke input med lagret hash verdi for å godkjenne brukerne. En ny salt blir generert ved hvert kall, noe som fører til at alle passord blir unike.

### 3.7.9 User secrets

Vi ønsket ikke å ha databasens tilkoblingsstreng direkte inn i koden. User secrets er en funksjon i ASP.Net Core applikasjoner som gjorde det mulig for oss å lagre informasjon utfor selve prosjektet i en Json fil (2.5.1). Ved oppretting av en user secret genereres en unik GUID som legges til i .csproj filen i prosjektet. Alle user secrets vil være koblet opp mot den unike GUIDen for videre bruk (79).

### 3.7.10 Swagger

Swagger er et rammerverk som hjelper utviklere med å enkelt kunne automatisere arbeid som API dokumentasjon og kode generening. Swagger vil autogenere en HTML-side, og enkelt visualisere Api-et som et JSON object (80).

### 3.7.11 Material.ui

Material UI er et bibliotek som lar brukerne ta i bruk deres ferdiglagde komponenter (81). Komponentene er laget med material design, og disse fungerer som byggeklosser for utviklere til å bygge et brukergrensesnitt for Android, iOS, Flutter og web.

### 3.7.12 Formik & Yup

Formik er lite et bibliotek som kan tas i bruk dersom nettsiden din inneholder skjemaer som skal fylles ut.  Formik gjør ikke alt for mye av seg selv, men gjør det lettere som utvikler å håndtere dataen som blir skrevet inn i skjemaene (82).  Yup er et annet bibliotek som ofte brukes sammen med formik.  Ved bruk av Yup blir det enklere å skrive kode for å validere det som blir fylt ut, samtidig som man kan lage tilpassede feilmeldinger for hvert felt (83).

### 3.7.13 Recharts

Recharts er et bibliotek som tilbyr grafisk fremstilling av data. Diagramtypene som tilbys er Linjediagram, arealdiagram, søylediagram, spredningsdiagram og kakediagram. Diagrammene har en enkel men fin utforming og snakker godt med en API. Recharts har innebygde mekanismer for å håndtere skalering og responsitivitet slik at dette skjer automatisk (84).

### 3.7.14 Axios

Axios er en http-basert klient for nettsider og node.js som lar oss gjøre http-spørringer fra nettsider. Dette i seg selv er ikke noe banebrytende teknologi, men Axios gjør dette på en måte som er enklere for utviklere å jobbe med. Istedenfor å måtte kalle json på svaret fra http-kallet, får man den faktiske dataen direkte ved bruk av Axios. I tillegg til denne mer eller mindre trivielle forbedringen, er errorhåndtering ved feil i et http-kall enklere å fange ved bruk av Axios, i forhold til andre alternativer (85).

### 3.7.15 React

React er et Javascriptbibliotek skapt av ingeniører i Facebook, til å løse problemer med utvikling av komplekse og dynamiske brukergrensesnitt med datasett som endres over tid. React prøver å løse dette ved å tilby en solid abstraksjon for utviklingen. React blir brukt som et utgangspunkt i utviklingen av enkelt-side, og er optimalisert for å hente skiftende data som må registreres. Ved utvikling av en mer kompleks React-applikasjon kreves det vanligvis at du bruker ekstra biblioteker for tilstandsstyring, ruting og interaksjon med API (86).

### 3.7.16 React bootstrap

React bootstrap er et bibliotek for komponenter og knapper. Biblioteket tilbyr en rekke forskjellige komponenter med responsive, elegante og estetisk fine trekk. Komponentene kommer med mange tilpasningsmuligheter, som gjør det lett å jobbe med (87).

### 3.7.17 React router

React router, også kalt React router dom, er et verktøy som håndterer rutingen av en nettside. Fordelen ved å bruke dette biblioteket fremfor eldre rutearkitektur, er at React router bruker dynamisk ruting. All ruting foregår samtidig som applikasjonen prosesseres på din datamaskin, i motsetning til eldre metoder hvor rutingen foregår i konfigurasjoner på utsiden av applikasjonen (88).

### 3.7.18 React testing library

React testing library er et bibliotek som oppfordrer til bedre praksis i testing av React applikasjoner. Hovedprinsippet for biblioteket er å gjøre testene så like som den faktiske måten programvaren er ment til å bli brukt. React testing library bruker instanser av React komponenter istedenfor faktiske DOM noder. Måten biblioteket tester en applikasjon er at det blir gjort spørringer på en DOM opp imot merkelappteksten akkurat slik en bruker ville gjort det. For komponenter uten en logisk merkelapp, blir det laget en «data-testid» som blir brukt for å referere til komponenten i testing.

### 3.7.19 Shared Preferences

Shared preferences blir hovedsakelig brukt for lagring av permanente nøkkelverdier. Disse verdiene kan bli brukt til alt fra oppstartsinformasjon til tokens for innlogging. Noen tilbaketrekk med Shared Preferences er at det kun kan lagres verdier av primitive datatyper som *Int, double, bool, string* og *stringList.* Det er heller ikke ment for lagring av store datamengder (89).

### 3.7.20 Convert

*Dart:Convert* er bibliotek som gjør det mulig å enkelt dekode og kode innkommende data for å konvertere mellom flere typer data som for eksempel fra Json (2.6.1) til en liste med egendefinerte objekter (90).

### 3.7.21 Flutter Spinkit

*Flutter:Spinkit* er et bibliotek med en samling av mange forskjellige animasjoner på ikoner som brukes for å vise brukeren at noe laster inn (91). Dette biblioteket ble tatt i bruk ved bruk av «loading»skjermen brukt i mobilapplikasjonen.

### 3.7.22 Async

Biblioteket *Dart:async* gir muligheten for å kunne programmere asynkront ved hjelp av klasser som Future og Stream. Future, som er brukt i dette prosjektet, representerer et objekt som skal bli returnert fra en klasse, men som ikke har blitt utført ennå, som er forventet skal bli gjort. Klassen vil returnere en verdi i fremtiden når den er ferdig med oppgaven sin. Å vente på svar på en GET forespørsel fra en server er typisk noe som trenger å bli kjørt asynkront, siden det kan ta tid før programmet får svar og kan gå videre (92).

### 3.7.23 Local storage

Local storage fungerer som nettsidens database og holder på informasjon lokalt i minnet til brukeren. Brukeren klarer da å holde seg innlogget mellom sideoppdateringer (93).Dette blir ofte brukt i sammenheng med en Json Web Token hvor man lagrer tokenen tilhørende den innloggede brukeren.

## 3.8 Model-view-controller

Model-view-controller (MVC), er et konsept som tar for seg arkitekturen til en programvare. Designmønsteret MVC er en av de mest brukte konseptene (94), som er hovedgrunnen for vårt valg av designmønster. MVC er teknisk sett kun et teoretisk konsept, og vil i praksis fungere på mange forskjellige måter. En av grunnene for at MVC ofte blir brukt i web-applikasjoner er at det separerer funksjonaliteten og logikken i en applikasjon. Dette tillater at programmeringen blir enklere å organisere når et helt team jobber sammen på samme prosjekt.

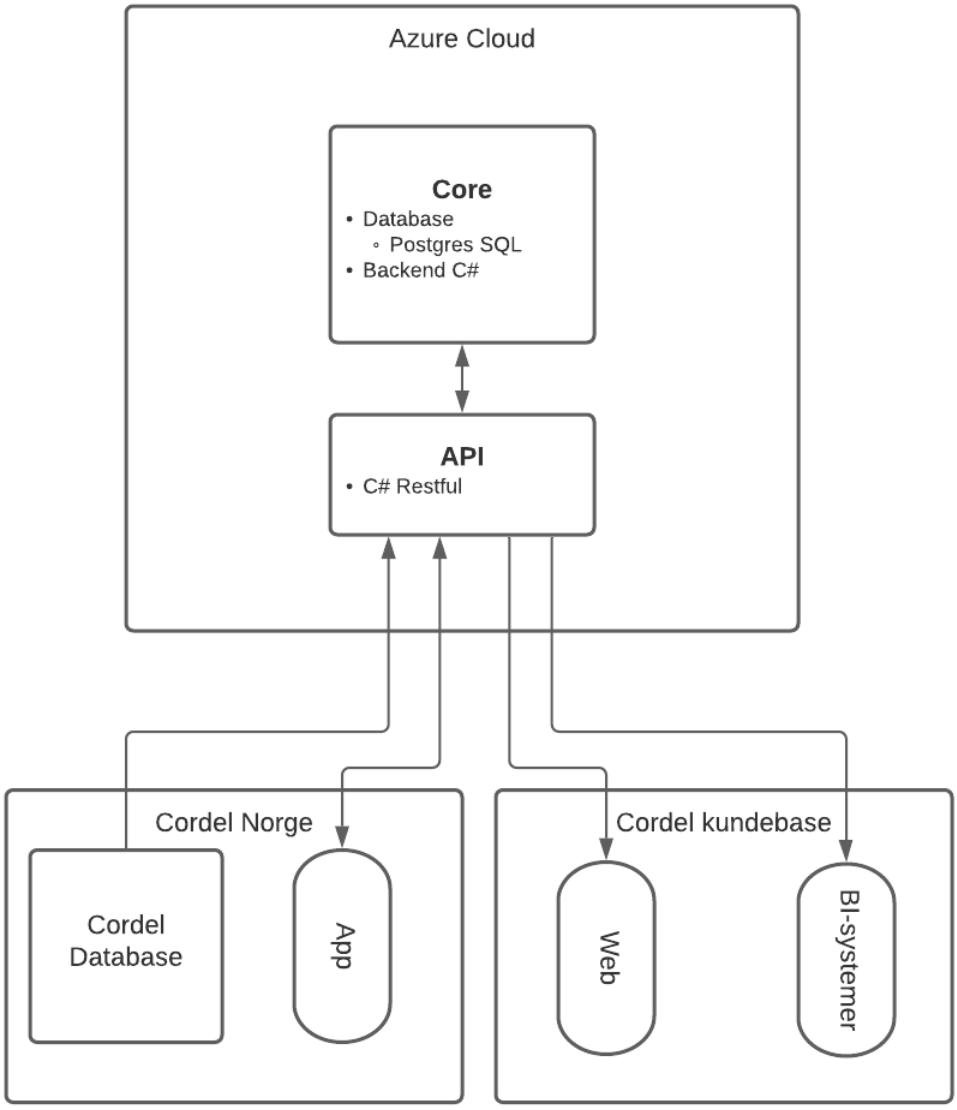
MVC fungerer slik at man deler systemet i tre deler, data (model), brukergrensesnitt (view) og en kontroller. Model er det som holder på dataen og vil snakke med databasen. View viser frem data til brukeren. Kontrolleren er den som flytter dataen rundt i systemet. Dette fører til at man prøver å unngå at modellen snakker direkte med viewet, men alltid går igjennom kontrolleren (94).

# 4 RESULTATER

I dette kapittelet går vi gjennom byggesteinene vi har brukt i utviklingen av vårt prosjekt. Vi skal først skape en oversikt over sammenhengen mellom API-et og de to frontendene, hvordan det henger sammen, for så å gå inn på detaljene som ligger bak hele prosjektet.

## 4.1 Systemoversikt

Figur 4.1 forklarer sammenhengen mellom Cordel klienten, API-et og de to frontendene. I tillegg illustreres hvordan de ulike delene kommuniserer med hverandre.

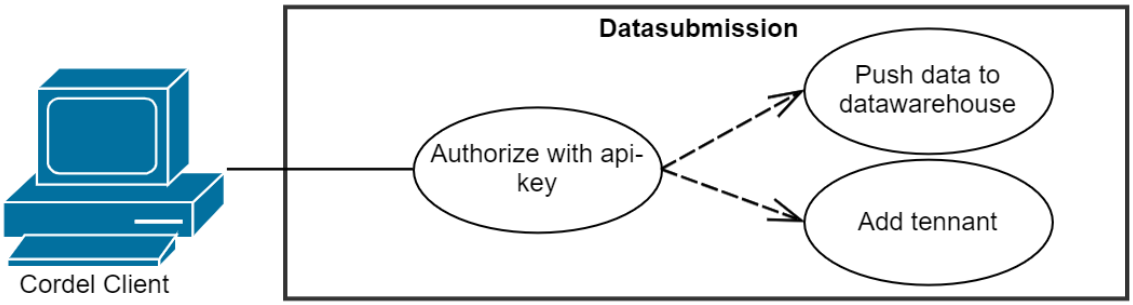



*Figur 4.1 Systemforklarende diagram*

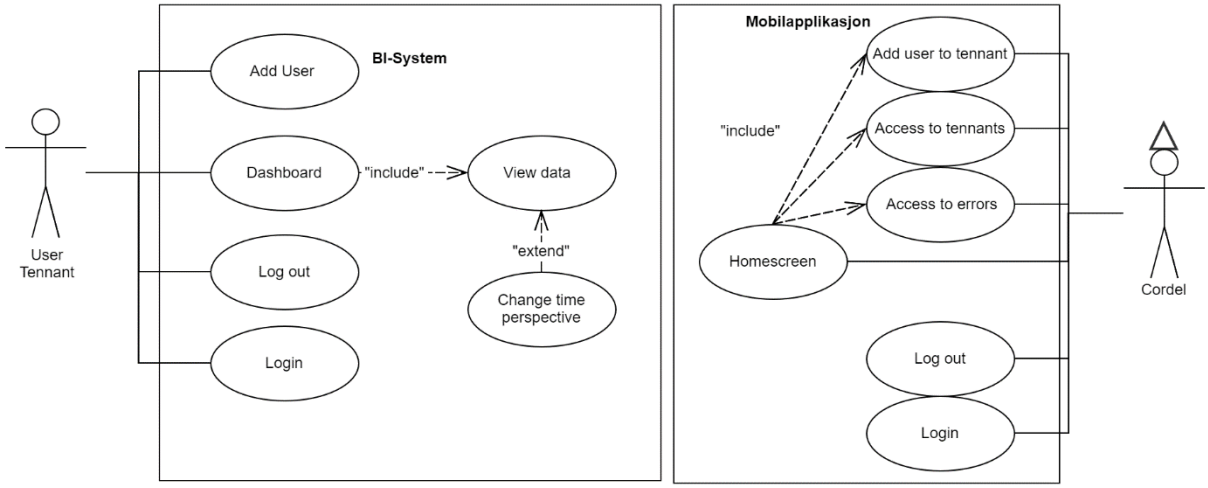
I kjernen av systemet ligger databasen, som skal kjøres i Cordels Azuremiljø. Her blir all informasjon lagret og knyttet sammen ved hjelp av PostgreSQL (3.4.2). All form for kommunikasjon til kjernen går gjennom API-et (2.4.2). API-et består av kontrollere (4.4.3) for hver applikasjon med kontakt til kjernen (se figur 4.1) for å kontrollere informasjonen som går ut og inn.

## 4.2 Use Case

I figur 4.2 og figur 4.3 viser vi et bruksmønsterdiagram som vi har laget for de forskjellige bruksområdene til systemet.




*Figur 4.2 Klient som sender data til datavarehuset*




*Figur 4.3 Use Case diagram*

Det er 3 bruksområder for dette systemet:

1. **Innsending av data.**

   Ved overføring av data vil Cordel sine systemer automatisk overføre informasjon til databasen regelmessig. Dataen som overføres er enten oppretting av en tennant eller informasjon om tennants.

2. **BI-System via nettleser.**

   BI-systemet er designet for kundene. Her kan de logge seg inn for å få et overblikk over sin bedrift ved bruk av grafer og prosessert data fra datavarehuset.

3. **Overvåkning og administrering via mobil applikasjon.**

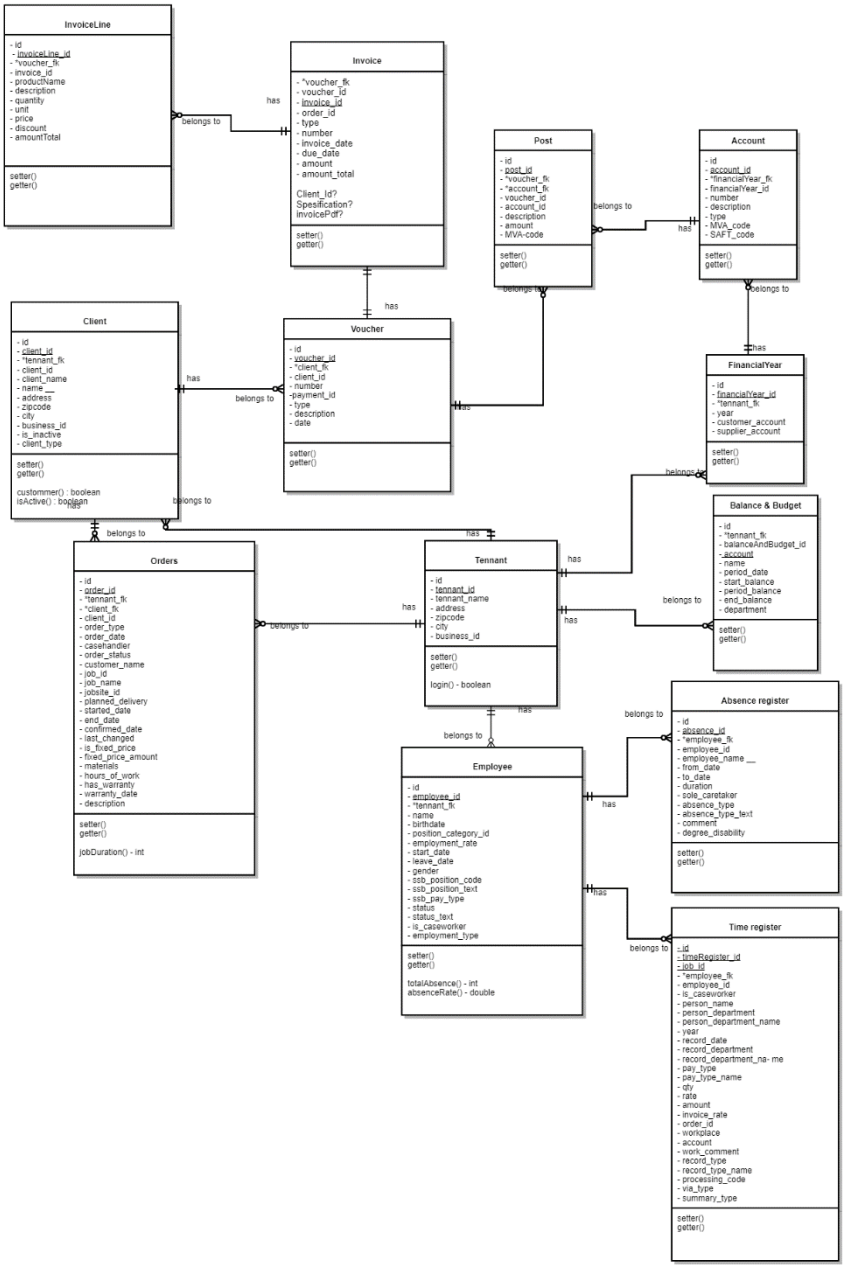
   Gjennom den mobile applikasjonen har Cordel overblikk over hvem som benytter seg av systemet. I tillegg kan de se feilmeldinger fra serveren, samtidig som de har mulighet til å initialere første bruker for en tennant.

