Høgskulen på Vestlandet

BACHELOROPPGAVE

Automatisert rapporteringstjeneste for bedriftsstatistikk

Automated report service for enterprise statistics

Henrik Hammer

Anders Tuft Buanes

Markus Bjermeland

Mikael Aare

DAT191
Fakultet for ingeniør- og naturvitenskap
Institutt for datateknologi, elektroteknologi og realfag
Dataingeniør

Veileder Richard Kjepso
23.05.22

Høgskulen på Vestlandet

Fakultet for ingeniør- og naturvitskap

Institutt for datateknologi, elektroteknologi og realfag

TITTELSIDE FOR HOVEDOPPGAVE

Rapportens tittel: Automatisert rapporteringstjeneste for bedriftsstatistikk Automated report service for enterprise statistics	*Dato:* 23.05.22
Forfatter(e): Henrik Hammer, Anders Tuft Buanes, Markus Bjermeland, Mikael Aare	*Antall sider u/vedlegg:* 55 *Antall sider vedlegg:* 136
Studieretning: Dataingeniør	*Antall disketter/CD-er:* 0
Kontaktperson ved studieretning: Richard Kjepso	*Gradering:* Ingen
Merknader: Ingen	

Oppdragsgiver: ZData AS	*Oppdragsgivers referanse:* Ingen
Oppdragsgivers kontaktperson: Kjetil Tollevsen	*Telefon:* 936 37 637

Sammendrag:

Prosjektet omhandler utvikling for ZData AS som spesialiserer seg i digitale finanstjenester. Løsningen skal fjerne manuelle rutiner for uthenting av statistikk og nøkkeltall. Dette vil bli gjort gjennom å utvikle et frittstående programmeringsgrensesnitt som skal tjene et brukergrensesnitt, samt kunne benyttes av andre tjenester i ZDatas system.

This project covers software development for ZData AS, a company specializing in digital financial services. The solution aims to remove manual routines for retrieving statistics and key performance indicators. This will be done by developing a stand-alone application programming interface that will serve a user interface, as well as complying with use by other services in ZDatas system.

Stikkord:

React – TypeScript	Programmeringsgrensesnitt	ASP.NET Core

Høgskulen på Vestlandet, Fakultet for ingeniør- og naturvitskap
Postadresse: Postboks 7030, 5020 BERGEN Besøksadresse: Inndalsveien 28, Bergen
Tlf. 55 58 75 00 Fax 55 58 77 90 E-post: post@hvl.no Hjemmeside: http://www.hvl.no

Høgskulen på Vestlandet

Fakultet for ingeniør- og naturvitskap

Institutt for datateknologi, elektroteknologi og realfag

Forord

Dette prosjektet har tilhørighet til studieprogrammet Dataingeniør ved Høgskulen på Vestlandet. Prosjekt ble gjennomført i vårsemesteret 2022 i perioden 17.01.22 - 23.05.22. Prosjektarbeid og dokumentasjon er utført av Henrik Hammer, Anders Tuft Buanes, Markus Bjermeland og Mikael Aare.

Oppgaven ble valgt da den virket spennende, og ga stor frihet i fremgangsmetode. Videre så gruppen nytteverdien for oppdragsgiver dersom løsningen av oppgaven var god.

Vi ønsker å takke oppdragsgiver ZData for en spennende og lærerik oppgave. Videre ønsker vi å takke vår kontaktperson Kjetil Tollevsen og seniorutvikler Jonas Sørsdal for god veiledning og et godt samarbeid i prosjektperioden. Avslutningsvis ønsker vi å takke veileder ved instituttet, Richard Kjepso, for god veiledning og konstruktiv kritikk til alt arbeid utført gjennom prosjektperioden.

INNHOLDSFORTEGNELSE

Innhold

9. VEDLEGG

Ordliste

Application Programming Interface (API)	Programmeringsgrensesnitt som tilgjengeliggjør funksjonalitet mellom komponenter i et system.
Assembly-språk	Assembly-språk er en samlebetegnelse for lavnivå språk som gir direkte instruksjoner til prosessoren, som videre kompileres til maskinkode.
Bundle	En komprimert samling kode.
CSR	Client-side rendering – En metode for å sende en JavaScript-bundle til klienten med instruksjoner for å endre HTML-dokumentet på klienten.
Ensideapplikasjon	En ensideapplikasjon er en nettside hvor alt innhold blir dynamisk oppdatert på siden, ovenfor å dirigere brukeren til en annen side.
HTTP	Hypertext Transfer Protocol – HTTP er den mest brukte protokollen for overføring av data over nett. Spesielt til bruk i nettsider.
Hypermedia	En videreutvikling av hypertekst, hvor koblingene kan omfatte tekst, lyd, grafikk, film og bilde.
Injeksjonsangrep	Et injeksjonsangrep er i kontekst av kode en samlebetegnelse på angrep hvor en ondsinnet aktør har mulighet til å kjøre ondsinnet kode i programvaren. Eksempelvis SQL-injeksjon, hvor en bruker kan passere autentisering uten å oppgi gyldige innloggingsdetaljer.
JSON	JavaScript Object Notation – En tekstbasert standard som tillater oversettelse mellom dataobjekter og tekst. Tross navnet er standarden uavhengig av JavaScript.
MVC	Model-View-Controller-arkitektur – Programmet er delt i model som holder data og view som viser data. Controller flytter data mellom model og view.
Open-source	Åpen kildekode – Alle har tilgang til, og mulighet til å kontribuere til kildekode.
REST	Representational Entity State Transfer – En metode for å utvikle APIer med visse organisatoriske prinsipper.
Skjelettinnlasting	Rammene av visningen til brukergrensesnittet er synlig mens data lastes inn
SSG	Static Site Generation – En metode for å lage alle HTML-dokumenter på server ved build-time og sende de til klienten etter behov.
SSR	Server-side Rendering – En metode for å lage HTML-dokumenter på serveren og deretter sende til klienten.
Supersett	I mengdelære er et supersett en betegnelse på en mengde som omslutter en annen mengde.
WebSocket	WebSocket er en kommunikasjonsprotokoll som gir full toveis kommunikasjon over en TCP-tilkobling. Protokollen er ofte brukt for tjenesten som krever levering av sanntidsdata.

1. INNLEDNING

1.1. Kontekst

ZData AS er et finansteknologiselskap som tilbyr en plattform for finansielle applikasjoner. Plattformen kobler regnskapssystemer og andre finansielle aktører sammen med banker, og tilbyr tjenester for effektiv betaling, automatisert bokføring og kontroll av regnskap (ZData AS, u.å.). De har opplevd stor vekst de siste årene, med blant annet dobling av antall ansatte mellom 2019 og 2020. Til tross for dette har flere interne rutiner forblitt manuelle, deriblant uthenting av faktureringsgrunnlag basert på tjenestebruk. Slike rutiner er overkommelig å opprettholde hos mindre bedrifter, imidlertid er de lite skalerbar da tidsbruk øker proporsjonalt med antall kundeforhold.

1.2. Motivasjon

Oppdragsgiver ønsker å fjerne alle eksisterende manuelle rutiner. I dag bruker bedriften unødvendige timer ved måneds- og årsslutt på å hente ut data manuelt for å fakturere sine kunder. Dette gjør vekst utfordrende, da økt antall kundeforhold vil følgelig øke timebruk for datauthenting.