## 4.3 Backend

Vi har laget systemet slik at brukerne ikke har direkte kontakt med det som ligger i backenden. Her lagrer og manipulerer vi informasjonen slik at de er forståelig for frontend-applikasjonene som benyttes av brukerne. Videre i rapporten går vi inn på systemets backend for å se hvordan ting henger sammen og hva som skjer når frontend-applikasjonene kontakter backenden.

### 4.3.1 Datamodell

I utviklingen av prosjektet har vi gjentatte ganger gjennomgått endringer av UML-diagrammet for databasen (3.2). Arbeidet med datamodellen ble omfattende, da det var veldig mange forskjellige attributter og komplekse sammenhenger. Figuren 4.3 er vårt sluttresultat av UML-diagramet.




*Figur 4.3 UML-diagram av datamodellen*

Diagrammet består av mange entiteter som inneholder mye informasjon, for å få en bedre oversikt av hva figuren viser skal vi videre i rapporten beskrive de viktigste trekkene for entitetene.

#### 4.3.1.1 Entitetsforhold

Alle relasjonene i databasen består av en-til-flere (2.1.2) forhold, med unntak av faktura og bilag. Disse entitetene har et en-til-en-eller-null-forhold (2.1.2). Derfor kan vi lage et bilag uten en faktura, mens en faktura ikke kan lagres uten å knyttes opp til et bilag.

#### 4.3.1.2 Identitetsnøkler

Oppdragsgiver ønsket at vi skulle benytte oss av egne identitetsnøkler (ID) i databasen, samtidig som vi lagrer deres ID-er. På denne måten kan vi bruke våre ID-er ved relasjonsetablering og spørringer. Samtidig kan Cordel bruke sine ID-er ved innsending av data slik at vi vet hvilke entiteter de referer til, uten at de trenger innsyn i vår database. Derfor vil hver entitet i databasen være lagret med en ID generert av oss og en «entitetsnavn_id» som Cordel sender til oss. Det eneste unntaket for dette er entiteten *invoice* (faktura), som bruker *voucher* (bilag) sin ID og lagrer det som «voucherFK». Dette er grunnet deres relasjon, nevnt i 4.3.1.1.

#### 4.3.1.3 Tennant-entiteten

Tennant-entiteten fungerer som et startpunkt for all fremtidig informasjon som skal lagres i datavarehuset. Derfor vil alle entiteter i databasen underligge en spesifikk tennant gjennom en eller flere relasjoner, se figur 4.3. Tennant-modellen i seg selv består av en ID, API-nøkkel, et navn og et organisasjonsnummer.

#### 4.3.1.4 Context

Fordi vi har to databaser, har vi konfigurert databasene med to context-klasser. De bruker DbContext fra EFCore (3.7.1) og er definert i klassene «WarehouseContext» og «LoginDatabaseContext» i vårt system.

```
3 references
public DbSet<AbsenceRegister> AbsenceRegisters { get; set; }
3 references
public DbSet<BalanceAndBudget> BalanceAndBudgets { get; set; }
5 references
public DbSet<Client> Clients { get; set; }
6 references
public DbSet<Employee> Employees { get; set; }
4 references
public DbSet<Invoice> Invoices { get; set; }
4 references
public DbSet<Order> Orders { get; set; }
6 references
public DbSet<Tennant> Tennants { get; set; }
4 references
public DbSet<TimeRegister> TimeRegisters { get; set; }
4 references
public DbSet<ErrorLog> ErrorLogs { get; set; }
2 references
public DbSet<InvoiceLine> InvoiceLines { get; set; }
4 references
public DbSet<Voucher> Vouchers { get; set; }
2 references
public DbSet<Post> Posts { get; set; }
3 references
public DbSet<Account> Accounts { get; set; }
3 references
public DbSet<FinancialYear> FinancialYears { get; set; }
```

*Figur 4.7 Definerte tabeller i klassen WarehouseContext*

For å forhindre lav kohesjon (2.4.1) utviklet vi spørringene i sin representerte context klasse med bruk av LINQ (3.7.2). Spørringene er enkle og oversiktlig, slik at det blir vanskeligere å gjøre menneskelige feil med vanlige SQL spørringer. Se figur 4.8 for eksempel på en spørring.

```
public List<AbsenceRegister> getAllAbsenceFromDate(long tennantId, DateTime comparisonDate)
{
    var absence = AbsenceRegisters
    .Where(i => i.employee.tennantFK == tennantId)
    .Where(d => d.fromDate >= comparisonDate)
    .OrderBy(d => d.fromDate)
    .ToList();
    return absence;
}
```

*Figur 4.8 En databasespørring i WarehouseContext*

### 4.3.2 View models

Modellene som ligger i datavarehuset, består av mye informasjon. Mye av informasjonen som blir lagret i modellene brukes ikke av frontend-applikasjonene og kan anses som overflødig. Derfor har vi laget view models for modellene. View models består kun av den informasjonen som de forskjellige applikasjonene trenger, og blir ikke lagret i databasen. En slik løsning hindrer at overflødig informasjon blir sendt og det blir enklere å håndtere ferdigsortert data i frontenden. Figur 4.9 viser et eksempel på en av view modellene som er brukt.

```
namespace Datawarehouse_Backend.ViewModels
{
    21 references
    public class AbsenceView {
    14 references
    public int year {get; set;}
    14 references
    public int month {get; set;}
     5 references
     public int day  {get; set;}
    8 references
    public string weekDay {get; set;}
    14 references
    public double totalDuration {get; set;}
    }
}
```

*Figur 4.9 ViewModel for Absence*

### 4.3.3 Docker

Som nevnt i kapittel 3.4.3 tok vi i bruk Docker. Figur 4.10 viser docker-compose.yml filen som er brukt for å konfigurere containerne med databasene. Disse containerne er i samme nettverk og har derfor muligheten til å kommunisere sammen. Vi benytter oss av port 5432 og 5433 for våre databaser. For å løse det brukte vi en command i docker-compose-filen som endrer porten til 5433, som vist i figur 4.10.

```
version: '3'
services:
  logindb:
    container_name: "Logindatabase"
    working_dir: /Logindb
    env_file:
      - DockerFiles/Login.env # Configure postgres
    volumes:
      - database-data:/var/lib/postgresql/logindata/ # Persist data even if container shuts down
    ports:
      - 5432:5432

  warehousedb:
    container_name: "Datawarehouse"
    working_dir: /Warehouse
    env_file:
      - DockerFiles/Warehouse.env
    volumes:
      - datawarehouse-data:/var/lib/postgresql/warehousedata/
    ports:
      - 5433:5433
    command: -p 5433

volumes:
  database-data:
  datawarehouse-data:
```

*Figur 4.10 Docker-compose.yml*

Informasjon som innlogging til databasene lagret vi som en egen fil. Dette blir gjort istedenfor å hardkode det direkte inn i docker-compose-filen. Informasjonsfilen blir ikke delt på git (3.4.1) for å forhindre at sensitiv informasjon blir tilgjengelig for utenforstående. Figur 4.11 viser filen som blir brukt for datavarehuset. Denne definerer brukernavn, passord og databasenavn til datavarehuset.

```
# Warehouse.env
POSTGRES_USER=admin
POSTGRES_PASSWORD=admin
POSTGRES_DB=datawarehouse
```

*Figur 4.11 Datavarehus variabler for innlogging til datavarehuset*

### 4.3.4 Autentisering

Datavarehuset inneholder bedriftshemmelig informasjon om flere tennants. Det er derfor viktig at alle forespørsler autentiseres, slik at tennants kun får tilgang til informasjon om sin egen bedrift. For å forklare hvordan autentiseringen er bygd opp, går vi først gjennom «User» modellen, for så å forklare hvordan vi har brukt «roller» i prosjektet vårt.

#### 4.3.4.1 User

User-modellen er for brukere av systemet. En bruker består av en ID, e-post, passord, rolle og et forhold til en bestemt tennant (4.3.1.3). Forholdet mellom bruker og tennant bestemmer hvilken bedrift brukeren tilhører, som igjen bestemmer hvilken informasjon brukeren har tilgang til.

#### 4.3.4.2 Roller

Rollene som bruker-modellen benytter seg av har gjennom prosjektet vært noe omdiskutert. Dette med tanke på hvilke roller som skal eksistere og hvordan de skal benyttes. Siden prosjektet anses å være en MVP (3.3.2), kom vi sammen med Cordel frem til at det bare trengs to roller, User og Admin. Userrollen er for alle kundene som har opprettet en bruker for dette systemet. Adminrollen er kun for Cordel's ansatte. Brukere med rollen Admin vil ha tilgang til administrative kall som for eksempel mobil-applikasjonen benytter seg av.

#### 4.3.4.3 Json Web Token generator

For å finne ut hvilken informasjon som skal hentes benyttet vi oss av Json Web Tokens (JWT) (2.6.2). Vår JWT-token er bygd opp av flere claims som inneholder informasjon om Tennant (4.3.1.3), User (4.3.4.1) og Role (4.3.4.2), vist i figur 4.12. Dette er data som lagres i payloaden og brukes for verifisering av brukere. Claim nummer en, som er «tennantFK», er kanskje den viktigste siden den sier hvilken tennant brukeren er registrert under.

```
var claims = new[]
{
    new Claim(JwtRegisteredClaimNames.Sub,userinfo.tennantFK.ToString()),
    new Claim(JwtRegisteredClaimNames.Email,userinfo.Email),
    new Claim(ClaimTypes.Role, userinfo.role),
    new Claim(JwtRegisteredClaimNames.Jti,Guid.NewGuid().ToString())
};
```

*Figur 4.12 Kodeutsnitt for generering av JWT.*

### 4.3.5 StartUp.cs

Startup klassen utfører konfigureringer og er nødvendig for alle ASP.NET Core applikasjoner. Figur 4.13 viser hvordan vi har koblet opp databasene i denne klassen.

```
// Configures database connection. One setup for the login database and one for the warehouse.
logindb = Configuration["loginDatabase"];
Console.WriteLine(logindb);
services.AddDbContext<LoginDatabaseContext>(opt =>
opt.UseNpgsql(logindb));

warehousedb = Configuration["warehouseDatabase"];
Console.WriteLine(warehousedb);
services.AddDbContext<WarehouseContext>(opt =>
opt.UseNpgsql(warehousedb));
```

*Figur 4.13 Konfigurering av databasene*

Konfigureringen «loginDatabase» og «warehouseDatabase» er tilkoblingsstrenger definert i User-secrets (3.7.9). Figur 4.14 viser User secrets for databasekoblingen.

```
{
  "loginDatabase": "Host=localhost;Port=5432;Username=test;Password=test;Database=logindb",
  "warehouseDatabase": "Host=localhost;Port=5433;Username=admin;Password=admin;Database=datawarehouse"
}
```

*Figur 4.14 User secrets med tilkoblingsstrenger*

### 4.3.6 API

Som forklart i 4.1 fungerer API-et som et mellomledd for selve datavarehuset og de forskjellige applikasjonene som etterspør informasjon. På grunn av at det er ulike kall fra ulike applikasjoner, har vi laget en kontroller for hver applikasjon. Ved å lage slike kontrollere er det lettere å holde oversikt over hvilke kall de forskjellige applikasjonene benytter seg av, få en god mappestruktur og gjøre det lettere for videreutvikling av API-et.

#### 4.3.6.1 Authentication Controller

Denne kontrolleren håndterer funksjoner som omhandler innlogging og registrering. Den første delen vi skal ta for oss er registrering av ny bruker. En tennant blir opprettet automatisk når de legges til i Cordel sine systemer (4.3.6.2). Når tennanten opprettes følger det ikke med brukere for tennanten. Den første brukeren vil bli opprettet manuelt av Cordel i mobilapplikasjonen. Når den første brukeren er opprettet kan brukeren legge til kollegaer manuelt via nettsiden (4.4.4.4). Derfor har vi laget to registreringskall for oppretting av

bruker. Den første for administratorbrukeren til Cordel, som benyttes i appen. Den andre som kan benyttes av brukere for å legge til kollegaer, regnskapsførere eller andre som har behov for egen bruker. Under forklarer vi de to forskjellige registreringsmetodene.

1. **Den initielle registreringen**.

   API-kallet vårt benytter seg av POST (2.5.1.1) for å sende inn e-post, passord og API-nøkkel til tennanten. Først sjekker vi om innsendt e-post allerede er tatt i bruk, så sjekker vi om API-nøkkelen er gyldig. Hvis disse to faktorene er på plass, vil API-et finne tennanten basert på API-nøkkelen, for så å opprette første brukeren. Ved oppretting av bruker vil passordet hashes (2.9.1) og saltes (2.9.2), og en userrolle (4.3.2.3) vil bli gitt. Til slutt blir brukeren knyttet opp mot tennanten og lagret i databasen.

2. **Brukerregistrering.**

   Brukerregistrering, vist i figur 4.15, har mye likt som den initielle registreringen. En av forskjellene er at det ikke medsendes en API-nøkkel for å si hvilken tennant brukeren skal knyttes opp mot. Istedenfor benytter vi oss av JWT-tokenet (2.6.2) for å finne ut hvilken tennant, brukeren tilhører. Ved å løse det slik vil det ikke være mulig å legge til brukere utenfor sin egen bedrift.

```
/*
* This register-call is for the businesses to add additional users to their business.
* To use this function, the tennant need to allready have a user connected to the tennant.
*/
[Authorize(Roles = "User")]
[HttpPost("register")]
1 reference
public IActionResult register([FromForm] string email, [FromForm] string pwd)
{
    IActionResult response;
    User userCheck = _db.findUserByMail(email.Trim().ToLower());
    Tennant tennant = _warehousedb.findTennantById(getTennantId());

    if (userCheck == null && tennant != null)
    {
        // Hashing password with a generated salt.
        var hashedPassword = BCrypt.Net.BCrypt.HashPassword(pwd);
        User newUser = new User();
        newUser.tennant = tennant;
        newUser.Email = email.Trim().ToLower();
        newUser.password = hashedPassword;
        newUser.role = Role.User;

        // Adds and saves changes to the database
        _db.setAdded(newUser);
        _db.SaveChanges();
        response = Ok("User created");
    }
    else
    {
        response = BadRequest("User already exist");
    }
    return response;
}
```

*Figur 4.15 Registrering av bruker*

For innloggingen av brukere har vi et innloggingskall som består av en e-post og passord. Figur 4.16 viser hvordan vi først finner brukeren, ved bruk av e-posten gjennom et LINQ-kall (3.7.2). Når brukeren er funnet, sjekkes det om medsendt passord er likt det dekrypterte passordet som ligger i databasen. Dersom passordene er like, vil API-et generere og returnere en JWT-token (2.6.2). Hvis det viser seg at passordene ikke er like, vil responsen 401 unauthorized (2.5.1.2) returnert.

```
[HttpPost("login")]
[Consumes("application/x-www-form-urlencoded")]
2 references
public IActionResult login([FromForm] string email, [FromForm] string pwd)
{
    User loginUser = _db.findUserByMail(email.Trim().ToLower());
    IActionResult response;
    try
    {
        //Checks if database password is equal to given password
        //Checks if email is empty
        if (loginUser.Email != null && BCrypt.Net.BCrypt.Verify(pwd, loginUser.password))
        {
            JwtTokenGenerate jwtTokenGenerate = new JwtTokenGenerate();
            var tokenStr = jwtTokenGenerate.generateJSONWebToken(loginUser, _config).ToString();
            response = Ok(tokenStr);
        }
        //Sets response to Unauthorized if the user is not registered in the database
        else
        {
            response = Unauthorized();
        }
    }
    catch (NullReferenceException)
    {
        response = Unauthorized();
    }

    return response;
}
```

*Figur 4.16 Innlogging av bruker*

### 4.3.6.2 Datasubmission Controller

Datasubmission-kontroller-klassen er knutepunket mellom Cordel sitt system og vår løsning, og består av to POST-forespørseler (2.5.1.1) som Cordel benytter seg av ved innsending av data. Den ene POST-forespørselen mottar data og legger dette inn i datavarehuset, mens den andre POST-forespørselen oppretter en tennant i datavarehuset. I Figur 4.17 ser vi hvordan en tennant blir registrert.

```
[HttpPost("registerTennant")]
[Consumes("application/x-www-form-urlencoded")]
0 references
public IActionResult registerTennant(
    [FromForm] string tennantName,
    [FromForm] string businessId,
    [FromForm] string apiKey)
{
    try
    {
        //If no apiKey is send with the request
        if (apiKey == null)
        {
            ErrorLog errorLog = new ErrorLog();

            string errorType = "API-Key empty when register tennant";
            string errorMessage = "API-nøkkelen er enten tom eller ikke presentert på riktig måte.\nAPI-key: null";

            //Creates a new errorLog to the datawarehouse
            logError(errorMessage, errorType);
        }
        else
        {
            //Creates a new tennant object with the information
            Tennant tennant = new Tennant();
            tennant.tennantName = tennantName;
            tennant.businessId = businessId;
            tennant.apiKey = apiKey;
            //Adds it to the database and saves the changes
            _db.Tennants.Add(tennant);
            _db.SaveChanges();
        }

    } //If it catches an error, it will log it in the datawarehouse
    catch (Exception e)
    {
        ErrorLog errorLog = new ErrorLog();
        string errorType = e.GetType().ToString();
        string errorMessage = e.ToString();
        //Creates a new errorLog to the datawarehouse
        logError(errorMessage, errorType);
    }

    return Ok();
}
```

*Figur 4.17 Registrering av tennant API i backend*

For at Cordel skal ha mulighet til å sende inn data til oss er det nødvendig å ha en tennant som allerede er registrert vårt system. For at en klient skal bli opprettet i datavarehuset må den ha en tennant å knytte seg opp mot og bruke tennantId som fremmednøkkel i datavarehuset. Siden denne forespørselen kun skal være tilgjengelig for Cordel har vi derfor en filtrering (3.7.4) som gjør at kun adminbrukere kan gjøre kall til denne forespørselen. Sammen med forespørselen vil det også bli sendt med tennantName, businessId og en apiKey som brukes for å registrere tennanten. Api-nøkkelen vil bli brukt for å autentisere brukeren når de senere skal sende inn data.

```
//Adds Invoice to datawarehouse
for (int i = 0; i < contentsList.Invoices.Count; i++)
{
    Invoice invoice = new Invoice();
    invoice = contentsList.Invoices[i];

    //Checks if the Invoice exists in the datawarehouse
    findInvoice(invoice, tennantId);

    //Gets the voucherFK that it will be connected to
    long voucherFK = getVoucherId(invoice.voucherId, tennantId);

    //If it finds a voucher to connect to, it will add it to the datawarehouse
    if (voucherFK != -1)
    {
        //Sets the voucherFK
        invoice.voucherFK = voucherFK;

        //Adds the Voucher to the datawarehouse
        _db.Invoices.Add(invoice);
        _db.SaveChanges();
    }
    else
    {
        //Sets the errorType and ErrorMessage
        string errorType = "Invoice - Voucher Connection";
        string errorMessage = "There is no Voucher with Id: "
            + invoice.voucherId
            + " that invoiceId: "
            + invoice.invoiceId +
            " can be connected to. TennantId: " + tennantId;

        //Creates a new errorLog to the datawarehouse
        logError(errorMessage, errorType);

        //Throws a new exception so the program dont handle more data
        throw new InvalidModelFK();
    }

}
```

*Figur 4.18 Eksempel på hvordan data blir lagt inn i datavarehuset*

Data som blir sendt inn til vårt system vil komme som en JSON (2.6.1). For å kunne forsikre seg om at personen som prøver å legge til data er autorisert til å gjøre dette, bruker vi en medsendt API-nøkkel. Denne API-nøkkelen vil bli sendt både i URL-en og i JSON kroppen. For å verifisere, sjekker vi først API-nøkkelen i URL-en mot de lagrede tennantene i datavarehuset. Blir API-nøkkelen verifisert vil den også bli sammenlignet med API-nøkkelen som blir sendt med i JSON kroppen. Informasjonen som ankommer under en forespørsel, vil være bygget opp som flere lister med informasjon. Hver av disse listene er modeller i backenden vår.