Videre bruker oppdragsgiver også timer på å uthenting av data for å analysere trender. Dette resulterer i at oppdragsgiver har manglende innsikt i sine kunders bruksmønstre og lønnsomheten av egne produkter. Ved å automatisk hente, prosessere og fremvise relevante nøkkeltall kan oppdragsgiver oppnå økt tidsbesparelse i fakturering av sine kunder, samt økt innsikt i sine produkter.

1.3. Prosjekteier

Prosjekteier er ZData AS, et norsk finansteknologiselskap stiftet i 1990. Selskapet opererer hovedsakelig i Norge, men leverer også en plattform mot det nordiske bedriftsmarkedet. Tjenestene de tilbyr ligger i sjiktet mellom bank og finans. De utvikler løsninger for å forenkle

bedrifters økonomi ved å automatisere betalingsflyt, avstemming og løsninger som gir oversikt over regnskap (ZData AS, u.å.). Per 2020 har bedriften 36 ansatte, og en årlig salgsinntekt rundt 18 millioner kroner (Proff, u.å.).

1.4. Problembeskrivelse og mål

Det er lite praktisk for et selskap i vekst å opprettholde manuelle rutiner. I oppdragsgivers tilfelle må man be om data fra databasen med tilpassede spørringer. Slike spørringer krever en proporsjonal mengde tidsbruk målt mot antall kundeforhold. På grunn av dette ønsker oppdragsgiver en egen tjeneste som henter data i sanntid, lagrer dem og tilgjengeliggjør dem.

Hovedmålet for prosjektet er derfor å utvikle en applikasjon som leverer statistikk på tvers av produktene til oppdragsgiver. Det første delmålet er å utvikle et programmeringsgrensesnitt som aggregerer statistikk og kan benyttes som en tjeneste av andre produkter. Det andre delmålet er å utvikle et brukergrensesnitt som fremstiller og visualiserer statistikken fra dette programmeringsgrensesnittet.

Oppgaven gir stor frihet i hvordan den skal løses, men med målene som er satt ble følgende problemstilling formulert:

“Hvordan kan man utvikle et system som tilgjengeliggjør bedriftsstatistikk og bruksdata?”

1.5. Oppbygging av rapporten

Rapporten er delt i syv hovedkapitler. Kapittel 2 inneholder praktisk bakgrunn, avgrensninger for prosjektet, og en beskrivelse av ressursene benyttet i løsningen. I kapittel 3 vil alternative løsninger diskuteres, samt at valgt løsning vil drøftes. I dette kapittelet beskrives også verktøy og metodikk som skal brukes i prosjektet. Videre i kapittel 4 vil designet og utviklingen av løsningen beskrives i større detalj. Kapittel 5 tar for seg evaluering av prosjektet, resultat av utførte evalueringer, samt prosjektresultatet. I kapittel 6 vil resultater drøftes opp mot problemstillingen, delmålene og kravene for løsningen. Kapittel 7 konkluderer på prosjektets løp, innhold og resultat, samt forslag til videre arbeid. Avslutningsvis følger litteraturliste (kapittel 8) og vedlegg (kapittel 9).

2. PROSJEKTBESKRIVELSE

2.1. Praktisk bakgrunn

2.1.1. Initielle krav

Løsningen skal være todelt, hvor første del er et programmeringsgrensesnitt med oppkobling mot database og MVC-arkitektur. Det skal hentes bruksdata fra hendelser i oppdragsgivers skybaserte tjenester til en database og tilgjengeliggjøre aggregert statistikk (en samling av nøkkeltall over en tidsperiode). Programmeringsgrensesnittet skal være frittstående, veldokumentert og utformes i den hensikt at det kan benyttes av andre tjenester.

Videre skal løsningen inneholde et frittstående brukergrensesnitt som visualiserer samlet statistikk. Brukergrensesnittet skal visualisere statistikk levert av programmeringsgrensesnittet i respektive prosjektsider. Oppdragsgiver skal ha tilgang på all statistikk for alle klienter, samt at klienter skal ha tilgang på sin egen bruksstatistikk gjennom en brukerportal.

Det er satt krav til god testdekning og loggføring i løsningen. Videre er det satt krav om at all infrastruktur skal defineres med kode. Testdeknings-, loggførings- og infrastrukturkrav er grunnet i oppdragsgivers ønske om å videreutvikle løsningen etter prosjektslutt. Avslutningsvis skal løsningen kunne skaleres horisontalt (ved å instansiere flere vertsmaskiner for programvaren) etter behov. Skaleringen bør ha begrenset innvirkning på ytelse på klientsiden og ressurser på serversiden.

2.1.2. Initiell løsnings-idé

I de første møtene med oppdragsgiver ble det presentert en oppgave hvor det skulle utvikles en faktureringstjeneste. Denne tjenesten skulle beregne faktureringsgrunnlag basert på kundens tjenesteforbruk, og deretter generere faktura direkte i applikasjonen. Dette var utgangspunktet før prosjektstart. Etter oppstart ønsket oppdragsgiver en mer generell løsning som kunne brukes som en basistjeneste for andre systemer.

Prosjektet ble generalisert fra en fastsatt faktureringstjeneste til en tjeneste for levering av statistikk. Dataene som leveres av den nye tjenesten vil være den samme som skulle bli brukt i den opprinnelige faktureringstjenesten, men inneha mer generelt brukspotensiale. Dette gjorde det enkelt å endre oppgavens mål.

Som et resultat av dette skal det utvikles et programmeringsgrensesnitt som henter data fra en tjenestespesifikk database. Databasen vil motta aggregert data fra hendelser i oppdragsgivers skybaserte produkter. Dette grensesnittet skal tilby diverse endepunkter for levering av statistikk. Noen eksempler på statistikk som inkluderes er antall transaksjoner, betalinger og innbetalinger sendt gjennom produktene deres.

Videre skal det utvikles et brukergrensesnitt som mottar statistikk for produktene til oppdragsgiver fra endepunkter i programmeringsgrensesnittet. Brukergrensesnittet skal ha egne sider for hver av oppdragsgivers produkter, med mulighet for å filtrere på vilkårlige klienter.

2.2. Avgrensninger

Oppdragsgiver stilte krav til at løsningen ble utviklet i deres foretrukne serverrammeverk, og kunne vertes på deres eksisterende skytjeneste. Dette resulterte i at valg av løsning i all hovedsak omfattet klientsideteknologi.

Som nevnt ble omfanget av oppgaven endret tidlig i løpet. Ved oppstartsmøtet ble også andre avgrensninger for prosjektet satt. Dette innebar at prosjektgruppen ikke lenger var ansvarlig for aggregeringen av hendelser til prosjektets database, noe som definerer problemdomenet for oppgaven. Det var heller ikke lenger et krav til å utvikle en ekstern brukerportal, som nevnt i kapittel 2.1.1, men dette var ønskelig dersom tidsrammen tillot det.

2.3. Ressurser

Samtlige av oppdragsgivers ressurser er tilgjengelig for prosjektet. Blant annet er bruks- og salgsdata fra deres tjenester essensielle ressurser for løsningen.

Videre skal prosjektet benytte seg av oppdragsgivers foretrukne utviklingsplattform, som samler versjonskontroll, prosjektstyring og kravhåndtering i en løsning. Prosjektet skal også nytte eksisterende autentiserings- og autorisasjonstjenester, samt at ytelsesmåling og loggføring skal utføres med deres foretrukne tjenester. Det er også stilt kontorplasser til rådighet ved oppdragsgivers hovedkontor.

Avslutningsvis vil oppdragsgiver ha en sentral rolle som brukertester og ekstern veileder.