Når data blir behandlet, vil programmet først sjekke om dataen allerede ligger inne i datavarehuset. Hvis dette er tilfellet, vil dette bli slettet for at vi senere kan legge det inn med oppdatert informasjon. Videre vil fremmednøkkelene bestemmes ved bruk av en ID som blir sendt med objektet (Figur 4.19). Finner funksjonen en fremmednøkkel som kan brukes vil programmet gå videre, oppdatere fremmednøkkelen og så lagre det nye objektet i datavarehuset. Hvis funksjonen ikke finner en mulig fremmednøkkel, vil funksjonen returnere -1, en feilmelding vil bli logget i datavarehuset og en exception vil bli kastet for å forhindre at programmet behandler mere data.

```
/*
The function getClientId checks for an existing client in the datawarehouse,
and then returns the unique id back if it exists, or returns -1 if it does not exist
*/
2 references
private long getClientId(long cordelId, long tennantId)
{
    Client databaseClient = _db.getClientById(cordelId, tennantId);

    if (databaseClient != null)
    {
        return databaseClient.id;
    }
    return -1;
}
```

*Figur 4.19 Eksempel på uthenting av Unik id*

Det har vært viktig for oss å kunne oppdage feil ved innsending av data. Dette er også noe Cordel ønsket for å kunne enkelt diagnostisere problemer som kan oppstå. For å oppnå høy kohesjon (2.4.1) valgte vi å lage en funksjon som enkelt kan bli brukt fra flere steder i koden. Om det skulle oppstå en feil med innsending av data vil den først bli fanget (figur 4.20). Her vil programmet finne hvilke type feilmelding, teknisk beskrivelse på hva som er feil, og tidspunkt når feilen oppsto. Deretter vil denne informasjonen bli sendt videre til «logError()» funksjonen, og lagret i datavarehuset. Figur 4.21 viser hvordan dette gjøres.

```
}
/*
The programm picks up this error if the JsonConvert.DeserializeObject fails.
Fields that are null or not the expected datatype can cause this
*/
catch (JsonSerializationException e)
{

    ErrorLog errorLog = new ErrorLog();
    string errorType = e.GetType().ToString();
    string errorMessage = e.Message.ToString() + " businessId: " + apiKey;

    //Creates a new errorLog to the datawarehouse
    logError(errorMessage, errorType);
```

*Figur 4.20 Eksempel på hvordan vi plukker opp feilmeldinger*

```
/*
This function creats a new errorLog, and uses the information in the
parameters to create a new errorLog in the datawarehouse, and saves the changes
*/
16 references
private void logError(string errorMessage, string errorType)
{
    ErrorLog errorLog = new ErrorLog();

    DateTime timeOfError = DateTime.Now;

    errorLog.errorType = errorType;
    errorLog.errorMessage = errorMessage;
    errorLog.timeOfError = timeOfError;

    _db.ErrorLogs.Add(errorLog);
    _db.SaveChanges();


}
```

*Figur 4.21 Lagring av feilmelding i datavarehus*

#### 4.3.6.3 Web Controller

Web kontrolleren består av GET-forespørsler (2.5.1.1) som henter ut informasjon til BI-systemet (2.3). Informasjonen som skal vises i nettsiden er ferdigbehandlet og filtrert på dato der det lar seg gjøre. Kontrolleren opererer stort sett på samme måte for alle kallene. Vi vil derfor forklare noen fellesfaktorer for så å gå inn i detalj på kallet for fravær og kundefordringer.

For å unngå lav kohesjon (2.4.1) og gjøre det lettere for videreutvikling har vi lagd en funksjon for å hente ut korrekt dato basert på hvor langt tilbake brukeren vil begrense informasjonen. Figur 4.22 viser koden for filtrering av dataen.

```
/*
* Sets the filter to a pre-defined option based on what's requested, if no option is specified, all will be selected.
* When modifying days, years or months, the timer for that day will allways be set to 00:00:00, so there is no need to
* remove the hours aswell.
*/
4 references
private DateTime compareDates(string time)
{
    DateTime dateTimeNow = DateTime.Now;
    DateTime comparisonDate = dateTimeNow;
    int tempMonth = dateTimeNow.Month;
    int tempWeek = (int)dateTimeNow.DayOfWeek;
    int tempDay = dateTimeNow.Day;

    switch (time)
    {
        case "lastSevenDays":
            comparisonDate = dateTimeNow.Date.AddDays(-7);
            break;
        case "lastThirtyDays":
            comparisonDate = dateTimeNow.Date.AddDays(-30);
            break;
        case "lastTwelveMonths":
            comparisonDate = dateTimeNow.Date.AddYears(-1);
            break;
        case "thisWeek":
            comparisonDate = dateTimeNow.Date.AddDays(-tempWeek + 1);
            break;
        case "thisMonth":
            comparisonDate = dateTimeNow.Date.AddDays(-tempDay + 1);
            break;
        case "thisYear":
            comparisonDate = dateTimeNow.Date.AddMonths(-tempMonth + 1).AddDays(-tempDay + 1);
            break;
        default:
            Console.WriteLine("No filter added, listing all..");
            comparisonDate = dateTimeNow.Date.AddYears(-30);
            break;
    }
    return comparisonDate;
}
```

*Figur 4.22 Filtrering i WebController*

Ved starten av funksjonen hentes datoen og tidspunktet funksjonen blir kalt, deretter vil den velge riktig case basert på hvilket filter som blir valgt. De forskjellige mulighetene har egne utregninger for å finne korrekt dato basert på datoen og tidspunktet hentet i starten av funksjonen.

Kontrolleren benytter seg også av view models (4.3.2) for å gjøre data fra datavarehuset om til objekter nettsiden kan benytte seg av direkte. Ved at vi gjør det på denne måten minsker vi trafikken mellom backend og frontend, slik at trafikken over nettet forholdes til et minimum. Dette gjør at brukeropplevelsen føles bedre, og handlingers ventetid minskes.

**Fravær**:

Hver tennant har en liste med ansatte, og hver ansatt har en liste med fravær. Hvis dette skulle vises direkte i en graf ville det virke veldig uoversiktlig, og vanskelig å tolke. Løsningen som ble etterspurt var å summere hvor mange timer per måned, men også for hver dag hvis tidsbegrensingen var veldig kort, som for eksempel når filteret er satt for denne uken.
Det ble utviklet to metoder for å håndtere kalkuleringene av fravær. Den første metoden bestod av flere for-løkker inne i hverandre, noe som kan gi høy kjøretid (2.10). Den andre metoden, som vi valgte å beholde, er å sortere listen fra LINQ-kallet på dato slik at vi kun trenger å sammenligne nåværende fravær med neste fravær i listen.

```
235     [Authorize]
236     [HttpGet("absence")]
        1 reference
237     public IList<AbsenceView> getAbsenceRegister(string filter)
238     {
239         long tennantId = getTennantId();
240         DateTime comparisonDate = compareDates(filter);
241         //Gets a list of absences for the given tennant within the specified timelimit, ordered by date(ascending).
242         var absence = _warehouseDb.getAllAbsenceFromDate(tennantId, comparisonDate);
243
244         List<AbsenceView> absenceViews = new List<AbsenceView>();
245         double totalAbsence = 0;
246         if (filter == "thisWeek" || filter == "thisMonth" || filter == "lastThirtyDays" || filter == "lastSevenDays")
247         {
248             try
249             {
250                 for (int i = 0; i < absence.Count; i++)
251                 {
252                     if (i != absence.Count - 1)
253                     {
254                         //since the list is ordered allready, we can compare current month with next, if it is, add the duration to months total
255                         if (absence[i].fromDate.DayOfYear == absence[i + 1].fromDate.DayOfYear)
256                         {
257                             totalAbsence += absence[i].duration;
258                         }
259                         // Next absence is a new day, add the current absence we're on and add the view to the new list of views.
260                         else
261                         {
262                             totalAbsence += absence[i].duration;
263                             AbsenceView view = new AbsenceView();
264                             view.year = absence[i].fromDate.Year;
265                             view.month = absence[i].fromDate.Month;
266                             view.day = absence[i].fromDate.Day;
267                             view.weekDay = absence[i].fromDate.DayOfWeek.ToString();
268                             view.totalDuration = totalAbsence;
269                             absenceViews.Add(view);
270                             totalAbsence = 0;
271                         }
272                     }
273                     //last absence has the same day as the one previously added absence
274                     else if (absence[i].fromDate.DayOfYear == absence[i - 1].fromDate.DayOfYear)···
287                     //last absence is in a new day
288                     else···
301                 }
302             }
303             catch (Exception e)
304             {
305                 Console.WriteLine("Error: " + e);
306             }
307             return absenceViews;
308         }
309         else  //Monthly-based filter is chosen, now we summarize for each month instead of day.
310         {
311             try
```

*Figur 4.23 viser essensen av koden for å oppsummere fravær, deler av koden er lukket og logikken for månedlig oppsummering er ikke med i bilde.*

Figuren 4.23 viser logikken bak kallet for å hente fravær. Den henter først hvilken tennant brukeren tilhører ved hjelp av JWT-token (2.6.2), så henter den en dato for å filtrere på. Med disse variablene finner den de riktige fraværene og sorterer de etter dato. Før den går inn i oppsummering av fravær oppretter den en liste med view modellen (4.3.2) AbsenceView (figur 4.24).

```
21 references
public class AbsenceView {
10 references
public int year {get; set;}
10 references
public int month {get; set;}
 5 references
 public int day  {get; set;}
7 references
public string weekDay {get; set;}
10 references
public double totalDuration {get; set;}
}
```

*Figur 4.24 klassen AbsenceView*

For oppsummeringen av fraværstiden vil den velge mellom daglig eller månedlig oppsummering basert på hvilket filter som er valgt. Begge oppsummeringene fungerer på samme måte, og benytter seg av samme view model. Eneste forskjellen er kriteriene for når den skal opprette et nytt objekt. Det daglige sammenligningskriteriet ser på hvilken dag i året det er. Så lenge neste i rekken er samme dag vil den fortsette å gå gjennom listen og legge til fraværstiden, men hvis neste i rekken er ulik vil den ta fraværstiden den har oppsummert, legge til datoen for så å legge view modellen i listen. Slik går den helt til den har iterert seg gjennom hele listen, med visse unntak for å unngå feilmeldinger. Sammenligningskriteriet for månedlig oppsummering bruker hvilken måned det er istedenfor hvilken dag. Ved å benytte oss av samme viewmodel og bare endre kriteriene for oppsummeringen basert på hvilket filter som blir valgt, unngår vi mye repetitiv kode og unødvendige endepunkt i API-et.

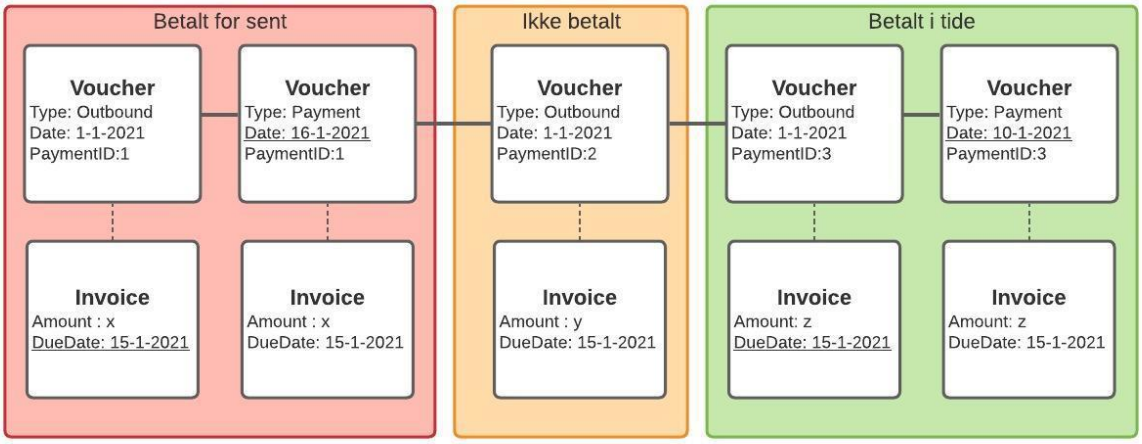
**Kundefordringer**:
Målet med kundefordringer er å få en oversikt over alle utgående fakturaer som ikke ble betalt i tide eller fortsatt er ubetalt. Siden bilagene har fire mulige typer, utgående/inngående og

betaling/utbetaling, må vi hente ut de med riktig type som i dette tilfellet er utgående og betaling.

```
1 reference
public List<Voucher> getVouchersInDescendingByPaymentThenByType(long tennantId, DateTime comparisonDate)
{
    var vouchers = Vouchers
    .Where(v => v.client.tennantFK == tennantId && v.date >= comparisonDate)
    .Where(d => d.type == "outbound" || d.type == "payment")
    .Include(c => c.invoice)
    .OrderByDescending(p => p.paymentId).ThenBy(d => d.type)
    .ToList();
    return vouchers;
}
```

*Figur 4.25 Spørring for bilag*

Figur 4.25 viser spørringen som henter ut alle bilagene som er av type utgående, og betalinger for en gitt tennant innen en bestemt tidsramme. Bilagene blir sortert på paymentID og type, hvor paymentID er en betalings ID som knytter en utgående faktura til en betaling på fakturaen. Dermed vil hver utgående faktura som har en betaling ha denne betalingen som neste bilag i listen, illustrert i modellen under.




*Figur 4.26 Illustrasjon av hvordan listen vil se ut etter spørringen.*

Resten av algoritmen består av to steg:

1. En for-løkke som fungerer på samme måte som i fraværs-metoden, sjekker det nåværende bilaget mot neste bilag i rekken. Gjennom en rekke kriterier i form av if-setninger henter vi ut alle bilag som er betalt for sent eller ikke betalt, også vist som rød og oransje boks i figuren 4.26. For hvert tilfelle som er rød eller oransje så regner vi ut hvor forsinket betalingen var/er. Når forløkken er ferdig sitter vi igjen med en liste over alle bilag som er betalt for sent eller ikke betalt. Disse bilagene er lagret som

nye objekter og holder på informasjon om når betalingen skulle blitt betalt, hvor forsinket betalingen var/er og hvor mye penger betalingen innehar.

2. En while-løkke som starter på datoen for filteret som ble valgt. Som for eksempel hvis filteret var «siste 30 dagene» vil den starte på datoen som var for 30 dager siden. Videre vil den bevege seg med et tidsintervall på 7 dager, helt til den når dagens dato. For hver syvende dag lages det en ny view model som går gjennom listen fra steg 1 og henter ut de som har en betalingsfrist før- og betalingsdato etter nåværende dato. Her lagrer den hvor mye penger som er skyldig basert på hvor forsinket betalingen er. De forskjellige kategoriene på hvor forsinket de er, består av: 1-30 dager, 31-60 dager, 61-90 dager eller 90+ dager forsinket. Til slutt har vi en sortert liste fra filtreringsdato til forespørrende dato som oppdateres hver syvende dag, og til enhver tid har oversikt over hvor mye som skyldes i hver kategori.

De resterende kallene er noe mindre komplisert og løses ofte ved en database spørring etterfulgt av en enkelt forløkke. Siden disse er mindre kompliserte, vil vi ikke gå inn på hvordan disse er bygget, men si noe om hva de tilbyr:

- Alle avventede ordre.
- All tidligere ordre.
- Alle klientene bedriften har.
- Statistikk over hvor mange kunder og leverandører bedriften har
- Statistikk over bedriftens kjønnsfordeling av ansatte.
- Statistikk over bedriftens saldo og budsjett.

#### 4.3.6.4 App Controller

App kontrolleren består av POST-forespørsler (2.5.1.1) og noen enkle metoder for å assistere kallene. Denne kontrolleren henter ut informasjonen som er nødvendig for mobilapplikasjonen, med unntak av registrering og innlogging (4.3.4.1). Figur 4.27 viser et eksempel på et endepunkt i app controlleren.

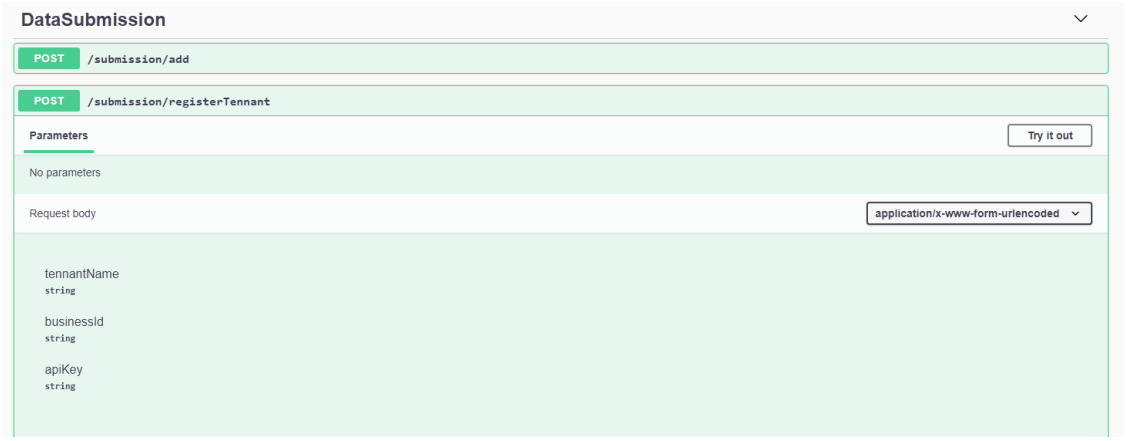
```
/*
*   Returns number of tennants and errors last 24 hours as json.
*/
[Authorize(Roles = "Admin")]
[HttpPost("homeinfo")]
1 reference
public IActionResult getNumberOfTennantsAndErrorsAsJson()
{
    IActionResult response;

    AppInfoHomeMenu appInfoHomeMenu = new AppInfoHomeMenu();

    appInfoHomeMenu.numberOfTennants = getNumberOfTennants();
    appInfoHomeMenu.numberOfErrors = getNumberOfErrorsLastTwentyFour();

    var dataToJson = JsonConvert.SerializeObject(appInfoHomeMenu);

    response = Ok(dataToJson);

    return response;
}
```

*Figur 4.27 Henter ut antall bedrifter og feilmeldinger*

Appkontrolleren er kun tilgjengelige for Cordels ansatte og har derfor en filtrering (3.7.4) som gjør at kun «Admin» brukere kan gjøre kall til disse metodene. Metoden i figur 4.27 benytter seg av en viewmodel (4.3.2) for å kunne sende nødvendig informasjon til applikasjonen. For å gjøre det enklere for mobilapplikasjonen konverteres objektet om til et JSON objekt ved å benytte oss av Newtonsoft (3.7.5).

### 4.3.6.5 API Dokumentasjon

For at Cordel skal vite hvordan de skal sende data til løsningen vår, har de behov for å vite hvordan forespørselen skal være bygd opp. Dette har vi løst ved å bruke Swagger (3.7.10) for å autogenere API dokumentasjon som Cordel kan bruke.




*Figur 4.28 Visualisering av API i swagger*

Når vi brukte Swagger ble det autogenerert to forskjellige visualiseringer av API-ene våre. Først ble det opprettet en HTML-nettside som viste alle API-kallene på en enkel og ryddig måte, med URL-en og hvilke typer parametere som er forventet (Figur 4.28). Dette er bra for å få en enkel oversikt, men ikke nok som en fullstendig dokumentasjon som Cordel kan bruke. For å få en mer detaljert beskrivelse på hvordan API-et er bygget opp har man også tilgang til et JSON dokument som går mye mer inn i detaljene på hva som er forventet (Figur 4.29). Dette JSON dokumentet har vi videresendt til Cordel for at de skal kunne starte på utviklingen av deres løsning.

```
"post": {
  "tags": [
    "DataSubmission"
  ],
  "requestBody": {
    "content": {
      "application/x-www-form-urlenc
        "schema": {
          "type": "object",
          "properties": {
            "tennantName": {
              "type": "string",
              "nullable": true
            },
            "businessId": {
              "type": "string",
              "nullable": true
            },
            "apiKey": {
              "type": "string",
              "nullable": true
            }
          }
        },
        "encoding": {
          "tennantName": {
            "style": "form"
          },
          "businessId": {
            "style": "form"
          },
          "apiKey": {
            "style": "form"
          }
        }
      }
    }
  }
```


*Figur 4.29 JSON API dokumentasjon*

### 4.3.7 Sikkerhet

For API-et har vi tatt noen forutsetninger når det kommer til sikkerhet. Spesielt SQL-injeksjoner (2.9.3) er en vanlig måte for angripere å gjøre skade på databasen vår. Måten vi har unngått SQL-injeksjoner er ved å unngå bruk av IQueryable<T> (95). Grunnen til at man skal unngå returnering av IQueryable<T> er fordi angriperen kan kjøre spørringer på resultatet slik at det eksponerer ekstra data eller øker mengden data som blir sendt tilbake (96). Angriperen kan også fange unntak som blir kastet, som igjen kan inneholde informasjon som ikke skal eksponeres (96). Istedenfor å bruke IQueryable har vi valgt å returnere lister av objekter, hvor vi selv har kjørt spørringene og omgjøre disse til nye modeller som kun inneholder nødvendig informasjon til websider og applikasjoner. Vi har også benyttet oss av LINQ (3.7.2) istedenfor vanlige SQL spørringer i Entity Framework (3.7.1) da det har innebygd sikkerhet ved å overføre data til databasen som SQL parametere.

Andre former for sikkerhet inngår i kapittel 4.3.4 hvor all form for bruk av applikasjonene innebærer JWT-token. Det vil også være umulig å opprette en bruker hos en bedrift med mindre Cordel manuelt har opprettet bedriftens første bruker, etter det er det bedriften selv som velger hvem som skal få tilgang til deres data. Siden alle datakall fra webcontrolleren (4.3.6.3) benytter seg av JWT-token for å finne ut hvilken bedrift brukeren tilhører, vil det være vanskelig å hente ut informasjon om andre tennants.

## 4.4 Frontend

Som forklart i 4.1 har vi laget to applikasjoner som visualiserer informasjon lagret i datavarehuset. Nettløsningen er laget for oppdragsgivers kunder og er en nettløsning hvor kundene kan hente informasjon om sin bedrift. Mobilapplikasjonen er laget for oppdragsgiver og er laget for overvåkning av backend og brukerne av nettløsningen. I denne delen går vi i detalj på nettsiden og applikasjonen, demonstrert med kodesnutter og bilder av brukergrensesnittene.

### 4.4.1 Språk

Cordel spesifiserte at de ønsket at språket skulle være norsk da kundene deres er norske. Derfor er både mobilapplikasjonen (4.4.4) og nettsiden (4.4.3) utviklet med norsk som språk, og gir derfor også norske tilbakemeldinger.

### 4.4.2 Designmønster

Både nettsiden og mobilapplikasjonen har et enkelt design med hovedfokus på brukervennlighet. Fargekodene som er brukt er fargekoder gitt av Cordel.

I utviklingsprosessen av nettsiden fokuserte vi på et design tilpasset datamaskiner. Likevel skal ikke dette gå på bekostning av brukere på mobile enheter. Nettsiden er responsiv i den grad at den har tre forskjellige utseender avhengig av størrelsen på enheten som blir brukt. En for datamaskin, en for nettbrett og en for mobil. Mobilapplikasjonen er laget for å kunne brukes på mobiler og nettbrett, men egner seg best til mobiler.

Både mobilappen og nettsiden har vi designet med Don Norman (2.1.1) sine designprinsipper i bakhodet. Unngå muligheten for at brukeren gjør feil, men samtidig ha et design som ser fint ut.

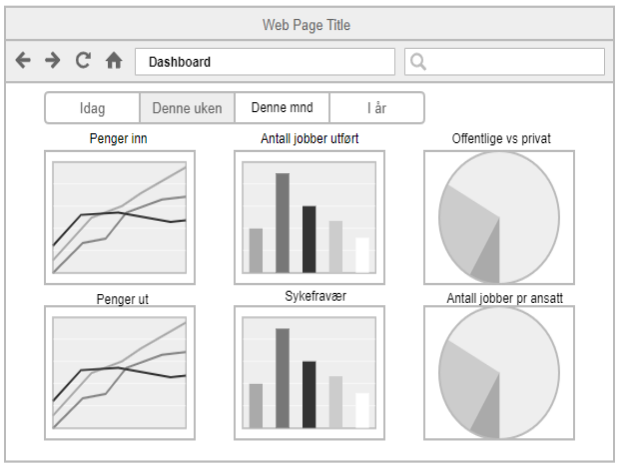
### 4.4.3 Innlogging og brukerhåndtering

Autentisering og innlogging er en viktig del av både nettsiden og mobilapplikasjonen. Det er flere likheter i hvordan det blir håndtert da begge har inntastingsfelt for epost og passord. Når innloggingsknappen blir trykt, blir informasjonen i disse feltene sendt til backenden, hvor den blir prosessert (4.3.6.1). Ved vellykket autentisering blir en Json Web token (2.6.2) sendt som et resultat fra backenden. Denne blir håndtert på litt forskjellige måter i nettside og mobilapplikasjon, og vil bli spesifisert nærmere senere i kapitelet.

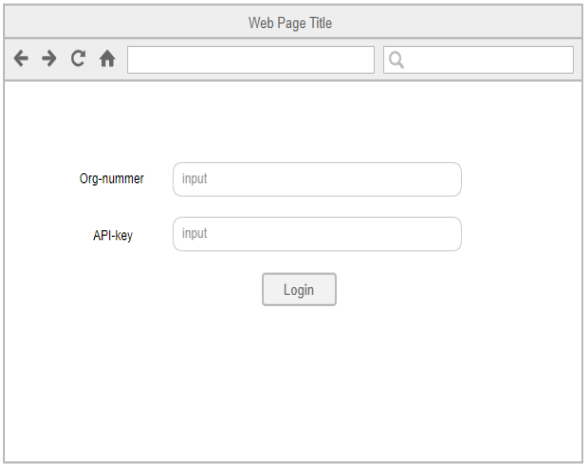
### 4.4.3 Nettside med bi-system

Bi-systemet (2.3) er det eneste Cordel sine kunder vil se. Det har derfor vært viktig for oss at presentasjonen er forståelig og at kunden føler at de får utbytte av nettsiden.

Nettsiden har fire forskjellige sider, en innloggingsside, et dashbord, en side for å legge til nye brukere og en utloggingsside. Innloggingssiden er veldig enkel og har to inntastingsfelt for brukernavn og passord. Dashbordet fungerer som hovedsiden til nettsiden. Her blir alt av grafer og økonomisk informasjon presentert til brukeren. Siden hvor du kan legge til nye brukere inneholder tre felt, et for e-post og to for passord. Utloggingssiden er ikke i seg selv en fysisk side som du kan se, men den omdirigerer deg til innloggingssiden når den blir kalt. I tillegg til dette har vi en header. Denne er til stede på alle sidene og blir brukt for navigering frem og tilbake mellom sidene.

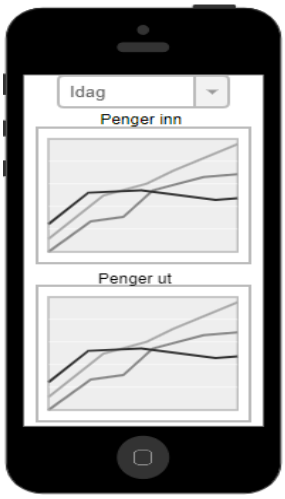



*Figur 4.30 Wireframe dashbord for datamaskinvisning*




*Figur 4.31 Wireframe Login for datamaskinvisning*


*Figur 4.32 Wireframe dashbord for Nettbrettvisning*

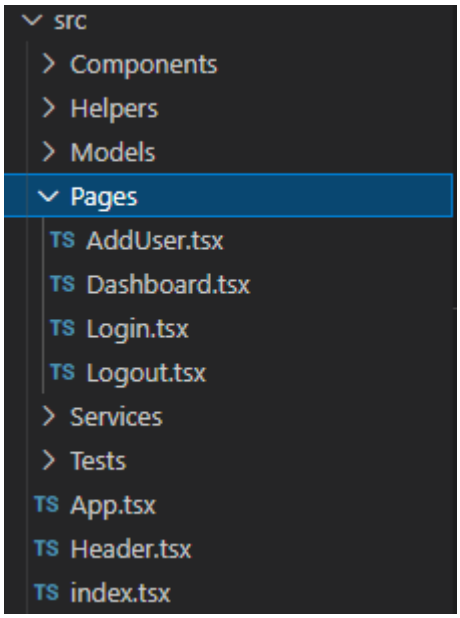



*Figur 4.33 Wireframe dashbord for telefon*

I startfasen av prosjektet lagde vi wireframes for å hjelpe oss i utviklingen av nettsiden. Her har vi tre forskjellige versjoner. En for datamaskin, en for nettbrett og en for telefon. Som vi skal se senere i rapporten, har sluttproduktet endret seg litt fra planlagt utseende, men hovedtrekkene er likevel de samme.

#### 4.4.3.1 Mappestruktur

Figur 4.34 viser vi mappestrukturen til nettsiden. I mappen «Components» ligger alle filene for fremstilling av grafer. Helpers-mappen inneholder filer for «Grid», «PrivateRoute» og «HandleResponse». Dette er filer som hjelper med bygging av layouten og håndtering av responser. Models inneholder kun en fil, «User». Her er det felt som beskriver en bruker. I pages lagrer vi alle nettsidene. Services er en mappe for filer som snakker med API-et. Eksempler på dette er innlogging eller å hente navnet til brukeren som er logget inn. Tests er en mappe for nettsidens tester. «App», «Header» og «index» ligger på utsiden av mappene.




*Figur 4.34 Mappestruktur nettside*

### 4.4.3.2Routing

Sidenavigasjonen foregår i App.tsx hvor React Router (3.7.17) blir brukt for å håndtere sidenavigasjonen i applikasojnen. Figur 4.35 viser hvordan App.tsx er bygget opp. Komponentene Login, Dashbord, AddUser og Logout blir importert og kalt ved sin tilsvarende path. Vi bruker komponenten PrivateRoute for rutene hvor innlogging er nødvendig. Prøver brukeren å få tilgang til disse sidene uten å logge inn, vil den bli sendt tilbake til loginsiden.

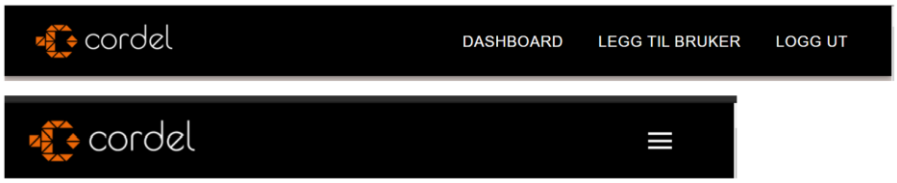
```
return (
  <Router>
    {<Header />}
    <Switch>
      <Route path="/" exact component={Login} />
      <PrivateRoute path="/dashboard" component={Dashboard} />
      <PrivateRoute path="/AddUser" exact component={AddUser} />
      <PrivateRoute path="/logout" exact component={Logout} />
    </Switch>
  </Router>
```

*Figur 4.35 App.tsx*

### 4.4.3.3 Header

Headeren for nettsiden består av en trykkbar logo og forskjellige navigasjonsknapper for de ulike rutene i 4.4.3.2. Trykker man på logoen blir man sendt til dashbord- eller innloggingssiden basert på om brukeren er logget inn eller ikke. Headeren er responsiv, men vil også endre utseende hvis størrelsen på nettleseren når en bestemt størrelse. I versjonen for

større skjermer vil navigasjonsmulighetene vises som vanlige knapper, men for mindre skjermer vil navigasjonsmulighetene gjemmes inne i en meny. Dette demonstreres i figur 4.36.



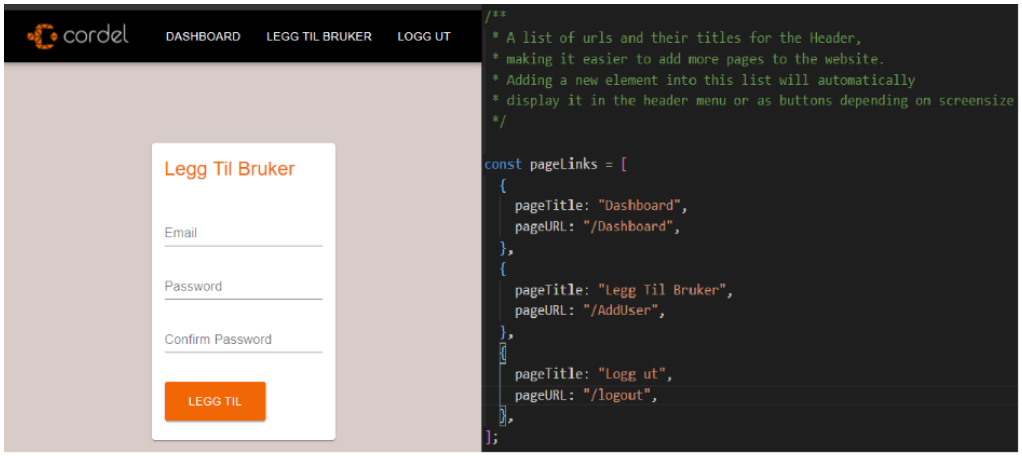

*Figur 4.36 Responsiv header*

Måten dette gjøres på er ved bruk av MaterialUI sitt breakpoint for temaer. I kodesnutten vist i figur 4.37 har vi benyttet oss av størrelse «xs» som er for skjermer mellom 0 til 600px (97).

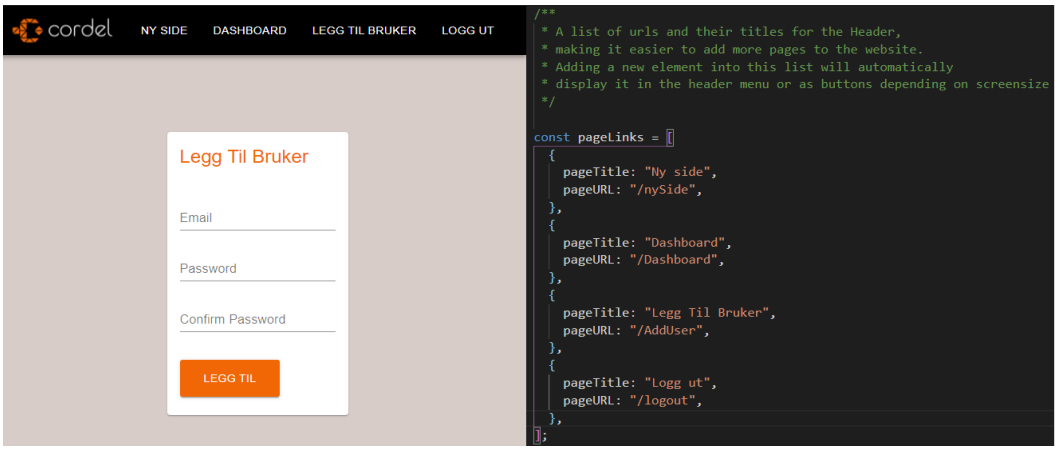
```
/* Swaps theme based on the breakpoint below*/
const theme = useTheme();
const isMobile = useMediaQuery(theme.breakpoints.down("xs"));
```

*Figur 4.37 Bruk av breakpoints*

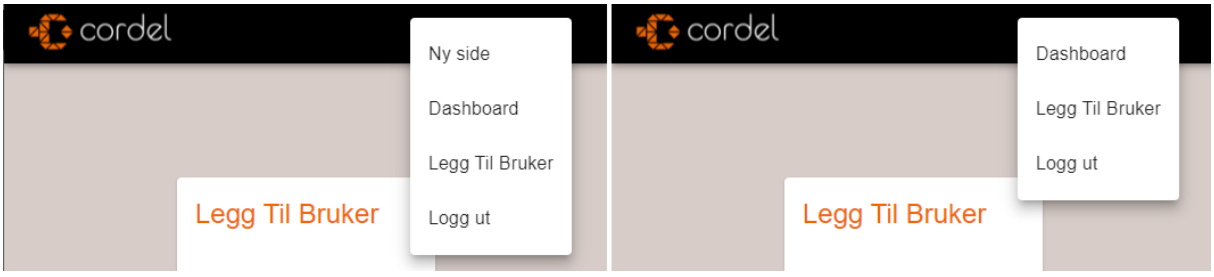
Da vi utviklet denne løsningen fokuserte vi på å gjøre det så enkelt som mulig for videreutviklingen av denne funksjonaliteten. Til vanlig må både headeren for større skjermer og mindre skjermer oppdateres hvis en ny side skal legges til.  Måten vi har løst dette på er ved å lagre titlene og URL-ene sammen som objekter. Da kan headerne fra figur 4.36 gå gjennom objektene for så å legge inn hva de skal hete og hvilken side de skal henvise til. Nedenfor, i figurene 4.38 til 4.39, er en kort demonstrasjon av hvor enkelt det er å gjøre endringer på headeren.