3. DESIGN AV PROSJEKTET

3.1. Forslag til løsning

Som nevnt ble valg av serverrammeverk avgrenset av oppdragsgiver. Deres foretrukne rammeverk er ASP.NET Core. Programmeringsgrensesnittet skal benytte seg av HTTP-protokollen for levering av data ved å følge REST-prinsippene. Videre skal den benytte seg av en database for uthenting av data. Det mest sentrale valget som gjenstår er da valg av klientsideteknologi.

3.1.1. Alternativ 1: CSR

En løsning vil være å bruke CSR. Med CSR leveres et HTML-dokument og en JavaScript-bundle til klienten. Her er JavaScript-bundlen ansvarlig for å oppdatere dokumentet til korrekt tilstand. Alle beregninger og funksjonskall utføres lokalt på klientens maskin. CSR egner seg for interaktive applikasjoner som ikke utfører tunge beregninger. Det egner seg også for applikasjoner som skal skaleres, uten støttende infrastruktur.

3.1.1.1. React

En mulig kandidat for CSR-basert løsning er React. React er et JavaScript-bibliotek for utvikling av brukergrensesnitt for webapplikasjoner. I følge StackOverflow sine årlige spørreundersøkelser er React det mest populære rammeverket for å bygge nettsider (Stack Overflow, 2021). Oppdragsgiver bruker også dette rammeverket i flere av deres eksisterende tjenester.

Å bruke React kan gi en effektiv utviklingssyklus grunnet NPM-pakker. NPM gir utviklere tilgang til tusenvis av ferdigskrevne JavaScript-moduler (NPM, u.å.). Grunnet Reacts popularitet finnes det veldig mange ressurser for React, som dekker de fleste behov.

3.1.1.2. Blazor WASM

En annen kandidat for en CSR-basert løsning er Blazor WASM. Blazor WASM er et rammeverk for utvikling av ensideapplikasjoner. Denne løsningen gir hurtig lasting etter bundlen er lastet ned. Hastigheten er grunnet bruken av WebAssembly [WASM], som er et høynivå assembly-språk. WASM muliggjør kjøring av maskinkode i nettleseren, som gir økt tilgang og kontroll over

prosesseringskraften på vertsheten, sammenlignet med et tolket språk som JavaScript. Blazor WASM er en relativt ny teknologi, og kan medføre risiko av den grunn.

Å bruke Blazor WASM gir en applikasjon som kan bruke alle funksjoner i .NET og C#. En stor fordel med denne løsningen er at klient- og serverside vil bruke det samme språket, dermed vil kompleksiteten i kodebasen reduseres.

3.1.2. Alternativ 2: SSR

En annen løsning vil være å bruke SSR. Med SSR genereres hele HTML-dokumentet på serveren før den blir sendt i respons til klienten. Denne metoden unngår kall for datahenting og generering av mal på klientsiden, ettersom alt er blitt håndtert på serveren. Dette gjør SSR passende til større applikasjoner med tunge beregninger, som ikke skal skaleres i høy grad.

3.1.2.1. Next.js

En kandidat for SSR-basert løsning er rammeverket Next.js [Next]. Next bygger på Reacts funksjonalitet og tillater SSR og SSG. Dette vil tilby raskere lasting uavhengig av fastvaren på sluttbrukerens enhet.

Next gir i all hovedsak de samme fordelene som React. NPM-pakker vil stå sentral i utviklingen av en applikasjon skrevet med Next. Etter første utgivelse i oktober 2016 har Next sett rask økning i popularitet med omtrent 700,000 brukere på GitHub (GitHub, u.å.), samt bruk av store tjenester som Twitch og Hulu (Next.js, u.å.). Denne populariteten har ført til et mangfold av tilgjengelige ressurser for rammeverket, noe som kan gjøre utviklingsprosessen effektiv.

3.1.2.2. Blazor Server

En annen kandidat for SSR-basert løsning er Blazor Server. Blazor Server er SSR-varianten av Blazor, hvor all kommunikasjon mellom klient og tjener gjøres med SignalR (en implementasjon av WebSocket-protokollen). Dette gir rask lasting av sider, samt en uforstyrret opplevelse for sluttbrukeren.

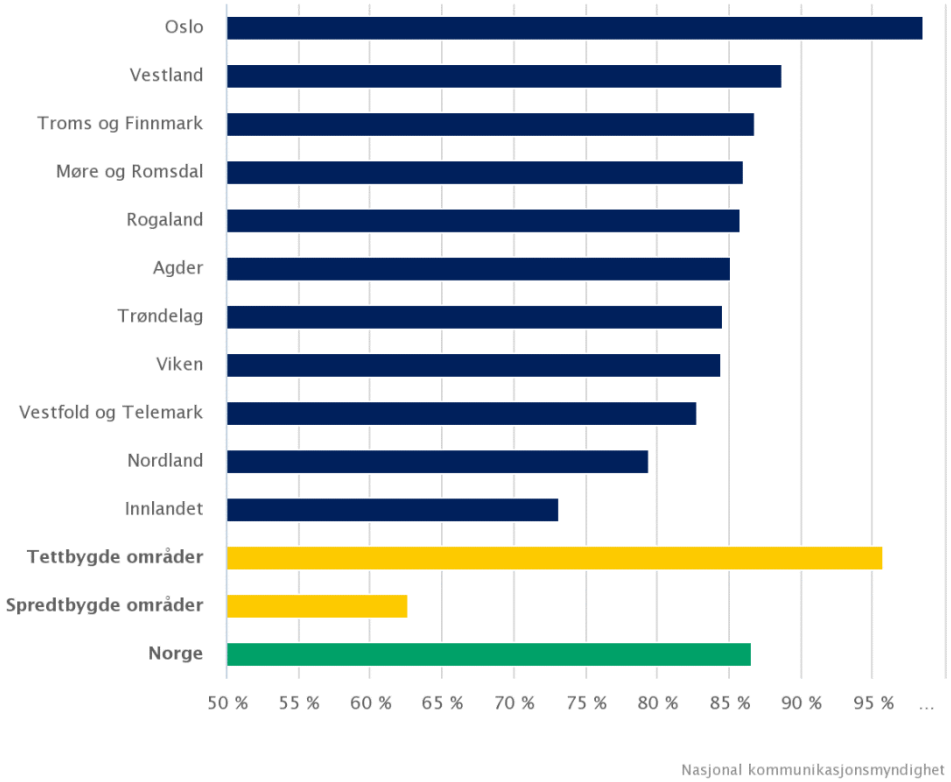
Blazor Server vil i all hovedsak gi de samme fordelene som Blazor WASM, men beregninger utføres på serveren ovenfor på klientsiden. Den største fordelen er igjen, å ha det samme språket på klientside og serverside.

3.1.3. Diskusjon av alternativene

De avgjørende faktorene for valg av løsning er: skaleringsevne, nedlastningstid, applikasjonens tyngde, ytelse og hvilke bruksenheter applikasjonen skal brukes på.

Oppdragsgiver ønsker at applikasjonen skal kunne skaleres uten betydelig økning i ressursbruk på serversiden. Dette er kun mulig med CSR ettersom nødvendige serverressurser vil øke proporsjonelt med antall brukere i en SSR-løsning.

En annen faktor er nedlastingstid. Her er SSR optimalt, ettersom man bare sender det aktuelle dokumentet for en gitt side. CSR krever at hele *bundlen* lastes ned ved innlasting av siden, dette vil følgelig gi høyere nedlastingstid enn en SSR-basert løsning. Ettersom den gjennomsnittlige nedstrømshastigheten har økt i de siste årene, kan forskjellen i nedlastningstid være neglisjerbar. Figur 1 viser dekningsprosentene for virksomheter som har tilbud om en nedstrøms hastighet på 100 Mbit/s eller mer (Nasjonal kommunikasjonsmyndighet [Nkom], 2021). Denne statistikken kan antas å være representativ for oppdragsgivers nettverk. Såfremt en levert applikasjon er liten nok til å nedlastes på under ett sekund burde dette være tilstrekkelig.