*Figur 4.38 Headerdesign og -kode*




*Figur 4.39 Headerdesign og -kode med en ekstra side*

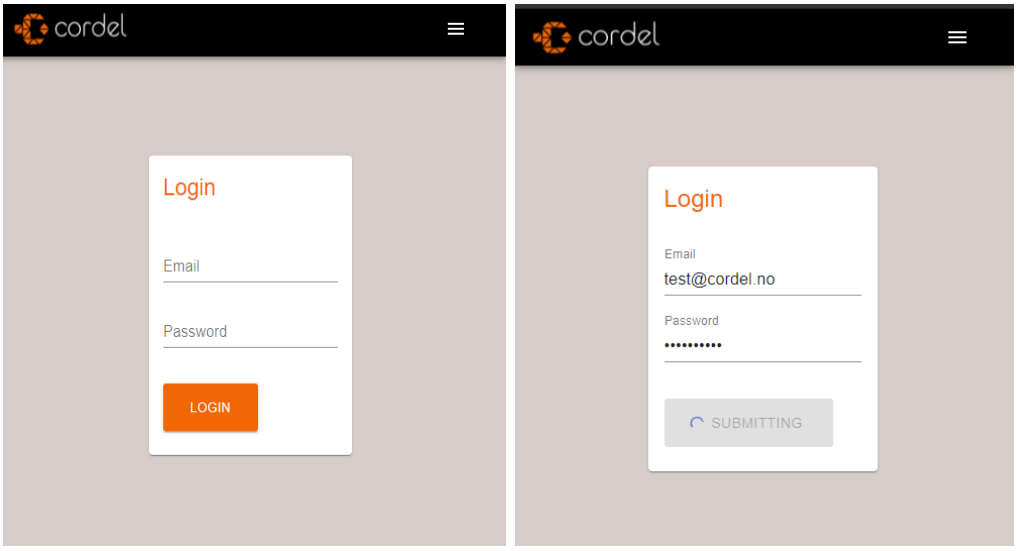



*Figur 4.40 Endring i header*

For å legge til, fjerne eller endre sidene som vises i headeren trengs det kun endringer i listen med titler og URL-er. Alt annet som medgår en slik endring vil skje automatisk. Håndtering av trykk på knapper, innlasting av ny side, posisjon i headeren for både større og mindre skjermer og til slutt utseende for både menyen og knappene.

#### 4.4.3.4 Innloggingsside

Innloggingssiden er den første siden du møter, og kundene til Cordel vil kunne logge seg inn her. Her kreves det en epost med rett utforming og et passord for å kunne gå videre til autentiseringen. Figur 4.41 viser innloggingsiden, med og uten utfylte felt.




*Figur 4.41 Innloggingsside*

Som nevnt i kapitel 4.4.3 blir det sendt en Json Web token ved autentisering av en bruker fra backenden. Denne tokenen blir lagret i LocalStorage (3.7.23) og holder på tokenen slik at du kan holde deg innlogget selv om du oppdaterer siden.

```
/* Makes an Api call and send the email and password  */
function login(email: string, pwd: string) {
    const requestOptions = {
        method: 'POST',
        headers: { 'Content-Type': 'application/x-www-form-urlencoded; charset=UTF-8' },
        body: new URLSearchParams({ email, pwd })
    };

    return fetch(`auth/login`, requestOptions)
        .then(handleResponse)
        .then(token => {
            // store user details and jwt token in local storage to keep user logged in between page refreshes
            localStorage.setItem('currentUser', (token));
            currentUserSubject.next(token);
            return token;

        });
}
```

*Figur 4.42 Login-Api-kall*

Figur 4.42 viser hvordan innloggingen blir gjort i nettsiden. Her bruker vi en post forespørsel (2.5.1.1) til authentication kontrolleren sitt innloggingskall (4.3.6.1). Email og passord (pwd) blir sendt som en body til backenden for verifisering.

Når backenden sender en respons tilbake, blir den håndtert i funksjonen handleResponse (Figur 4.43). Her blir de forskjellige responskodene (2.5.1.2) behandlet og gir tilbakemelding til brukeren dersom noe har gått galt. Dette er en universal metode, og blir også brukt i andre sammenhenger.

```
/* This class handles the response and console logs 400, 401 and 403 response codes. */
export function handleResponse(response: any) {
    return response.text().then((text: string) => {
        const data = text;
        if (!response.ok) {
            if ([401, 403].indexOf(response.status) !== -1) {
                console.log("Unauthorized");
                // auto logout if 401 Unauthorized or 403 Forbidden response returned from api
                AuthenticationService.logout();
            }
            if ([400].indexOf(response.status) !== -1) {
                console.log("Bruker finnes allerede")
                alert("Bruker finnes allerede")
            }

            const error = (data) || response.statusText;
            return Promise.reject(error);
        }

        return data;
    });
```

*Figur 4.43 handleResponse-metoden*

Om responsen fra backenden ikke blir fanget i handleResponse-metoden (figur 4.43), blir den sendt tilbake til login-funksjonen (figur 4.43). Dermed blir tokenen lagret i local storage (3.7.23).

### 4.4.3.5 Dashbord

Dashbordet fungerer som nettsidens hjemmeside. Her vil alt av informasjon om den innloggede bedriften presenteres grafisk. Dataen blir presentert på en ryddig måte med Material UI (3.7.11) sin Card komponent. I figur 4.44 ser vi at Card komponentens hovedattraksjon er grafen, linjediagrammet. Øverst til venstre i card komponentene er det en nedtrekksmeny som tillater brukeren å filtrere dataen på forskjellige tidsperioder; Dette året, siste 12 måneder, siste 30 dager etc. Øverst til høyre blir navnet på grafen presentert.


*Figur 4.44 Dashbord*

I utviklingen av nettsiden har vi fokusert på nettsidens mulighet for utvidelse. Derfor er komponentene i dashbordet enkle å endre og utvide. Et eksempel på dette er Grid komponenten som tillater enkel organisering av objekter i dashbord. Her kan man enkelt endre antall objekter per kolonne, plassering i kolonnen og plassering i rader. Figur 4.45 viser de forskjellige tilpasningsmulighetene. Figur 4.46 viser bruken av Grid komponenten i dashbordet.

```
/* How many colums the item is taking of the grid.
1 mmaking there be many small items in the column,
and 12 meaning only one item in the whole column. */
type GridSizes = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12;

interface GridProps {
  alignItems?: GridItemsAligns;
  column?: boolean;
  expanded?: boolean;
  justify?: GridJustify;
  lg?: GridSizes;
  md?: GridSizes;
  row?: boolean;
  sm?: GridSizes;
}
```

*Figur 4.45 Kodeutdrag fra Grid.tsx*

```
{/* Grid justification */}
<Grid
  column={true}
  sm={12}
  md={12}
  justify={"flex-end"}
  alignItems={"flex-end"}
>
```

*Figur 4.46 Bruk av Grid i Dashbord*

Figur 4.47 viser hvordan dataen i grafene blir hentet fra backenden med bruk av Axios (3.7.14). I dette eksempelet blir det hentet data fra endepunktet «absence» (fravær) i

webcontrolleren (4.3.6.3) ved bruk av Axios (3.7.14). Denne grafen skal vise data for de siste tolv månedene, derfor er filteret «lastTwelveMonths» brukt.

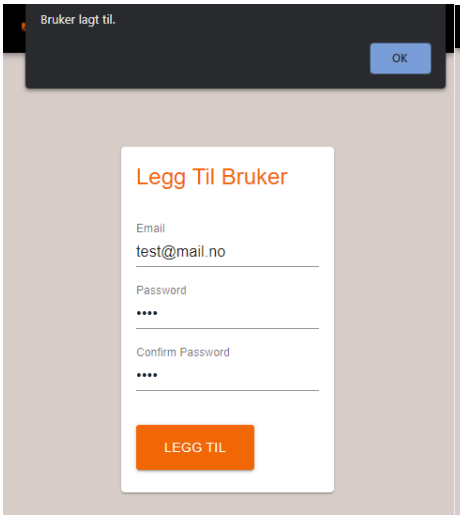
Json objektet (2.6.1) som blir sendt fra backenden må justeres for å bli brukt i nettsiden. Etter justeringen returneres Json objektet og «chartData» blir satt som «expectedData», som vist i figur 4.43.

```
const chart = () => {
  //Api call to the backend getting the information about Absence the last twelve months.
  //Authorizes using token stored in local storage.
  axios
    .get("web/absence", {
      params: { filter: "lastTwelveMonths" },
      headers: {
        Authorization:
          "bearer " + AuthenticationService.getCurrentUser("currentUser"),
      },
    })
    .then((res) => {
      //Alters the Json data to fit the chart in a specific way.
      var actualData = res.data;
      var ExpectedData = actualData.map((obj: any) => {
        // Get month number from date-string and then substract 1
        var monthNum = parseInt(obj.month) - 1;
        // Get month name from the array and adds years.
        obj.month = monthsName[monthNum] + " " + obj.year;
        // Return the object
        return obj;
      });
      setChartData(ExpectedData);
    })
    .catch((err) => {
      console.log(err);
    });
};
```

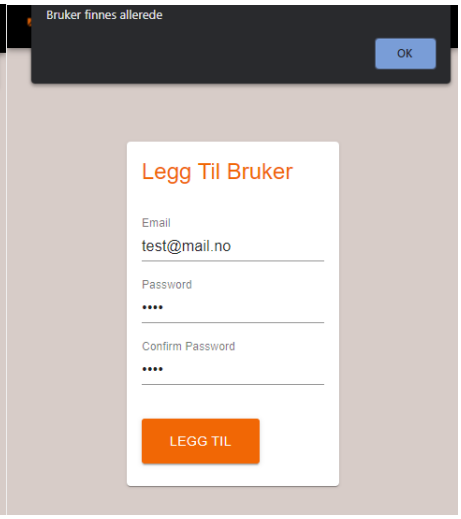
*Figur 4.47 Kodeutsnitt fra absence grafdata*

### 4.4.3.6 Legg til bruker

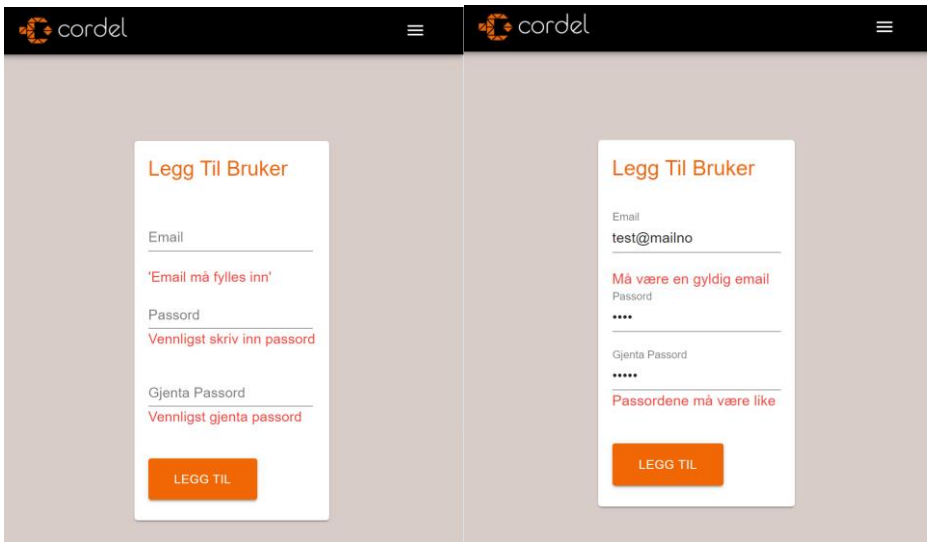
Siden for «legg til bruker» tillater brukeren å legge til nye brukere under sin bedrift. Det kreves at brukeren er logget inn for å bruke denne funksjonen.




*Figur 4.48 Legge til bruker*




*Figur 4.49 Bruker finnes allerede*



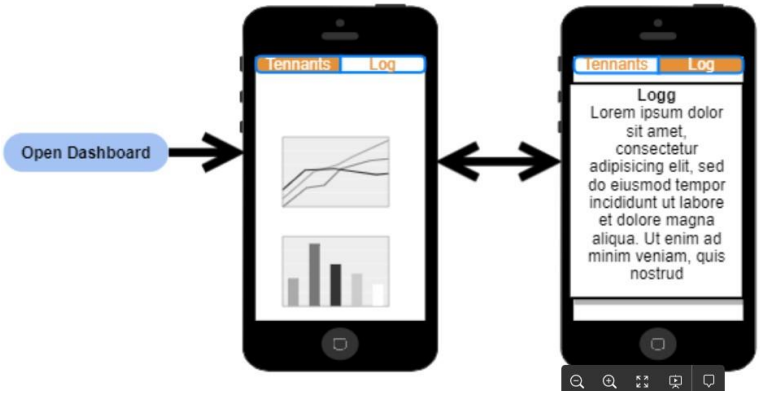

*Figur 4.50 Validering av inntastingsfelt*

Når «Legg til» knappen blir trykt, vil det bli sendt en spørring til backenden med eposten og passordet i inntastingsfeltene, for oppretting av bruker. Hvis responsen fra backenden er 200 vil tilbakemeldingen til brukeren være «Bruker lagt til» (figur 4.48), hvis responsen er 400 vil tilbakemeldingen være «Bruker finnes allerede» (figur 4.49). Figur 4.50 viser validering av inntastingsfelt.

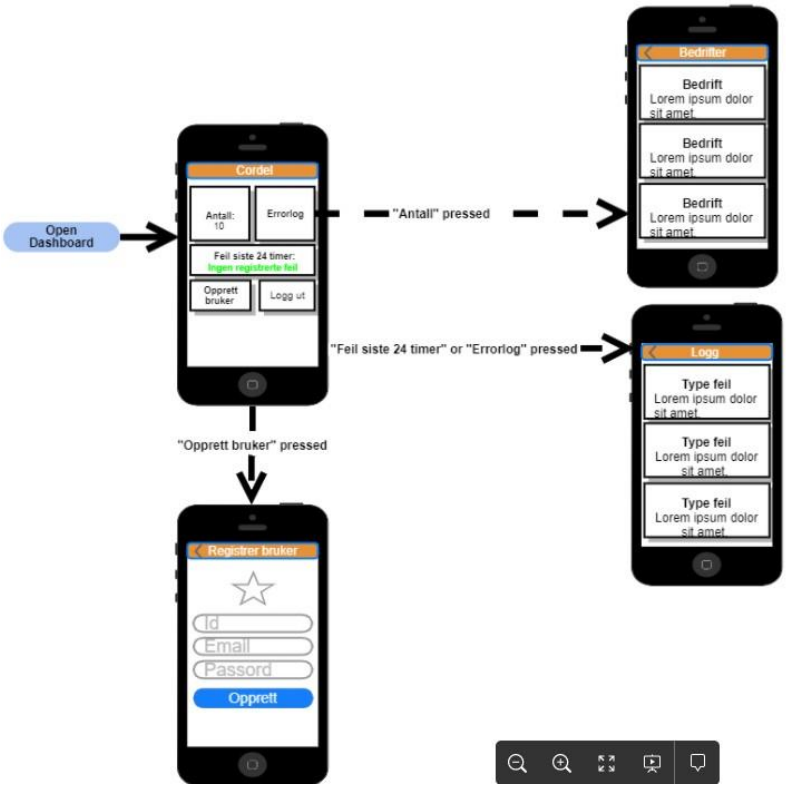
### 4.4.4 Mobilapplikasjon

Arbeidsgiver hadde i tillegg til en webløsning for sine kunder ønske om en mobilapplikasjon. Mobilapplikasjonen vi har laget gir muligheten for et overordnet innblikk i antall brukere og feilmeldinger som oppstår i systemet. Denne applikasjonen er kun ment for Cordel og deres ansatte, og har en egen kontroller i backend som utfører kall til mobilapplikasjonen.

Appen består av fem forskjellige sider; hovedskjerm, antall registrerte bedrifter, antall feilmeldinger, logg inn og brukerregistrering. Før utviklingsprosessen startet laget vi figurer og diagrammer. Figur 4.51 viser hvordan vi trodde første utkast av mobilapplikasjonen kom til å se ut, mens 4.52 viser en endelig versjon som ble utgangspunktet i vårt design for appen.




*Figur 4.51 Første utklast av mobildashbord*




*Figur 4.52 Endelig versjon av applikasjonen*

Dashbordet som demonstreres i figur 4.52 har flere trykkbare bokser som utfører flere ulike handlinger etter hvilken knapp som blir trykket på. Figuren ovenfor gir en forståelse om hvor du ender opp dersom brukeren trykker på de ulike boksene inne på hovedskjermen.

#### 4.4.4.1 Routing

For å bestemme hvilke sider som skal vises for brukeren brukte vi routing. Routing er i klassen «main.dart» og blir brukt for bestemme hvilken side som skal lastes ved ulike handlinger som vist i figur 4.53. Inititalroute bestemmer hvilken side som lastes når appen først starter. Ved å ha initialroute som «/login» vil det ikke være mulig å få tilgang til de andre sidene, da en må ha logget inn for å komme videre til andre sider.

```
void main() {
  runApp(MaterialApp(
    initialRoute: "/login",
    routes: {
      "/login": (context) => Login(),
      "/loading": (context) => Loading(),
      "/register": (context) => Register(),
      "/home": (context) => Home(),
      "/home/tennantListView": (context) => Tennant_View(),
      "/home/logListView": (context) => LogView()
    }
  ));  // MaterialApp
}
```

*Figur 4.53 Routing for mobilapplikasjon*

#### 4.4.4.2 API Client

For å sende API-forespørsler til serveren, har vi laget en getClient-funksjon (Figur 4.54). Denne funksjonen er universal og kan brukes av alle POST-forespørsler (2.5.1.1) i applikasjonen vår, og dette gjør vi for å oppnå høy kohesjon (2.4.1). Funksjonen vil returnere en response hvis alt går feilfritt, eller *null* hvis det oppstår en feil. For å benytte seg av getClient-funksjonen er det 3 parametere som blir sendt med. Disse er *customUrl*, *bodyInfo* og *token*. *CustomUrl* er URL-en som skal bli brukt under forespørselen. *BodyInfo* blir brukt for data som blir sendt med forespørselen til serveren. Dette kan for eksempel være innloggingsinformasjon. Til slutt sender man også med en JWT-token (2.6.2) til funksjonen. Denne tokenen vil bli brukt for å indentifisere brukeren når informasjon ankommer serveren. Hvis appen ikke får noen respons i løpet av 10 sekunder, vil en TimeoutException bli plukket opp. En error vil da bli lagret ved hjelp av *SharedPreferences* (3.7.19) og *null* vil bli returnert for å indikere at en feil har skjedd.

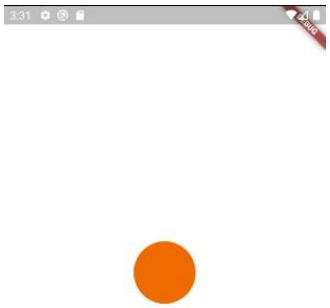
Etter at forespørselen er sendt og appen har mottatt en respons, vil den bruke response-koden (2.5.1.2) for å identifisere mulige problemer som har oppstått. Er respons-koden noe annet enn 200, vil programmet plukke opp dette og så lagre dette i *SharedPreferences* som senere blir brukt til å informere brukeren at en feil har oppstått. Samtidig vil funksjonen returnere *null.*

```dart
Future<http.Response> getClient(String customUrl, Map bodyInfo, String token) async {

  SharedPreferences preferences = await SharedPreferences.getInstance();

  String mainUrl = baseURL + customUrl;
  print("sender response");

  try {
    //Creates a post request with a timeout after 10 seconds
    http.Response response = await http.post(customUrl,
        headers: {
          "Content-Type": "application/x-www-form-urlencoded",
          'Accept': 'application/json',
          'Authorization': 'Bearer $token',
        },
      body: bodyInfo,
      encoding: Encoding.getByName("utf-8")

    ).timeout(Duration(seconds: 10));

    //If everything is OK
    if (response.statusCode == 200) {
      return response;

      //If user is not authorized
    } else if (response.statusCode == 401) {
      preferences.setString("error", "Unauthorized");
    }

    preferences.setString("error", "Error");
    //If there is other internal issues
    return null;

    //If the app cannot connect to the api
  } on TimeoutException catch (e) {
    preferences.setString("error", "Connection");
    return null;
  }
  //Catches other errors and sets an error that will be handled later
  catch (e) {
    preferences.setString("error", "Error");
    return null;
  }
}
```

*Figur 4.54 getClient funksjon for mobilapplikasjon*

### 4.4.4.3 Loading

Mellom de forskjellige interaktive sidene, valgte vi å implementere en loadingskjerm som vises mens nødvendig data lastes inn. Figur 4.55 viser en rund flate, som bytter mellom å rotere vertikalt og horisontalt.  For å forstå illustrasjonen bedre kan en tenke at sirkelen er som en mynt.

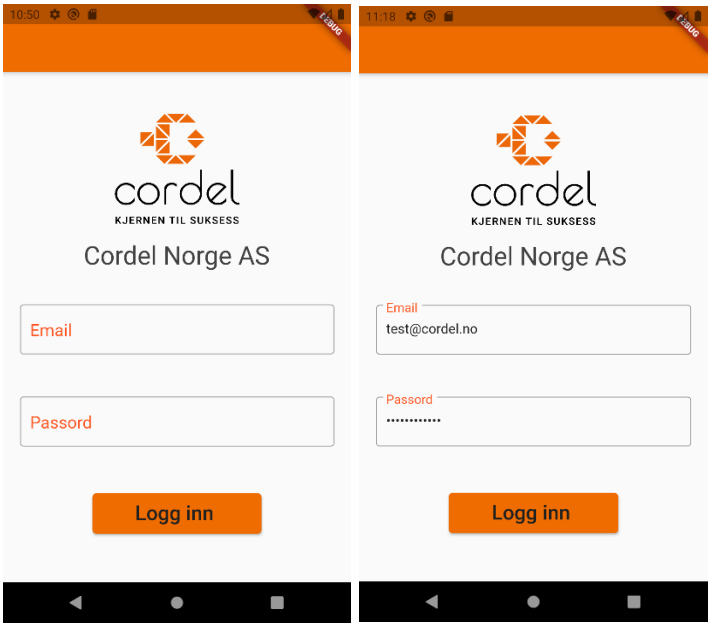


*Figur 4.55 Loadingskjermen til applikasjonen*

Når en bruker beveger seg fra for eksempel login-skjermen til home-skjermen, vil brukeren først bli sendt til loadingskjermen som utfører API-kallene og tar med seg responsen videre til neste skjerm. Applikasjonen vil da laste (figur 4.55) frem til den får en respons fra serveren. Brukeren blir så navigert til ønsket skjerm.

#### 4.4.4.4 Innlogging

Brukeren møtes med en innloggingsskjerm når appen først åpnes. Den inneholder to inntastingsfelt og en knapp for innlogging. Innloggingsskjerm er vist i figur 4.56.




*Figur 4.56 Innloggingsskjerm*

Som figur 4.56 viser så er det et veldig enkelt design. Etter Don Normans design prinsipper (2.1.1) er inntastingsfeltene tydelig markert etter hvilken type verdi som er ønsket for å unngå unødvendige brukerfeil. Knappen har også tydelig tekst som forklarer hva den blir brukt til.

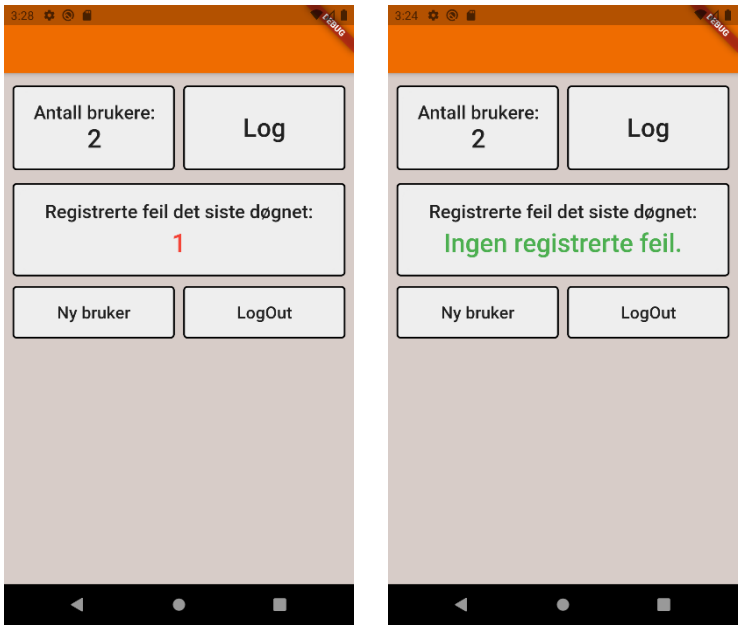
```
//Sends request to backend for password
Response response = await ApiClient().getClient(apiClient.baseURL + authURL, loginInfo, "");
SharedPreferences preferences = await SharedPreferences.getInstance();

//If there is an error with the http request
if(response == null) {
  String errorMessage = preferences.getString("error");

  //returns true if there is no error, returns false if there is an error
  errorCheck = await RequestErrorHandler().errorHandler(errorMessage, context);
}

//if there is no error, it will update the token
if(errorCheck) {
  preferences.setString("token", response.toString());
  return true;
}
return false;
}
```

*Figur 4.57 Innlogging av bruker*

Kodesnutten i figur 4.57 viser logikken applikasjonen benytter seg av for å sende en HTTP-POST forespørsel til backenden for autentisering (4.3.4.1). Dersom forespørselen returnerer 200 OK (2.5.1.2), blir en JWT-token (2.6.2) returnert til applikasjonen og lagret med bruk av Shared Preferences (3.7.17). Metoden brukt for å sende forespørselen er async (3.7.20) og vil derfor ikke droppe noen oppgaver underveis dersom responsen skulle ta lang tid.

#### 4.4.4.5 Hovedside

Etter innlogging blir brukeren ført videre til hovedsiden. Hovedsiden viser fem blokker som kan trykkes på for å få ulik informasjon, logge ut eller registrere en ny bruker. Som figur 4.58 viser, endres fargen mellom rød og grønn avhengig av om feilmeldinger har oppstått eller ikke i løpet av de siste 24 timene.




*Figur 4.58 Hovedskjerm til applikasjon*

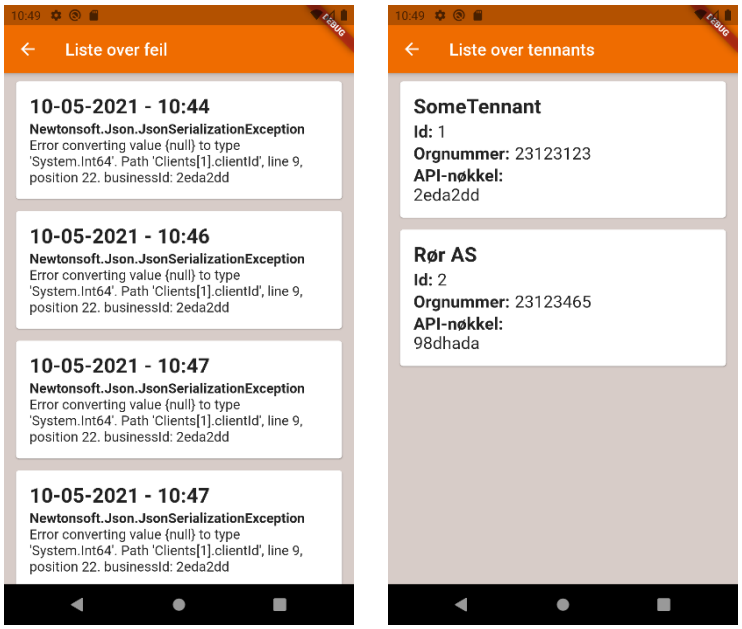
Verdiene som vises hentes med et kall fra en egen kontroller i backenden (4.3.6.4). Figur 4.58 viser forespørselen som blir sendt til backenden fra loading-skjermen (4.4.4.1) for å få ønskede verdier tilbake. Denne metoden er også async (3.7.22), og vil derfor vente med andre oppgaver frem til applikasjonen får en respons.

```
Future<void> getNumberOfTennantsAndErrors() async {
  Response response =
      await ApiClient().getClient(apiClient.baseURL + homeInfo, {}, "");

  recievedHomeInfo = jsonDecode(response.body);

  Navigator.pushReplacementNamed(context, "/home",
      arguments: recievedHomeInfo);
}
```

*Figur 4.59 Forespørsel om antall brukere og feilmeldinger*

### 4.4.4.6 Liste feilmeldinger og brukere

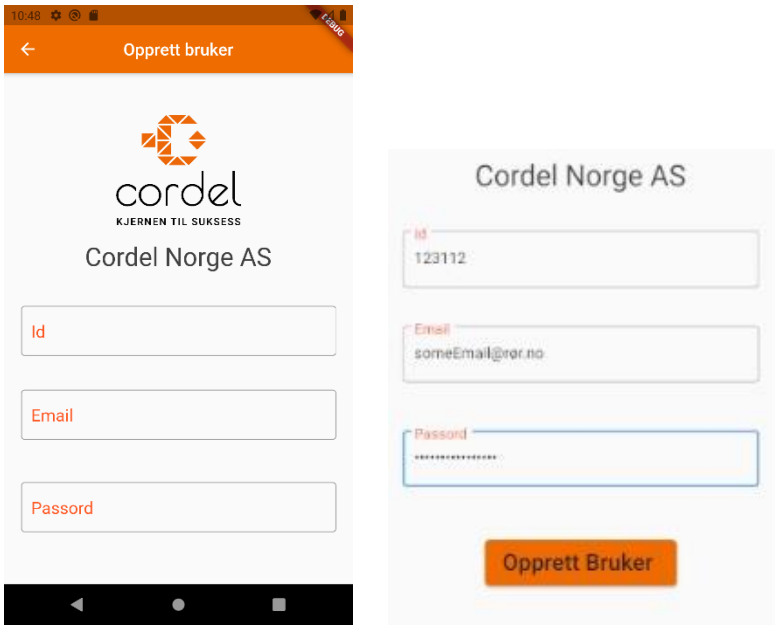
Ved å trykke på antall brukere, log, eller Registrerte feil siste 24 timene blir du tatt videre til en liste, med loading-skjermen som et mellomledd for å hente data. Listene er definert med to forskjellige view klasser, definert som «logView.dart» og «tennantListView», hvor logView blir brukt for alle feilmeldinger og feil siste 24 timer.




*Figur 4.60 Liste over feilmeldinger og brukere*

### 4.4.4.7 Registrering av bruker

Cordel må opprette første bruker for en bedrift før BI-systemet kan tas i bruk (4.4.3). Mobilapplikasjonen har en funksjon for dette, der Cordel kan trykke på «Ny bruker» knappen, som tar deg videre til en side for registrering av en ny bruker (figur 4.61).




*Figur 4.61 Registrering av første bruker*

Akkurat som i innloggingsskjermen (Figur 4.56) er registreringsskjermen designet etter Don Normans design prinsipper (2.1.1). En knapp med tekst gir en tydelig forklaring på hva den gjør, og inntastingsfeltene har tydelig tekst som forteller hva som er nødvendig i hvert felt.

## 4.5 Tester

Tester for prosjektet er implementet i både frontend og backend. I backend er det skrevet unittester (2.11.1) med bruk av Xunit (3.7.6) og Moq (3.7.7). Ved testing i mobilapplikasjonen er det utført widget tester (2.11.2) som tester ulike komponenter i en widget. Testene i React er implementert med React testing library (3.7.18). Figur 4.62 viser en kodesnutt fra en test til AuthenticationController (4.3.3.1) klassen og blir testet i et isolert miljø.

```
[Fact]
0 references | Run Test | Debug Test
public void loginReturnsExpectedResponse()
{
    // Values needed for the class to be tested
    string email = "someemail@mail.no";
    string password = "somePassword";
    var token = Guid.NewGuid().ToString();

    // Creates Mockdata and sets it to return wanted outcome used for testing.
    var mockConfig = new Mock<IConfiguration>();
    mockConfig.Setup(c => c["Jwt:Key"]).Returns(token);

    var mockLoginDb = new Mock<ILoginDatabaseContext>();
    mockLoginDb.Setup(l => l.findUserByMail(email)).Returns(TestUser());

    // Creates a controller-object to test, warehousedb set to null because it's not used in this method.
    var controller = new JWTAuthenticationController(mockConfig.Object, mockLoginDb.Object, null);
    // Sets result to the method that is being tested
    var resultOk = controller.login(email, password);

    // changes tennantId to be able to retrieve unauthorized result
    password = "someNewPassword";
    var resultUnAuth = controller.login(email, password);

    //Verifies that the object is of a given type
    var viewResultOk = Assert.IsType<OkObjectResult>(resultOk);
    var viewResultUnAuth = Assert.IsType<UnauthorizedResult>(resultUnAuth);

    // Verifies that two objects are equal. This checks if the responsecode is 200.
    Assert.Equal(200, viewResultOk.StatusCode);
    Assert.Equal(401, viewResultUnAuth.StatusCode);
}
```

*Figur 4.62 Test skrevet for AuthenticationController for login*

```
/*
*   Returns a user as dummydata for testing
*/
3 references
private User TestUser()
{
    var user = new User
    {
        id = 1,
        Email = "someemail@mail.com",
        password = BCrypt.Net.BCrypt.HashPassword("somePassword"),
        tennant = TestTennant(),
        tennantFK = 1,
        role = "User"
    };
    return user;
}
```

*Figur 4.63 Dummydata for en bruker*

Testdataen i figur 4.63 oppretter en bruker som er identiske med ekte brukerobjeker. Ettersom login bruker biblioteket Bcrypt (3.7.8) for å verifisere passord, er det nødvendig med hashing (2.9.1) av passordene for dummydataene. Testing har bidratt til høyere kvalitet av koden vår, og gjennom testing fant vi muligheter for å skape høyere kohesjon (2.3.1) og lavere kobling (2.3.2) i backenden med å ta i bruk i interface klasser for context klassene. Vi valgte da også å flytte alle spørringer med LINQ (3.7.2) til context klassene.

For utviklingen av tester til mobilapplikasjonen lagde vi widget tester (2.11.2). I figur 4.64 viser vi en test som tester inntastingsfelt og en knapp.

```
void main() {
  testWidgets('Test login widgets', (WidgetTester tester) async{
    // Widgets needed
    var emailField = find.byKey(ValueKey("emailField"));
    var passField = find.byKey(ValueKey("passwordField"));
    var loginBtn = find.byKey(ValueKey("emailField"));

    // Execute the test
    await tester.pumpWidget(MaterialApp(home: Login()));
    await tester.enterText(emailField, "someemail@email.no");
    await tester.enterText(passField, "somePassword");
    await tester.tap(loginBtn);
    // rebuilds widget
    await tester.pump();

    // Check output
    expect(find.text("someemail@email.no"), findsOneWidget);
    expect(find.text("somePassword"), findsOneWidget);

  });
}
```

*Figur 4.64 Test for loginsiden i mobilapplikasjonen*

For nettsiden lagde vi tester i React-applikasjonen, der vi brukte React testing library (3.7.18) for å lage widget tester. I figur 4.65 viser vi en test som sjekker om rendering i nettsiden fungerer som planlagt.

```
test("Renders Dashboard correctly", async () => {
  await act(async () => {
    const { getByTestId } = render(<Dashboard />);
    expect(getByTestId("chartcontainer")).toBeInTheDocument();
  });
});

test('Dropdown renders', async() =>{
  await act(async () => {
    const { getByTestId } = render(<Dashboard />);
    expect(getByTestId("dropDownButton")).toBeTruthy();

});
});
```

*Figur 4.65 Dashboard Widget test fra React*

# 5 DRØFTING

I dette kapittelet ser vi på ulike perspektiver av utførelsen og samhandlingen vi har gjort i dette prosjektet. Først går vi gjennom hvilke tanker vi har rundt resultatene og utviklingen av de forskjellige komponentene i systemet. Til slutt drøfter vi selve prosjektgjennomføringen.

## 5.1 Resultater

Ved produktutvikling i samarbeid med oppdragsgivere er det alltid rom for misforståelser. Derfor har flere deler av prosjektet vårt gjennomgått endringer fortløpende i henhold til oppdragsgiverens ønske. I tillegg har vi tilegnet oss mye kunnskap rundt de forskjellige teknologiene som er brukt underveis i prosjektet. Dette har ført til at tidligere utfordringer ble redigert og endret, da vi fant bedre måter å løse problemene på.