Figur 1: Dekning for bredbånd med nedstrøms hastighet på 100 Mbit/s for virksomheter i Norge (Nkom, 2021)

Dersom applikasjonen blir kompleks og tung er SSR den klart optimale løsningen både i forhold til nedlastningstid, innlastingstid og ytelse i applikasjonen. I henhold til den initielle løsningsideen skal ikke den ferdigstilte løsningen være så tung at dette vil påvirke nedlastnings- og innlastningstid. Dette gjør at CSR antakeligvis er det optimale valget for den planlagte løsningen. Bruken av CSR vil også kunne gi høyere ytelse når applikasjonen er lastet ned, gitt at bruksenheten til sluttbrukeren er kraftig nok for å kjøre applikasjonen. Dette kan sees som neglisjerbart problem, ettersom applikasjonens brukere sannsynligvis vil besøke siden fra kapable, bedriftsorienterte datamaskiner.

For hver foreslåtte teknologi er det viktig å undersøke hvordan de påvirker disse faktorene. Samt hvordan teknologien påvirker løsningens kompleksitet.

Som CSR-baserte løsninger vil både React og Blazor WASM kunne skalere godt uten økt bruk av serverressurser. Videre stiller løsningene ulikt i applikasjonens tyngde. En gjennomsnittlig JavaScript-bundle vil være på en størrelse mellom 500kB og 2MB, en lignende Blazor WASM bundle vil være omtrent 4 ganger større. Dette gjør Blazor WASM betydelig tregere å laste inn enn en lignende JavaScript-løsning.

Videre er ytelse et sentralt moment. I en sammenligning av ytelse for frontend-rammeverk, illustrert i figur 2, fremstår React som betydelig raskere i utførelsen av operasjoner med liten til medium kompleksitet, mens WASM fremstår som raskere for operasjoner med høy kompleksitet. Samlet over alle referansepunktene (*geometric mean*) fremstår React som omtrent dobbelt så hurtig som Blazor WASM.

Duration in milliseconds ± 95% confidence interval (Slowdown = Duration / Fastest)

Name Duration for...	create rows creating 1,000 rows	replace all rows updating all 1,000 rows (5 warmup runs).	partial update updating every 10th row for 1,000 rows (3 warmup runs). 16x CPU slowdown.	select row highlighting a selected row. (no warmup runs). 16x CPU slowdown.	swap rows swap 2 rows for table with 1,000 rows. (5 warmup runs). 4x CPU slowdown.	remove row removing one row. (5 warmup runs).	create many rows creating 10,000 rows	append rows to large table appending 1,000 to a table of 10,000 rows. 2x CPU slowdown.	clear rows clearing a table with 1,000 rows. 8x CPU slowdown.	geometric mean of all factors in the table
react-zustand-v17.0.1 + 3.6.8	124.2 ± 7.2 (1.00)	96.7 ± 0.6 (1.00)	199.0 ± 3.5 (1.00)	49.6 ± 1.4 (1.00)	324.9 ± 12.0 (1.92)	25.5 ± 1.1 (1.00)	1,335.7 ± 19.5 (1.00)	251.9 ± 8.8 (1.00)	79.4 ± 1.8 (1.00)	1.08
blazor-wasm-v6.0.1	256.9 ± 1.6 (2.07)	212.3 ± 3.3 (2.20)	638.5 ± 2.7 (3.21)	527.8 ± 1.8 (10.65)	169.0 ± 0.9 (1.00)	54.1 ± 0.3 (2.12)	2,242.6 ± 34.2 (1.68)	497.6 ± 2.8 (1.98)	144.0 ± 2.3 (1.82)	2.32

Figur 2: Sammenligning av ytelse mellom React og Blazor-WASM. *Transponert retning fra originaltabell.* (Krausest, u.å)

Det som da skiller disse løsningene er kompleksitet og utviklingshastighet. Med React vil utviklingshastigheten bli høyere, men den samlede løsningen vil ha høyere kompleksitet. Med Blazor vil kompleksiteten i koden forbli lav, men applikasjonens størrelse blir betydelig større, samt at utviklingshastigheten antakeligvis blir senket.

Som SSR-baserte løsninger vil både Next og Blazor Server ha utfordringer innen skalering. Dette problemet kan optimaliseres, men er ikke et fokus for dette prosjektet. Videre stiller løsningene likt i nedlastningstid og tyngde. Ytelsen fra disse løsningene er også nokså like, og vil som nevnt kunne utklasse CSR-løsningene i tyngre arbeid, på bekostning av serverressurser. Det som da skiller disse løsningene er igjen kompleksitet og utviklingshastighet. Next gir høyere utviklingshastighet og Blazor Server gir mindre kompleksitet i kodebasen.

3.2. Valgt løsning

Før et teknologivalg kan gjøres må en overordnet leveransemetode for applikasjonen etableres. Med krav om skaleringsevne, forventet størrelse og ønsket ytelse, blir det naturlig å velge en CSR-basert løsning.

Da den planlagte applikasjonen ikke skal utføre kompleks logikk, og hovedsakelig skal visualisere data, vil det være mest naturlig å velge React. React vil, grunnet prosjektets natur, gi den best

ytende applikasjonen med akseptabel nedlastingstid. React vil også tillatte bruken av NPM-pakker for datavisualisering, som øker utviklingshastigheten betydelig. Dette valget vil øke kompleksiteten i kodebasen, men denne økningen er neglisjerbar grunnet Reacts dokumentasjon og popularitet. Som nevnt er React brukt i flere av oppdragsgivers tjenester, noe som kan forenkle videreutvikling av den endelige løsningen.

3.3. Valg av verktøy og språk

3.3.1. Språk og rammeverk

C#

C# er et objektorientert språk med statisk typesjekking utviklet av Microsoft. Språket er basert på C++ og Java med funksjoner som automatisk minnehåndtering, nullbare typer og unntakshåndtering. Språket innarbeider elementer fra imperativ og funksjonell programmering og er et vanlig språk å bruke i rammeverket .NET.

ASP.NET

ASP.NET er et open-source rammeverk for utvikling av webapplikasjoner. ASP.NET bygger på det mer generelle .NET-rammeverket. ASP.NET muliggjør bygging av webapplikasjoner ved hjelp av C#.

TypeScript

TypeScript er et supersett av JavaScript som legger på statisk typesjekking av variabler. Dette gjør det enklere å fange opp feil i utviklingsprosessen. Det muliggjør også bruken av grensesnitt, eksempelvis for innkommende objekter. Dette gir bedre utviklerverktøy, som leder til økt produktivitet og forutsigbarhet i utviklingsløpet.

React

React er et komponentbasert bibliotek, som fokuserer på å skape små komponenter som kan komponeres til større komponenter. Dette resulterer i høy gjenbrukbarhet. Alle komponenter

bygges opp med JSX. JSX er en krysning mellom HTML og JavaScript, som gir muligheter for å bygge maler med kompleks logikk og tilstand.

3.3.2. Verktøy

Visual Studio 2022

Visual Studio 2022 er et utviklingsmiljø utviklet av Microsoft rettet mot programvareutvikling i Windows, spesielt .NET og C#. Prosjektet bruker Visual Studio grunnet den tette integrasjonen mot .NET og utviklerverktøyene som medfølger.

Git

Git er et distribuert versjonskontrollsystem for kildekode, som tillater samarbeid mellom flere utviklere på samme kodebase. Git tilrettelegger for parallelt arbeid. Alle utviklerne kan skille ut sin egen gren av prosjektet, utføre endringer og senere slå sammen sin gren med hovedgrenen.

Terraform

Terraform er et verktøy for bygging, endring og versjonskontroll av infrastruktur. Terraform definerer infrastruktur ved hjelp av deklarativ kode. Bruken av Terraform skyldes oppdragsgivers krav om at all infrastruktur skal være definert i kodeformat.

3.3.3. Plattform

Microsoft Azure

Azure er en skyplattform skapt av Microsoft som inneholder alt man trenger for å utvikle programvare. Oppdragsgiver bruker allerede Azure, som gjør det til et naturlig startsted for prosjektets utviklingsprosess. All kode relatert til prosjektet lagres i Azures Git-tjener.

Det har også blitt benyttet Azure Boards for å jobbe med Scrum og holde følge med sprintene i utviklingen. Dette blir beskrevet i større detalj i delkapittel 3.4.1.

3.3.4. Samarbeidsverktøy

Office Online

Office Online tillater åpent samarbeid i Word og Excel med versjonskontroll. Dette blir benyttet som verktøy for dokumentasjon.

Discord

Discord er en kommunikasjonsplattform for tekst og lyd rettet mot kommunikasjon mellom privatpersoner. Den er markedsført og beregnet for dataspill-relatert bruk, men tilbyr likevel gode verktøy for organisering og planlegging. Discord blir brukt som kommunikasjonskanal innad i gruppen til både formelle og uformelle formål.

Slack

Slack er en kommunikasjonsplattform for tekst og lyd rettet mot kommunikasjon innad i bedrifter. Den blir brukt som kommunikasjonskanal med oppdragsgiver til trivielle spørsmål og mindre viktige avklaringer.

Kontorlokale

Oppdragsgiver har stilt et kontorlokale til rådighet for gruppens bruk. Her vil oppdragsgiver og seniorutviklere være tilgjengelig dersom det skulle være behov for større og mer detaljerte avklaringer.

Microsoft Teams

Microsoft Teams er en kommunikasjonsplattform med tekst, video og lyd som blir benyttet for møter med oppdragsgiver.

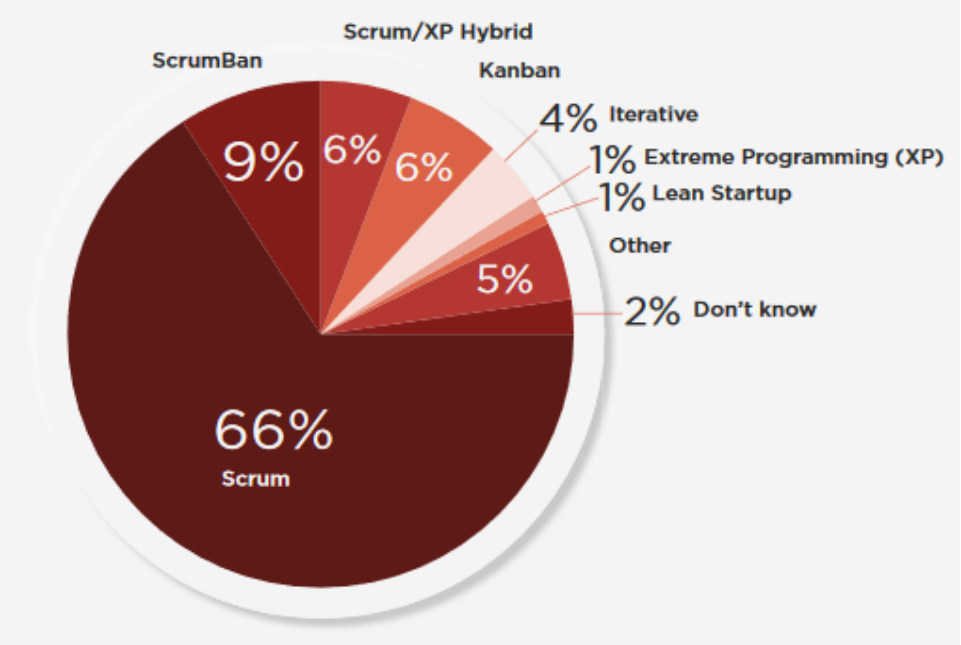
Zoom Meetings

Zoom Meetings er en kommunikasjonsplattform med tekst, video og lyd som blir benyttet for møter med veileder.

3.4. Prosjektmetodikk

3.4.1. Utviklingsmetodikk

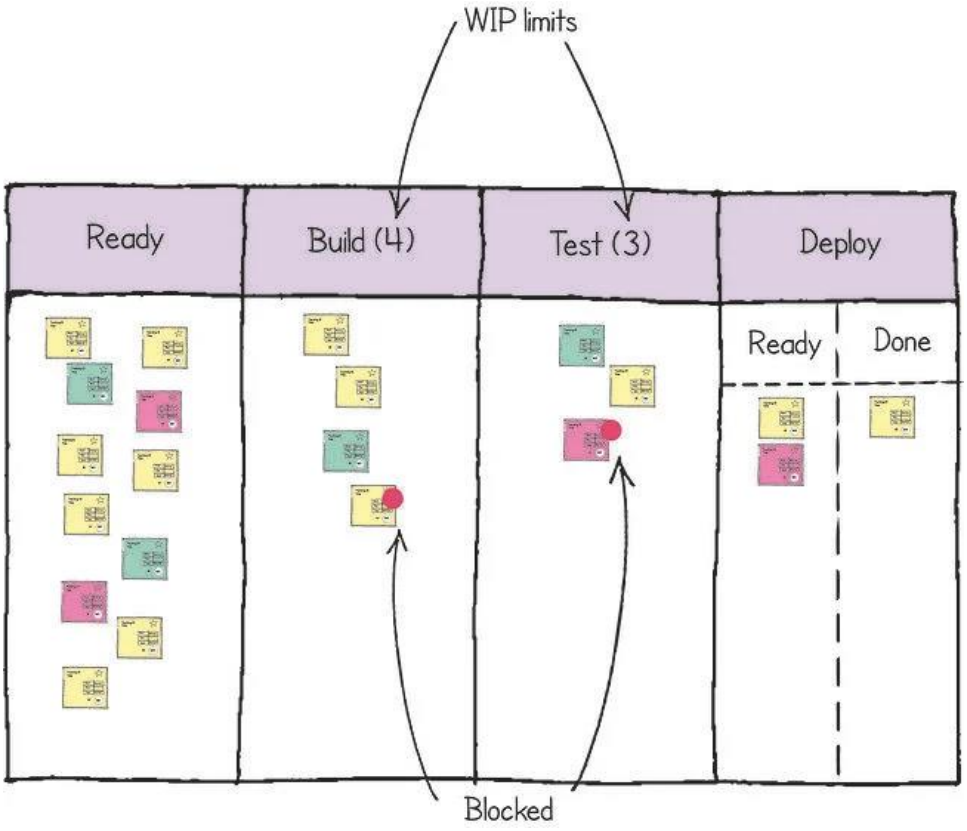
Under oppstartsfasen ble det bestemt sammen med oppdragsgiver at det skulle benyttes en smidig (agile) utviklingsprosess. For de fleste smidige utviklingsprosesser er iterativt, inkrementelt og progressivt arbeid sentralt. Det var spesielt tre alternativer som sto frem, da disse holder høyest markedsandel i arbeidslivet, som illustrert i figur 3. Disse metodene var Kanban, Scrum og Scrumban.