### 5.1.1 Datavarehus

Gjennom prosjektet har vi gjort flere endringer i datavarehuset. Dette i form av hvilke entiteter som skulle lagres, endres og/eller fusjoneres. Vårt beste eksempel på dette omhandlet kundefordringer, da vi i starten ikke hadde noen bilag i datavarehuset, men ferdigkalkulerte kundefordringer. Når vi fikk mer informasjon endret vi dette slik at kundefordringene ble utregnet i API-et ved bruk av bilag. Dette førte til at vi også gjorde en del endringer i datamodellen og API-et.

### 5.1.2 Rest API

Ved utvikling av de forskjellige kallene til datavarehuset har vi gjort flere endringer. Til å starte med var det simple kall som returnerte for mye informasjon og ville tatt lang tid å utføre. Siden vi endret datamodellene underveis i prosjektet, førte det også til at vi gjorde endringer i API kallene.

Et godt eksempel på et API-kall vi endret mye er kallet for kundefordringer. I starten av prosjektet var det noen misforståelser som førte til at vi trodde kundefordringene ville være ferdigkalkulert da de kom til datavarehuset. Etter flere møter med Cordel kom vi frem til hvordan de så for seg at kundefordringene skulle fremstilles i nettsiden, og hva vi må benytte oss av for å utregne disse. Sluttresultatet av dette kallet er noe vi er fornøyd med. Vi har unngått flere forløkker innad i hverandre og algoritmen er veldig fleksibel. Den er godt dokumentert og fungerer for alle former for filter. Algoritmen kan også enkelt endre intervallet for oppdateringer.

Selv om API-kallene har sine fordeler, er det fortsatt rom for optimalisering. Mange av algoritmene er avhengige av å sortere listen som hentes fra databasen. Siden dette også tar tid, finnes det muligheter for å holde databasen sortert på forhånd. Slike former for optimalisering ble nedprioritert siden Cordel ikke la stor vekt på det.

Vi har lagt ned mye arbeid med utarbeiding av API-et vårt, og er noe vi har fått god tilbakemelding på fra Cordel. Ved utviklingen la vi tidlig stor vekt på god mappestruktur og innholdet i de ulike klassene. Med dette som hovedfokus har vi gjort det enklere for Cordel å starte sin videreutvikling av API-et.

Det at prosjektet omhandler flere konsepter fra regnskap gjør at det har vært mer utfordrende for oss som ikke har så mye erfaring med dette fra før. Det er med stor sannsynlighet annen informasjon bedrifter ønsker å få visualisert i en graf, som enkelt kunne blitt tilføyd under de forskjellige kalkuleringene gjort i API-et. Vi valgte likevel å fokusere på det som ble etterspurt i en MVP, men vi så tidlig muligheter for videreutvikling, som er godt tilrettelagt for Cordel når de tar over.

### 5.1.3 Docker

Før prosjektet hadde vi lite kunnskap om docker og det tok tid å sette seg inn i det. Vi brukte en hel dag hvor alle i prosjektgruppen jobbet for å lære seg og forstå hva docker var i stand til å gjøre. Dette ga oss en god forståelse av hvordan vi kunne utnytte docker i vårt prosjekt.

I tillegg til god dokumentasjon fra internett var vi heldige og fikk god hjelp fra vår oppdragsgiver Cordel. Vi har fått muligheten til å snakke med flere ulike fagfolk, hvor vi blant annet har fått tilbakemeldinger og tips om hvordan en bør bruke docker.

### 5.1.4 Database

Alle i prosjektgruppen har tidligere erfaringer med postgreSQL (3.4.2) gjennom databasefaget vi har hatt i sammenheng med vår studie. Vi har brukt EFCore (3.7.1) som fungerer godt sammen med relasjonsdatabaser, hvor vi har brukt LINQ (3.7.2). For oss var det veldig gunstig å ta i bruk LINQ for spørringer til databasene da vi har unngått mange og lange SQL spørringer.

Vi brukte noe tid på å få databasene til å fungere som ønsket, men ble veldig fornøy med løsningen da vi fikk databasene til å fungere. Å bruke to databaser som også må snakke sammen var utfordrende, men en egen database for innlogging var nødvendig for oss i dette prosjektet, da dette var fastslått av Cordel.

Designet til datamodellen tok tid å bygge opp, men var nødvendig for å holde en og oversiktlig og fin utviklingsprosess. Vi har gjort flere endringer i datamodellen, både entiteter som skal være koblet sammen og hvilken type forhold de skal ha mot hverandre.

### 5.1.5 Sikkerhet

Oppdragsgiver var tidlig ute med å si at det er en MVP de ville ha og at vi ikke skulle legge for mye vekt på sikkerhet. Vi valgte derfor å fokusere på de grunnleggende stegene innenfor sikkerhet som JWT-token, HTTPS, hashing, salting og LINQ-spørringer. Ved å implementere dette var sikkerheten god nok for en MVP ifølge oppdragsgiver. Ved videre utvikling og integrering av systemet vil det implementeres mer avanserte metoder for å øke sikkerheten til systemet.

Et sikkerhetsbrudd vi ser for oss ved bruksfeil er om en innlogget bruker ikke logger ut av nettsiden før personen forlater enheten som er logget inn. Siden alle brukere har mulighet til å legge til en kollega kan dette utnyttes dersom enheten tas i bruk av en person som ikke skal ha tilgang til informasjon om denne bedriften. Vi diskuterte med oppdragsgiver om vi skulle implementere enda en rolle i systemet, slik at kun denne rolle hadde mulighet til å legge til andre brukere hos samme tennant. Dette var ikke nødvendig for en MVP, så vi prioriterte andre funksjonaliteter av systemet istedenfor.

### 5.1.6 Testing

Testing under prosjektet ble utviklet sent i utviklingsprosessen og er noe vi kunne hatt mer fokus på underveis. I tidligere prosjekter har vi hatt lite fokus på testing, så det har vært nytt for hele prosjektgruppen. Vi har likevel fått en god forståelse av hvor viktig testing er og hvordan det kan øke kvaliteten på koden. Det gikk mye tid til undersøking av testing, og vi så da hvor nyttig det kan være å jobbe med test-driven-development ved utvikling av systemer. Selv om vi så nytten av test-driven-development har vi ikke tatt nytte av dette i vårt prosjekt.

Vi skrev tester for backend, nettside og mobilapplikasjon (4.5). Enkelte deler av både frontend og backend ufullstendig testmessig, da vi kunne hatt flere tester. Vi var ganske sent ute med å begynne med å skrive tester, men vi har likevel fått et godt resultat i de testene vi har.

### 5.1.7 React og Typescript

Cordel hadde ønsker om hvilke teknologier og programmeringsspråk vi skulle bruke for nettsiden. Dette ga oss utfordringer da ingen av oss hadde tidligere erfaring med verken React (3.7.15) eller Typescript (3.6.2). Dette krevde at vi brukte mye tid i starten på å studere teknikker og nødvendige rammeverk for oppbygging av en nettside. Heldigvis fant vi mye dokumentasjon og flere videoer med høy kvalitet på Youtube som hjalp oss under hele utviklingsperioden.

### 5.1.8 Flutter

Flutter var ukjent for hele prosjektgruppen ved starten av mobilapplikasjonsutviklingen. Vi visste hva det var, men det var noe ingen av oss hadde tidligere erfaringer med. Det førte til at vi ble nødt til å bruke mye tid på å lære det. Det gjorde vi ved å utvikle en PoC. Ved å lese

mye dokumentasjon og bruke Youtube, fikk vi mye grunnleggende kunnskap om flutter slik at vi ble mer komfortable med å bruke det for den faktiske løsningen.

### 5.1.9 Brukervennlighet og design

I starten av frontend-utviklingen bestemte vi oss for å dele gruppen i to. Utviklingen av mobilapplikasjonen og nettsiden skjedde på samme tid og det var da lite kommunikasjon om design. Det gjorde at designet av mobilapplikasjonen og nettsiden ikke samsvarte med hverandre. Vi brukte derfor tid mot slutten på synkronisering av design mellom applikasjonene, hvor samme fargekoder, likheter i validering og knappedesign var i fokus.

### 5.1.10 Kommunikasjon

Gjennom prosjektet har vi jobbet ved bruk av hjemmekontor. Med respekt for korona har vi valgt å ikke benyttet oss noe særlig av skolen, men heller brukt digitale platformer som Discord og Microsoft-teams for å holde møter (3.4.5).

I forprosjektsrapporten bestemte vi faste arbeidstider, hvor den planlagte møteplassen skulle være på skolen. Grunnet pandemien har vi behandlet Discord (3.4.5) som vår arbeidsplass. Her har vi møtt opp hver dag til avtalt tid, og er alltid tilgjengelig i løpet av arbeidsdagen.

Møtene med arbeidsgiver og veileder var i startsfaen utført over nett ved bruk av Microsoft Teams (3.4.5). Etter hvert som koronasmitten sank flyttet vi møtelokalene fra det digitale til det fysiske. Cordel stilte med lokaler, drikke og noe godt å spise til hvert møte som ble holdt. Her møttes vi annenhver uke for å holde oppdragsgiver oppdatert på hvordan produktet så ut, diskutere produktet og andre ting som oppstod underveis. Etter hvert som smitten økte igjen flyttet vi møtene tilbake til den digitale verden. Når tidspresset økte, sendte vi flere e-poster med spørsmål til oppdragsgiver. Oppdragsgiver, veileder og ansatte hos Cordel var raskt ute med svar, og stilte alltid opp til et møte på kort varsel når det skulle være nødvendig. Det er noe vi har satt veldig stor pris på, og har hjulpet oss mye i bachelorprosjektet.

## 5.2 Prosjektgjennomføringen

I denne seksjonen tar vi for oss punkter om hvordan vår gjennomføring av prosjektet har vært. Dette omhandler teknikker og verktøy som har hatt innvirkning på hvordan vi har jobbet sammen under prosjektet.

### 5.2.1 Utviklingsmetodikk

I vår gjennomføring av prosjektet har vi brukt utviklingsmetodikken Scrum (2.8.1). Vi har vært konsistent i bruken av to ukers sprinter og har utført daglig scrum (2.8.1.2) hver arbeidsdag. Etter hver sprint brukte vi tid på å evaluere hvordan arbeidsperioden har vært ved å notere både positive og negative sider. Vi startet nye sprinter med en planleggingsfase etterfulgt av et møte med oppdragsgiver. Møte med veileder ble holdt uavhengig av sprint, men med to ukers mellomrom.

To ukers sprinter har fungert bra for oss da vi har hatt tid til å få mye gjort. I tillegg er ikke perioden for lang eller vanskelig å planlegge. Vi har enkelt kunne vise og forklare det vi jobbet med forrige sprint og samtidig vise hva vi skulle jobbe med i neste sprint. Møter med veileder har blitt arrangert når det har vært behov, men med hyppige mellomrom. Dette for å stille konkrete spørsmål om oppgaven, men også for å vise veileder vår progresjon.

Det å jobbe med Scrum (2.8.1) som arbeidsmetodikk har vært fordelaktig for oss i planleggingen av sprintene. Vi har enkelt kunne vurdere hvor langt vi har kommet og prioritere det som til enhver tid var viktigst for best mulig fremgang i prosjektet.

#### 5.2.1.2 Jira Backlog

For å ha kontroll over arbeidsoppgavene satte vi opp en backlog (2.8.1.3) for å få en oversikt over hva som måtte gjøres for å få en MVP. Denne backloggen formulerte vi i webapplikasjonen Jira, der vi enkelt kunne redigere backloggen. I sprintplanleggingsfasen hentet vi oppgaver fra backloggen og la de til en sprint. Å ha en backlog har gjort det enkelt for oss å planlegge frem i tid, ved at vi til enhver tid vet hvor mye jobb som er igjen for å fullføre prosjektet.

Vi merket oppgavene i backloggen med viktighet og antatt tidsbruk, som oss en pekepinn på hvilke oppgaver som skulle legges inn i de gjeldene sprinten.

#### 5.2.1.3 Tidsplanlegging

I forprosjektsrapporten satte vi opp en plan over hovedaktiviteter for gjennomføring av prosjektet. I denne planen prøvde vi å estimere tidsbruk på de forskjellige hovedaktivitetene for å få et estimat på hvor lang tid enkelte deler av prosjektet kom til å ta. Selv om det

selvfølgelig ble endringer i planen, hjalp det å ha en plan når vi planla sprinter for å vite omfanget av oppgavene.

#### 5.2.1.4 Git

Gjennom hele prosjektet benyttet vi oss av versjonskontrollverktøyet Git (3.4.1). Vi lagde en organisasjon i Github med navn «Datavarehus». Deretter delte vi prosjektet opp i tre forskjellige repositories, et for backend (4.3), et for mobilapplikasjonen og et for nettsiden (4.4). For å enklere benytte oss av Git brukte vi GitKraken, et GUI som lar oss utføre handlinger opp mot Github.

Til å starte med var det usikkerhet med hvilke gitignore-filer vi skulle benytte oss av. Siden vi først skulle lage PoC av alle delene fikk vi testet forskjellige gitignore filer som vi genererte med hjelp av www.gitignore.io (98). Etter hvert som vi utviklet la vi til flere gitignore-filer hvor det var nødvendig.

# 6 KONKLUSJON

Prosjektoppgaven har vært en lærerik opplevelse som har gitt oss mange gode erfaringer. Det har vært spennende og interessant for oss å være med på å utvikle et produkt fra starten av med en reell oppdragsgiver og en erfaren veileder. Vi har fått prøve oss på mange nye, spennende og relevante teknologier. Etter mye arbeid med disse teknologiene har vi fått mye relevant arbeidserfaring som har vært med på å videreutvikle oss som dataingeniører.

Det har vært en lang prosess med flere lange dager, men med en motiverende oppgave som vi alle synes er spennende har ikke dette vært noe stort problem. Vi har klart å utvikle et system vi er fornøyde med, som utfyller kriteriene gitt av arbeidsgiver. Sluttresultatet er utført på en måte hvor Cordel kan sende inn data til systemet, opprette bedrifter og knytte data opp mot bedriftene automatisk. Bedriftene som skal bruke Bi-systemet kan logge seg inn for å få en oversiktlig analyse av sin bedrift. Cordel kan benytte seg av mobilapplikasjonen og få en oversikt over brukerne og feilmeldinger som oppstår i datavarehuset.

Gjennom utviklingen møtte vi på flere utfordringer. Vi er veldig stolte av prosjektet vi har laget siden det fungerer som planlagt etter krav fra Cordel. En av de største utfordringene vi møtte var at prosjektet skulle utvikles i språk vi ikke hadde erfaring med fra før. Det bød på flere bekymringer i starten, men resulterte i at vi fikk bredere kunnskap innenfor faget. Vi har lært om nye konsepter som «proof of concept» som vi benyttet oss av for å bli kjent med de nye språkene.

Vi har klart å holde oss til en smidig arbeidsmetodikk gjennom hele prosjektet, hatt god kommunikasjon innad i gruppen, og regelmessige møter med både veileder og arbeidsgiver.

Kjernefunksjonene som var satt som krav til prosjektet har vi implementert. Det er likevel noen deler av prosjektet som er blitt nedprioritert. Overføring av systemet til Azure Cloud var egentlig planlagt i sluttperioden, men etter behov for viktigere endringer og lite tid ble dette satt på vent. Dette ser vi likevel ikke på som noe kritisk med hensyn til gjennomføringskvaliteten av prosjektet.

Systemet er et godt grunnlag for videreutvikling for Cordel. Koden er tydelig og godt dokumentert og det er lagt til rette for å gjøre endringer. I tillegg er det grundig gjort rede for hvordan systemet skal tas i bruk og hvordan det fungerer. Vi har derfor fått tilbakemelding på at Cordel er fornøyd med hele produktet og samarbeidet (vedlegg 11), og det synes vi er veldig hyggelig.

# 7 REFERANSER

1. Prosjektstyringsverktøy for håndverksbransjen [Internett]. Cordel. [sitert 5. mai 2021]. Tilgjengelig på: https://cordel.no/
2. Base64 URL [Internett]. [sitert 22. februar 2021]. Tilgjengelig på: https://www.base64url.com/
3. Granevang M. frontend. I: Store norske leksikon [Internett]. 2020 [sitert 16. mai 2021]. Tilgjengelig på: http://snl.no/frontend
4. Granevang M. backend. I: Store norske leksikon [Internett]. 2020 [sitert 16. mai 2021]. Tilgjengelig på: http://snl.no/backend
5. What Are Wireframes? | Wireframing Academy | Balsamiq [Internett]. [sitert 16. mai 2021]. Tilgjengelig på: https://balsamiq.com/learn/articles/what-are-wireframes/
6. What is widget? - Definition from WhatIs.com [Internett]. WhatIs.com. [sitert 16. mai 2021]. Tilgjengelig på: https://whatis.techtarget.com/definition/widget
7. About repositories - GitHub Docs [Internett]. [sitert 16. mai 2021]. Tilgjengelig på: https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/about-repositories
8. Git Guides - git commit [Internett]. GitHub. [sitert 16. mai 2021]. Tilgjengelig på: https://github.com/git-guides/git-commit
9. About branches - GitHub Docs [Internett]. [sitert 16. mai 2021]. Tilgjengelig på: https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-branches
10. What is an SDK? What do SDKs do? Find out here | Adjust | Adjust [Internett]. [sitert 16. mai 2021]. Tilgjengelig på: https://www.adjust.com/glossary/sdk/
11. TypeScript is a superset of JavaScript, meaning it's a layer around JS with more methods and that… | by Diogo Spínola | Medium [Internett]. [sitert 16. mai 2021]. Tilgjengelig på: https://medium.com/@daspinola/typescript-is-a-superset-of-javascript-meaning-its-a-layer-around-js-with-more-methods-and-that-46bbc9368be1
12. Kouraklis J. Introducing Delphi ORM: Object Relational Mapping Using TMS Aurelius [Internett]. Berkeley, CA: Apress; 2019 [sitert 16. mai 2021]. Tilgjengelig på: http://link.springer.com/10.1007/978-1-4842-5013-6
13. auth0. JWT.IO - JSON Web Tokens Introduction [Internett]. [sitert 26. mai 2021]. Tilgjengelig på: http://jwt.io/