Figur 3: Markedsandel for smidige utviklingsmetoder (Digital.ai, 2021)

Kanban er en metode som benytter et sett av prinsipper og praksiser for å administrere og optimalisere løpende arbeid. Metoden ble utviklet av Toyota på 1940-tallet for å effektivisere produksjonslinjen på fabrikkene (Høgstrand, 2019). Prinsippene bygger på at Kanban er en metode som anvendes i påbegynt arbeidsflyt og ikke pålegger noen organisatoriske endringer. Med andre ord skal Kanban forbedre en prosess som allerede er igangsatt uten å forstyrre strukturen i organisasjonen. På en annen side fremmer Kanban lederskap og idémyldring i alle nivåer av organisasjonen. Dette vil si at en persons stilling ikke skal begrense påvirkningsmulighet i prosessen.

Kanban er også en visuell metode. Sentralt i metoden er Kanban-tavlen, som illustrert i figur 4. Målet med tavlen er å visualisere arbeidsflyten ved å kategorisere arbeidsoppgaver i kolonner. Typisk vil kategoriene være *oppgaver som er klare*, *pågående oppgaver*, *oppgaver til vurdering* og *fullførte oppgaver*. I hver kolonne vil det kunne avgrenses hvor mange oppgaver som kan foretas (dette kalles *work-in-progress limits*).

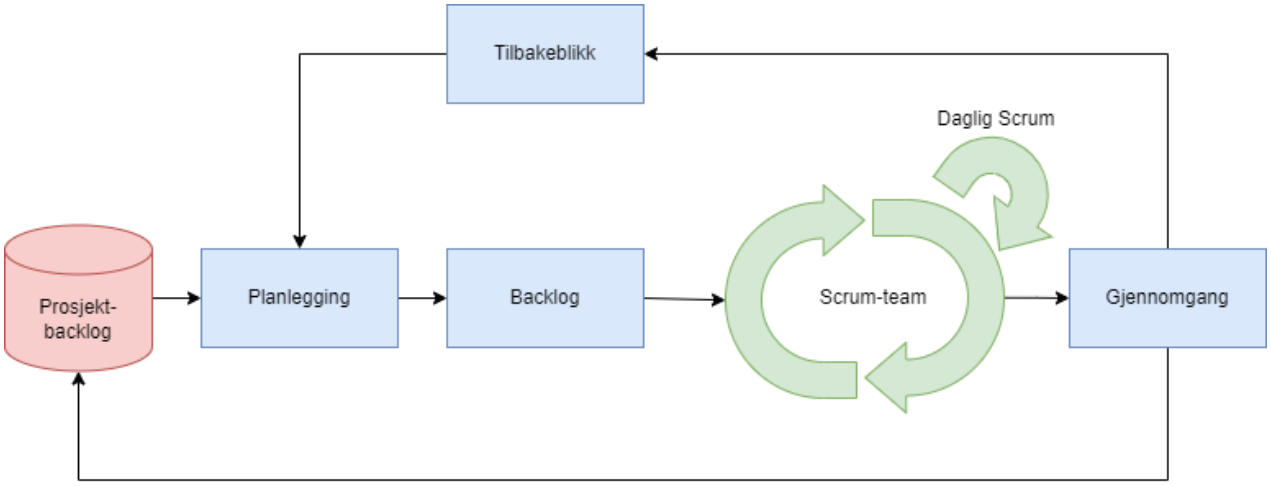


Figur 4: Eksempel på Kanban-tavle, med avgrensinger, for en programvareutviklingsprosess (Høgstrand, 2019)

Scrum er et smidig utviklingsrammeverk som definerer roller og prinsipper ovenfor teknikker. I følge Schwaber og Sutherland (2020) er hensikten til rammeverket å generere verdi gjennom tilpassede løsninger på komplekse problemer. Denne verdiøkning vil oppnås gjennom inkrementelt arbeid over korte intervaller, definert som sprinter.

En sprint, illustrert i figur 5, er én av flere *hendelser* i Scrum-rammeverket. Det som skiller en sprint fra andre hendelser er at den omfavner de andre hendelsene. Hendelsene som opptrer i løpet av en sprint er planlegging, daglige møter, gjennomgang, og tilbakeblikk. Ved planlegging vil

oppgaver fra prosjektkøen (*backlog*) bli lagt til den gjeldende sprintkøen. Dersom oppgaver i sprintkøen ikke blir gjennomført i løpet av sprinten, vil de bli lagt til i prosjektkøen.



Figur 5: Illustrasjon av flyten i en Sprint-hendelse i Scrum

Scrum-sprinten skal vare over en fast periode, vanligvis mellom én til fire uker. Denne perioden starter med planlegging der det settes et overordnet mål, og arbeidsoppgaver defineres ut fra det fastsatte målet. Gjennom perioden skal det avholdes daglige møter for å følge med på fremgangen mot målet, og gjøres endringer dersom det sees nødvendig. Ved slutten av perioden skal resultatet presenteres for produkteier, og avslutningsvis skal utviklingsteamet se tilbake på utfordringer og muligheter som oppstod i sprinten.

Scrumban er en utviklingsmetodikk som er hybrid av Kanban og Scrum. Scrumban ble opprinnelig utviklet som en metode for å konvertere fra Scrum til Kanban. Metoden baserer seg blant annet på det visuelle elementet i Kanban. Scrumban-tavlen følger de samme kategoriene som Kanban, men uten avgrensinger på arbeidsoppgaver under hver kategori. Alle elementer på tavlen blir gitt en prioritering, og oppgaver løses basert på prioritet. Dette er for å sikre kontinuerlig arbeid, ettersom man ikke jobber etter tidsbegrensninger som i Scrum. Et viktig likhetstrekk med Scrum er utførelsen av daglige møter. Ulikt Scrum, vil ikke disse møtene styres av en sentral person, men heller la utviklingsteamet diskutere dagens utfordringer.

Scrumban kan gi god fleksibilitet ved at man ikke er fastlåst til Scrums roller, samt tidsbesparelse ved å ikke avholde sprintmøter. Samtidig har utviklingsteamet god oversikt over arbeidsflyt og fremgang i Scrumban-tavlen.

3.4.2. Diskusjon av utviklingsmetodikk

Oppdragsgiver benytter seg av Kanban, noe som gjorde den aktuell i oppstartsfasen av prosjektet. Det er derimot en metode som egner seg for større prosjekter med større grupper, samt prosjekter med langsiktige mål og antatt lang varighet.

Scrum egner seg godt for prosjektarbeid. Scrum setter trygge rammer rundt tidsbruk og medlemmers rolle i prosjektet. For det første oppnås tett samarbeid med oppdragsgiver gjennom fastsatte møter. For det andre er det tydelige framgangsindikatorer i form av en sprintperiode eller resterende sprinter i prosjektet. Avslutningsvis hjelper metoden å igangsette arbeidsflyt ved tilby konkrete retningslinjer for prosess.

Scrumban egner seg godt når det er ønskelig med fleksibilitet i et prosjekt. Dette er ettersom utviklingsteamet selv bestemmer fordelingen av arbeidsoppgaver. Spesielt egner Scrumban seg for løpende arbeid, som vedlikehold av pågående prosjekter, ettersom den vektlegger prioritering av oppgaver og ikke er begrenset av sprinter.

3.4.3. Valg av utviklingsmetodikk

Valget av metode ble avgjort i oppstartsfasen av prosjektet. I denne fasen var det behov for å planlegge utviklingsarbeidet, og knytte dette opp mot oppdragsgivers visjon. Kanban og Scrumban egner seg som sagt til påbegynte arbeidsflyt med større grupper. Dermed ble det aktuelt å benytte Scrum. Scrum er tilpasset mindre grupper og en fastsatt prosjektvarighet. Ettersom gruppen består av fire medlemmer og prosjektet har en fastsatt varighet, ble det hensiktsmessig å benytte en modifisert versjon av Scrum med produkteier og utviklingsteam.

I Scrum-rammeverket er produkteier ansvarlig for å stille krav som kan omformes til brukerhistorier for en gitt sprint. Scrum-masteren står ansvarlig for å delegere oppgaver og motivere gruppen. Utviklingsteamet står ansvarlig for å løse delproblemene som er definert innad i en sprint (Scrum.org, u.å.).

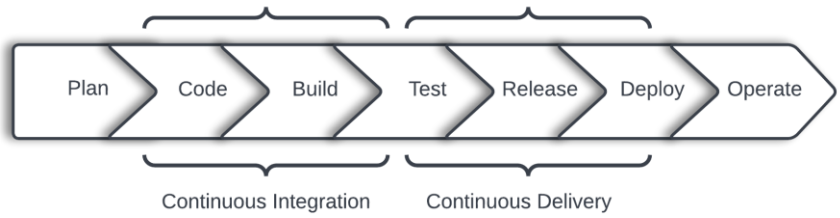
I prosjektet ble det bestemt at oppdragsgiver fyller rollen av produkteier og gruppen fyller rollen av utviklingsteam. Grunnet gruppens størrelse ble det lite hensiktsmessig å ha en dedikert Scrum-master. Videre ble det bestemt at gruppen jobber i sprinter på to uker. For å imøtekomme oppdragsgiver ble møter for sprintgjennomgang og følgelig sprintplanlegging kombinert.

Scrum gir muligheten for å være fleksibel i møte med utviklingen av prosjektet og håndtere hendelser eller problemer fortløpende. Videre resulterer det i et fungerende system ved en tidlig fase av utviklingen. Dette betyr at funksjonalitet som blir utviklet ved kommende sprinter kan testes og integreres som fullt fungerende komponenter av systemet.

3.4.4. CI/CD

Kontinuerlig integrasjon (CI) er et sett av metoder og prinsipper som blir utført når ny kode skal integreres i kodebasen. Eksempelvis utføring av enhetstester for kontrollere ny kode for feil. Det er som nevnt et krav at prosjektet skal ha god testdekning ved bruk av enhetstester og integrasjonstester.

Kontinuerlig levering (CD) håndterer koden etter at den er integrert i kodebasen, og behandler utleveringen av koden til ulike miljøer som demo- og produksjonsmiljø. Som nevnt tidligere benyttes skyplattformen Azure som tilbyr muligheten for å sette opp automatiske CI/CD-prosesser i Azure Pipelines. Dette kan gi konsekvente resultater i gjennomførelsen av prinsippene. Ved hvert sprintmøte blir det diskutert og avgjort om arbeidet gjort i sprinten skal bli sendt ut til produksjon, noe som følger prinsippet om kontinuerlig levering. Sammenhengen mellom CI og CD kan sees i figur 6.



Figur 6: Illustrasjon av CI/CD-prosess

3.4.5. Prosjektplan

Det ble benyttet et Gantt-diagram (se vedlegg A) for å utforme framdriftsplanen for prosjektet. Diagrammet er inndelt i tre ulike hovedkategorier. Denne inndeling er ment til å øke oversiktligheten av fremgangen i de forskjellige samlingene av oppgaver.

Et alternativ til kategorisk inndeling er å dele aktivitetene inn i faser, som vil resultere i en bedre oversikt av total framgang. Imidlertid er det flere aktiviteter, i forskjellige kategorier, som samkjører over hele prosjektperioden.

Dokumentasjon omhandler prosjektrapporten og alle dens støttedokumenter. Her er det viktig at alle dokumenter tilhørende forprosjektfasen er planlagt å være ferdigstilt til oppstart av hovedrapporten.

Under *Utvikling* er aktivitetene representert av sprintene som følger av utviklingsmetodikken. Hver sprint omfatter, som nevnt, et to-ukers intervall i utviklingsløpet med tilbakeblikk på forrige sprint og planlegging av den gjeldende sprinten, fulgt av utvikling og avslutningsvis brukertesting.

Punkter under *Diverse* er de aktivitetene som ikke har et direkte forhold til rapporten eller utviklingen av produktet, men som fortsatt er sentrale for gjennomførelsen av prosjektet.

3.4.6. Risikovurdering

I risikoanalysen, illustrert av figur 7, ble alvorlighetsgraden av risikoer vurdert ut fra sannsynligheten for at noe skulle inntreffe, og i hvor stor grad konsekvensen ville påvirke prosjektet. Både sannsynlighet og konsekvens av en risiko rangeres mellom 1 og 5, hvor 1 er lavest og 5 er høyest. Ved å multiplisere sannsynligheten og konsekvensen for en gitt hendelse får man risikofaktoren for hendelsen.

	Hendelse /Risiko	Årsak	Sannsynlighet	Konsekvens	Risiko - produkt	Tiltak
1	Sykdom i gruppen	Ikke god nok oppfølging av smittevern / uheldig.	Middels (3)	Svært Lav (1)	3	Hjemmekontor
2	Ukjente programmeringsverktøy	For lite kjennskap til oppdragsgivers verktøy og teknologistack.	Lav (2)	Middels (3)	6	Sikre god tid for læring av ukjente verktøy.
3	Dataangrep	Dårlige rutiner og tester på sikkerhetshåndtering.	Lav (2)	Svært Høy (5)	10	Sikre å følge gode rutiner for sikring av kode. Følge OWASPs råd for cybersikkerhet.
4	Urealistisk tidsperspektiv til å realisere applikasjon	Oppdragsgiver har for kompleks oppgave med mange funksjoner.	Middels (3)	Middels (3)	9	Oppdragsgiver og gruppen må komme til enighet om en oppgave som lar seg gjennomføre innenfor tidsperioden.
5	Applikasjonen er ustabil i kjøring	For lite testing. Dårlig DevOps-metodikk under utvikling.	Lav (2)	Høy (4)	8	Sikre høy testdekning av prosjekt. Samt å skape god struktur for sikring av kvalitet før integrasjon i kodebase.
6	Utvikler feil produkt	Oppgaven er ikke spesifisert godt nok.	Middels (3)	Svært Høy (5)	15	Oppgavene må bli spesifisert nøyaktig slik at begge parter er innforstått med hva som skal utvikles
7	Oppdragsgiver er ikke fornøyd med realisert oppgave	For dårlig spesifisering av oppgave.	Lav (2)	Svært Høy (5)	10	Oppgavene må bli spesifisert nøyaktig slik at begge parter er innforstått med hva som skal utvikles

Figur 7: Risikoanalyse for prosjektet (fra vedlegg D)

Videre ble de identifiserte risikoene satt inn i en risikomatrise (se figur 8) for å få et overblikk av den totale risikofaktoren for prosjektet. Først og fremst er det tydelig at det er få alvorlige risikoer i prosjektet, derimot er det flere risikoer av betydelig konsekvens.