14. Using API Keys | Maps JavaScript API [Internett]. Google Developers. [sitert 19. mai 2021]. Tilgjengelig på: https://developers.google.com/maps/documentation/javascript/get-api-key?hl=nb
15. Hauffe T. Design [Internett]. Norbok. Oslo: Cappelen; 1996 [sitert 4. mai 2021]. 192 s. (Cappelens kulturguider). Tilgjengelig på: https://urn.nb.no/URN:NBN:no-nb_digibok_2010091403053
16. Norman DA. Design rules based on analyses of human error. Commun ACM. april 1983;26(4):254–8.
17. Strøm E. Datamodellering -praksis og teori. MetodeData A.S; 2000. 151 s.
18. Edgar B. Datamodellering - praksis og teori. 3.
19. What is a Data Warehouse? [Internett]. [sitert 5. mai 2021]. Tilgjengelig på: https://www.oracle.com/database/what-is-a-data-warehouse/
20. Hva er Business Intelligence? [Internett]. [sitert 11. mai 2021]. Tilgjengelig på: https://www.visma.no/business-intelligence/hva-er-business-intelligence/
21. OOP (Object-Oriented Programming) Definition [Internett]. [sitert 5. mai 2021]. Tilgjengelig på: https://techterms.com/definition/oop
22. Barnes DJ, Kölling M. Objects First with Java. Sixth Edition. 630 s.
23. Johnsen R. kommunikasjonsprotokoll. I: Store norske leksikon [Internett]. 2020 [sitert 5. mai 2021]. Tilgjengelig på: http://snl.no/kommunikasjonsprotokoll
24. An overview of HTTP - HTTP | MDN [Internett]. [sitert 5. mai 2021]. Tilgjengelig på: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
25. HTTP request methods - HTTP | MDN [Internett]. [sitert 4. mai 2021]. Tilgjengelig på: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
26. HTTP - Methods - Tutorialspoint [Internett]. [sitert 4. mai 2021]. Tilgjengelig på: https://www.tutorialspoint.com/http/http_methods.htm
27. HTTP Methods GET vs POST [Internett]. [sitert 4. mai 2021]. Tilgjengelig på: https://www.w3schools.com/tags/ref_httpmethods.asp
28. HTTP response status codes - HTTP | MDN [Internett]. [sitert 7. mai 2021]. Tilgjengelig på: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
29. Hva er HTTPS? Og hva er forskjellen fra HTTP? | Nettrafikk [Internett]. Nettrafikk.no. 2015 [sitert 5. mai 2021]. Tilgjengelig på: https://nettrafikk.no/hva-er-https/
30. REST Principles and Architectural Constraints [Internett]. REST API Tutorial. 2018 [sitert 3. mai 2021]. Tilgjengelig på: https://restfulapi.net/rest-architectural-constraints/

31. Avraham SB. What is REST — A Simple Explanation for Beginners, Part 2: REST Constraints [Internett]. Medium. 2017 [sitert 3. mai 2021]. Tilgjengelig på: https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-2-rest-constraints-129a4b69a582
32. What is JSON [Internett]. Squarespace. [sitert 5. mai 2021]. Tilgjengelig på: https://developers.squarespace.com/
33. Atlassian. Is the Agile Manifesto Still a Thing? [Internett]. Atlassian. [sitert 5. mai 2021]. Tilgjengelig på: https://www.atlassian.com/agile/manifesto
34. Atlassian. Scrum - what it is, how it works, and why it's awesome [Internett]. Atlassian. [sitert 5. mai 2021]. Tilgjengelig på: https://www.atlassian.com/agile/scrum
35. Schwaber K, Sutherland DrJ. Ken Schwaber and Dr. Jeff Sutherland Update the Scrum Guide on the 25th Anniversary of the Scrum Framework [Internett]. Scrum.org. [sitert 18. mars 2021]. Tilgjengelig på: https://www.scrum.org/resources/ken-schwaber-and-dr-jeff-sutherland-update-scrum-guide-25th-anniversary-scrum-framework
36. What is a Daily Scrum? [Internett]. Scrum.org. [sitert 11. mars 2021]. Tilgjengelig på: https://www.scrum.org/resources/what-is-a-daily-scrum
37. What is a Product Backlog? [Internett]. Scrum.org. [sitert 11. mars 2021]. Tilgjengelig på: https://www.scrum.org/resources/what-is-a-product-backlog
38. Atlassian. Agile Scrum Roles [Internett]. Atlassian. [sitert 11. mars 2021]. Tilgjengelig på: https://www.atlassian.com/agile/scrum/roles
39. Pair Programming: Does It Really Work? [Internett]. Agile Alliance |. 2015 [sitert 4. mai 2021]. Tilgjengelig på: https://www.agilealliance.org/glossary/pairing/
40. What is hashing? [Internett]. Educative: Interactive Courses for Software Developers. [sitert 4. mai 2021]. Tilgjengelig på: https://www.educative.io/edpresso/what-is-hashing
41. Whitman ME, Mattord HJ. Principles of Information Security. Sixth Edition. 2017. 750 s.
42. Secure Salted Password Hashing - How to do it Properly [Internett]. [sitert 5. mai 2021]. Tilgjengelig på: https://crackstation.net/hashing-security.htm#salt
43. Adding Salt to Hashing: A Better Way to Store Passwords [Internett]. Auth0 - Blog. [sitert 4. mai 2021]. Tilgjengelig på: https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/
44. SQL Injection | OWASP [Internett]. [sitert 13. mai 2021]. Tilgjengelig på: https://owasp.org/www-community/attacks/SQL_Injection

45. Brute Force Attack: Definition and Examples [Internett]. www.kaspersky.com. 2021 [sitert 4. mai 2021]. Tilgjengelig på: https://www.kaspersky.com/resource-center/definitions/brute-force-attack

46. What is a Brute Force Attack? [Internett]. Forcepoint. 2018 [sitert 4. mai 2021]. Tilgjengelig på: https://www.forcepoint.com/cyber-edu/brute-force-attack

47. Hovde K-O, Grønmo S. algoritme. I: Store norske leksikon [Internett]. 2020 [sitert 13. mai 2021]. Tilgjengelig på: http://snl.no/algoritme

48. TDT4120: Algoritmer og datastrukturer - Wikipendium [Internett]. [sitert 13. mai 2021]. Tilgjengelig på: https://www.wikipendium.no/TDT4120_Algoritmer_og_datastrukturer/nb/

49. Types of Software Testing: Different Testing Types with Details [Internett]. [sitert 18. mai 2021]. Tilgjengelig på: https://www.softwaretestinghelp.com/types-of-software-testing/

50. What is Software Testing? Definition, Basics & Types [Internett]. [sitert 18. mai 2021]. Tilgjengelig på: https://www.guru99.com/software-testing-introduction-importance.html

51. What Is Unit Testing? [Internett]. smartbear.com. [sitert 3. mai 2021]. Tilgjengelig på: https://smartbear.com/learn/automated-testing/what-is-unit-testing/

52. Testing Flutter apps [Internett]. [sitert 12. mai 2021]. Tilgjengelig på: https://flutter.dev/docs/testing

53. Testing · React Native [Internett]. [sitert 12. mai 2021]. Tilgjengelig på: https://reactnative.dev/docs/testing-overview

54. Proof of Concept: The Complete Guide for Startups [Internett]. Pixelfield blog. 2020 [sitert 13. mai 2021]. Tilgjengelig på: https://pixelfield.co.uk/blog/proof-of-concept-the-complete-guide-for-startups/

55. AS E. MVP — akkurat nok til at det funker [Internett]. Medium. 2016 [sitert 13. mai 2021]. Tilgjengelig på: https://medium.com/escio/mvp-akkurat-nok-til-at-det-funker-b57c7dea19d6

56. Git [Internett]. [sitert 14. mai 2021]. Tilgjengelig på: https://git-scm.com/

57. Atlassian. What is Git: become a pro at Git with this guide | Atlassian Git Tutorial [Internett]. Atlassian. [sitert 14. mai 2021]. Tilgjengelig på: https://www.atlassian.com/git/tutorials/what-is-git

58. Hello World · GitHub Guides [Internett]. [sitert 14. mai 2021]. Tilgjengelig på: https://guides.github.com/activities/hello-world/

59. PostgreSQL: About [Internett]. [sitert 5. mai 2021]. Tilgjengelig på: https://www.postgresql.org/about/
60. Docker overview [Internett]. Docker Documentation. 2021 [sitert 8. mai 2021]. Tilgjengelig på: https://docs.docker.com/get-started/overview/
61. API Client for REST, SOAP, & GraphQL Queries [Internett]. Postman. [sitert 5. mai 2021]. Tilgjengelig på: https://www.postman.com/product/api-client/
62. Documentation for Visual Studio Code [Internett]. [sitert 5. mai 2021]. Tilgjengelig på: https://code.visualstudio.com/docs
63. Meet Android Studio [Internett]. Android Developers. [sitert 6. mai 2021]. Tilgjengelig på: https://developer.android.com/studio/intro?hl=nb
64. DataGrip [Internett]. DataGrip Help. [sitert 9. mars 2021]. Tilgjengelig på: https://www.jetbrains.com/help/datagrip/2021.1/meet-the-product.html
65. What is Flutter and Why You Should Learn it in 2020 [Internett]. freeCodeCamp.org. 2019 [sitert 5. mai 2021]. Tilgjengelig på: https://www.freecodecamp.org/news/what-is-flutter-and-why-you-should-learn-it-in-2020/
66. Typed JavaScript at Any Scale. [Internett]. [sitert 3. mai 2021]. Tilgjengelig på: https://www.typescriptlang.org/
67. CSS: Cascading Style Sheets | MDN [Internett]. [sitert 5. mai 2021]. Tilgjengelig på: https://developer.mozilla.org/en-US/docs/Web/CSS
68. BillWagner. A Tour of C# - C# Guide [Internett]. [sitert 5. mai 2021]. Tilgjengelig på: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/
69. ajcvickers. Overview of Entity Framework Core - EF Core [Internett]. [sitert 16. april 2021]. Tilgjengelig på: https://docs.microsoft.com/en-us/ef/core/
70. BillWagner. Language-Integrated Query (LINQ) (C#) [Internett]. [sitert 16. april 2021]. Tilgjengelig på: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/
71. LINQ & SQL Injection [Internett]. [sitert 4. mai 2021]. Tilgjengelig på: https://entityframework.net/linq-prevent-sql-injection
72. Eliminate SQL Injection Attacks Painlessly with LINQ [Internett]. [sitert 4. mai 2021]. Tilgjengelig på: https://www.devx.com/dotnet/Article/34653
73. ardalis. Overview of ASP.NET Core MVC [Internett]. [sitert 3. mai 2021]. Tilgjengelig på: https://docs.microsoft.com/en-us/aspnet/core/mvc/overview

74. Rick-Anderson. Introduction to authorization in ASP.NET Core [Internett]. [sitert 3. mai 2021]. Tilgjengelig på: https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction

75. Rick-Anderson. Role-based authorization in ASP.NET Core [Internett]. [sitert 3. mai 2021]. Tilgjengelig på: https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles

76. Newtonsoft Introduction [Internett]. [sitert 3. mai 2021]. Tilgjengelig på: https://www.newtonsoft.com/json/help/html/Introduction.htm

77. Xunit [Internett]. xUnit.net. [sitert 3. mai 2021]. Tilgjengelig på: https://xunit.net/

78. Kanjilal J. How to use Moq to ease unit testing in C# [Internett]. InfoWorld. 2018 [sitert 3. mai 2021]. Tilgjengelig på: https://www.infoworld.com/article/3264438/how-to-use-moq-to-ease-unit-testing-in-c.html

79. Rick-Anderson. Safe storage of app secrets in development in ASP.NET Core [Internett]. [sitert 12. mai 2021]. Tilgjengelig på: https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets

80. Getting Started with OpenAPI Tools | Swagger Open Source [Internett]. [sitert 13. mai 2021]. Tilgjengelig på: https://swagger.io/tools/open-source/getting-started/

81. MaterialUI [Internett]. Material Design. [sitert 13. mars 2021]. Tilgjengelig på: https://material.io/design/introduction

82. Overview | Formik [Internett]. [sitert 13. mars 2021]. Tilgjengelig på: https://formik.org/docs/[...slug]

83. Quense J. jquense/yup [Internett]. 2021 [sitert 13. mars 2021]. Tilgjengelig på: https://github.com/jquense/yup

84. AreaChart | Recharts [Internett]. [sitert 3. mai 2021]. Tilgjengelig på: https://recharts.org/en-US

85. axios/axios [Internett]. axios; 2021 [sitert 3. mai 2021]. Tilgjengelig på: https://github.com/axios/axios

86. Gackenheimer C. Introduction to React. Apress; 2015. 141 s.

87. React-Bootstrap [Internett]. [sitert 3. mai 2021]. Tilgjengelig på: https://react-bootstrap.github.io/

88. What is a React router? [Internett]. Educative: Interactive Courses for Software Developers. [sitert 3. mai 2021]. Tilgjengelig på: https://www.educative.io/edpresso/what-is-a-react-router

89. Store key-value data on disk [Internett]. [sitert 3. mai 2021]. Tilgjengelig på: https://flutter.dev/docs/cookbook/persistence/key-value

90. dart:convert library - Dart API [Internett]. [sitert 3. mai 2021]. Tilgjengelig på: https://api.dart.dev/stable/2.12.4/dart-convert/dart-convert-library.html

91. flutter_spinkit | Flutter Package [Internett]. Dart packages. [sitert 3. mai 2021]. Tilgjengelig på: https://pub.dev/packages/flutter_spinkit

92. dart:async library - Dart API [Internett]. [sitert 4. mai 2021]. Tilgjengelig på: https://api.dart.dev/stable/2.10.5/dart-async/dart-async-library.html

93. Client-side storage - Learn web development | MDN [Internett]. [sitert 14. mai 2021]. Tilgjengelig på: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage

94. Freeman A, Sanderson S. Pro ASP.NET MVC 3 framework. 3rd ed. New York : New York: Apress ; Distributed to the book trade worldwide by Springer Science+Business Media; 2011. 824 s. (The expert's voice in .NET).

95. dotnet-bot. IQueryable Interface (System.Linq) [Internett]. [sitert 18. mai 2021]. Tilgjengelig på: https://docs.microsoft.com/en-us/dotnet/api/system.linq.iqueryable

96. stevestein. Security Considerations (Entity Framework) - ADO.NET [Internett]. [sitert 4. mai 2021]. Tilgjengelig på: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/security-considerations

97. Breakpoints - Material-UI [Internett]. [sitert 7. mai 2021]. Tilgjengelig på: https://material-ui.com/customization/breakpoints/

98. gitignore.io - Create Useful .gitignore Files For Your Project [Internett]. [sitert 13. mai 2021]. Tilgjengelig på: https://www.toptal.com/developers/gitignore

# VEDLEGG

| | |
|---|---|
| Vedlegg 1 | Kildekode |
| Vedlegg 2 | Forprosjektrapport |
| Vedlegg 3 | UML diagram |
| Vedlegg 4 | Activity diagram |

| | |
|---|---|
| Vedlegg 5 | Mobil diagram |
| Vedlegg 6 | Use-case diagram |
| Vedlegg 7 | Wireframes for Bi-system |
| Vedlegg 8 | Kravspesifikasjon |
| Vedlegg 9 | Sprintrapporter |
| Vedlegg 10 | Arbeidslogg |
| Vedlegg 11 | Tilbakemelding fra arbeidsgiver |

Vedlagt under er også GitHub-link til de ulike delene av vårt prosjekt:

Githuborganisasjon:

- https://github.com/Database-Warehouse-Bachlor

Backend:

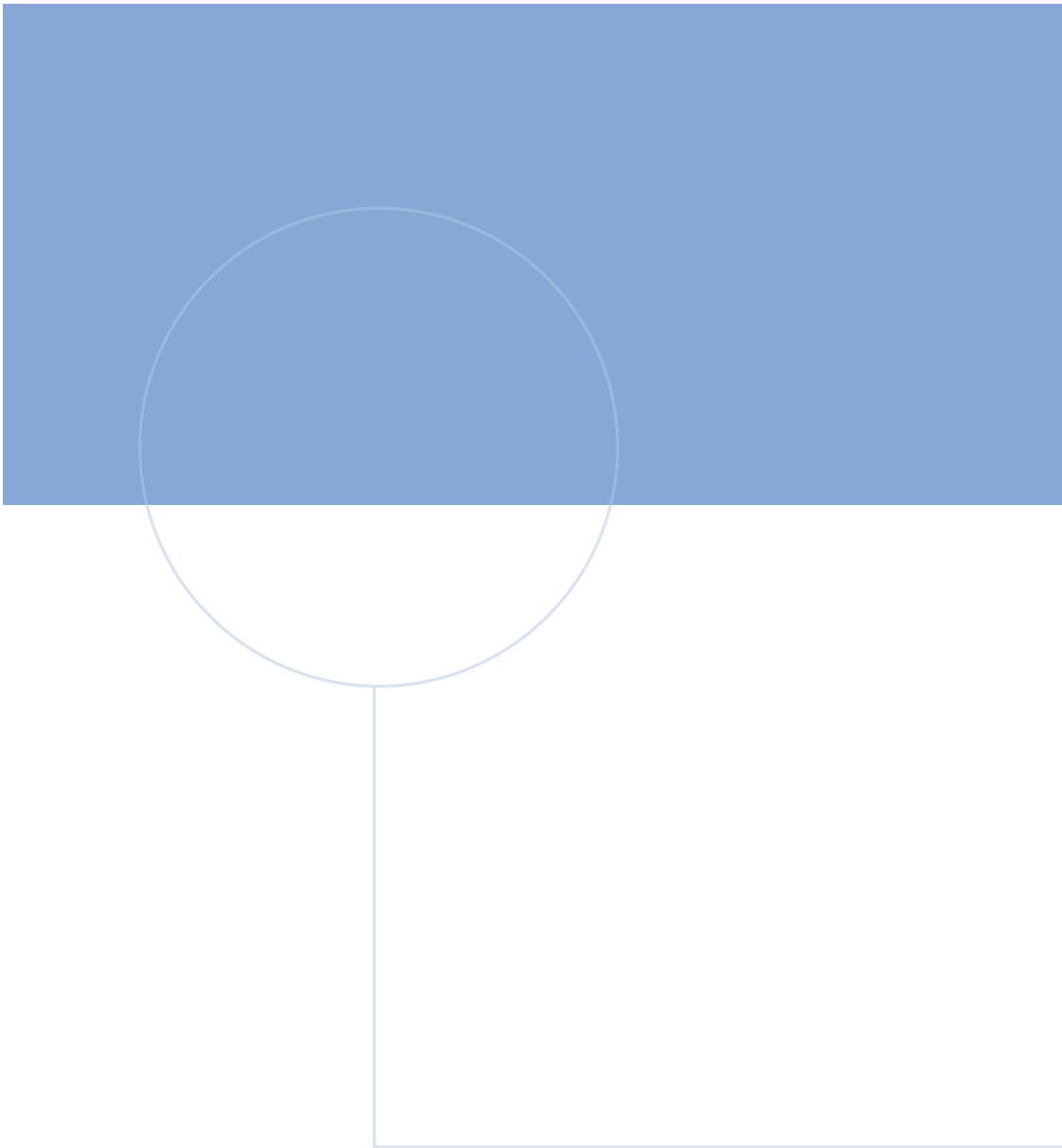
- https://github.com/Database-Warehouse-Bachlor/Datawarehouse-Backend

Bi-system:

- https://github.com/Database-Warehouse-Bachlor/Bi-System-frontend

Mobilapplikasjon:

- https://github.com/Database-Warehouse-Bachlor/MobilApp



NTNU

Kunnskap for en bedre verden