Sannsynlighet						
	Svært Høy (5)					
	Høy (4)					
	Middels (3)	1		4		6
	Lav (2)			2	5	3,7
	Svært Lav (1)					
		Svært Lav (1)	Lav (2)	Middels (3)	Høy (4)	Svært Høy (5)
	Konsekvens					

Figur 8: Risikomatrise (fra vedlegg D)

I risikoanalysen er det foreslått tiltak for å minimere både sannsynlighet og konsekvens av risikoene i prosjektet. Det vil bli gjort kontinuerlig evaluering av aktuelle risikoer, samt vurdering av nye risikoer.

3.5. Evalueringsplan

3.5.1. Systemtesting

God testdekning er påkrevd fra oppdragsgivers side. Alle sentrale funksjoner skal ha en tilknyttet enhetstest for å garantere funksjonalitet. Det skal også skrives integrasjonstester ved sammenkobling av funksjoner. Ved å kombinere disse testene med CI/CD-prinsipper vil prosjektet være sikret en minimum kvalitet på kode og produkt, som videre må kvalitativt testes.

3.5.2. Kontinuerlig evaluering

Det er viktig for prosjekts fremgang at oppdragsgiver utfører kontinuerlig evaluering av løsningen, slik at prosjektet opprettholder forventet standard. Som nevnt bruker prosjektet Scrum som utviklingsmetodikk, følgelig blir prosjektet fortløpende evaluert av oppdragsgiver ved sprintmøter.

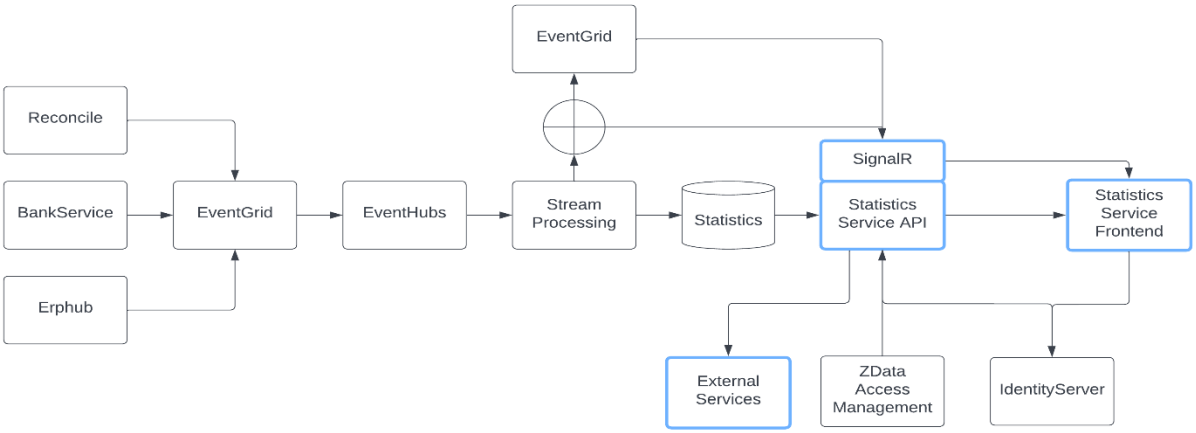
3.5.3. Sluttevaluering

Mot prosjektets slutt vil det bli utført en akseptansetest for å avgjøre om prosjektet oppfyller oppdragsgivers krav og behov. Testen vil bestå av en brukertest gjennomført av oppdragsgiver, samt et evalueringsskjema. Avslutningsvis vil det evalueres hvorvidt prosjektets mål har blitt møtt.

4. DESIGN OG UTVIKLING

4.1. Bakgrunn for design

Prosjektets design ble hovedsakelig inspirert av MVC-arkitektur, klient-tjener-mønster og moderne API-design. Dette resulterer i et system som har strenge ansvarsskiller og er delt opp i flere prosjekter, for bedre oversiktlighet. En viktig designbegrensning for prosjektet er åpne programmeringsgrensesnitt. Programmeringsgrensesnittet skal tilgjengeliggjøres for både eksisterende og fremtidige tjenester, mens brukergrensesnittet skal være et selvstendig prosjekt. Programmeringsgrensesnittet mottar aggregert statistikk fra en database. Denne databasen vil motta data fra hendelser i tjenestene til oppdragsgiver. Figur 9 skildrer hvor tjenesten som skal utvikles vil ta plass blant oppdragsgivers eksisterende løsninger.



Figur 9: Løsningens (markert i blått) kobling til oppdragsgivers eksisterende system

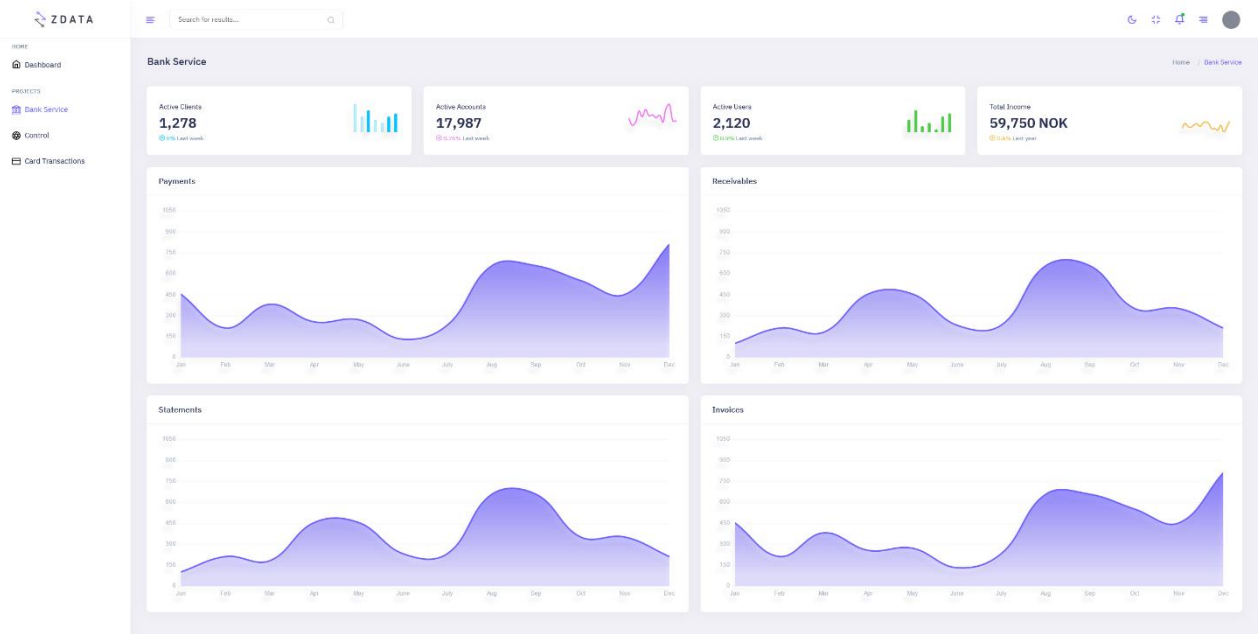
Oppdragsgiver har tre relevante produkter de ønsker å vise statistikk for i tjenesten. Produktene er Bank Service, Control og Settlement.

Bank Service er et programmeringsgrensesnitt som baserer seg på *Open Banking*-prinsippet. Open Banking går ut på at tredjepartsleverandører kan utvikle finansielle løsninger relatert til bankdata gjennom åpne programmeringsgrensesnitt. Bank Service dekker alle nordiske banker og støtter all standard finansiell data som kontoinformasjon, utbetalinger, innbetalinger og balanse på konto.

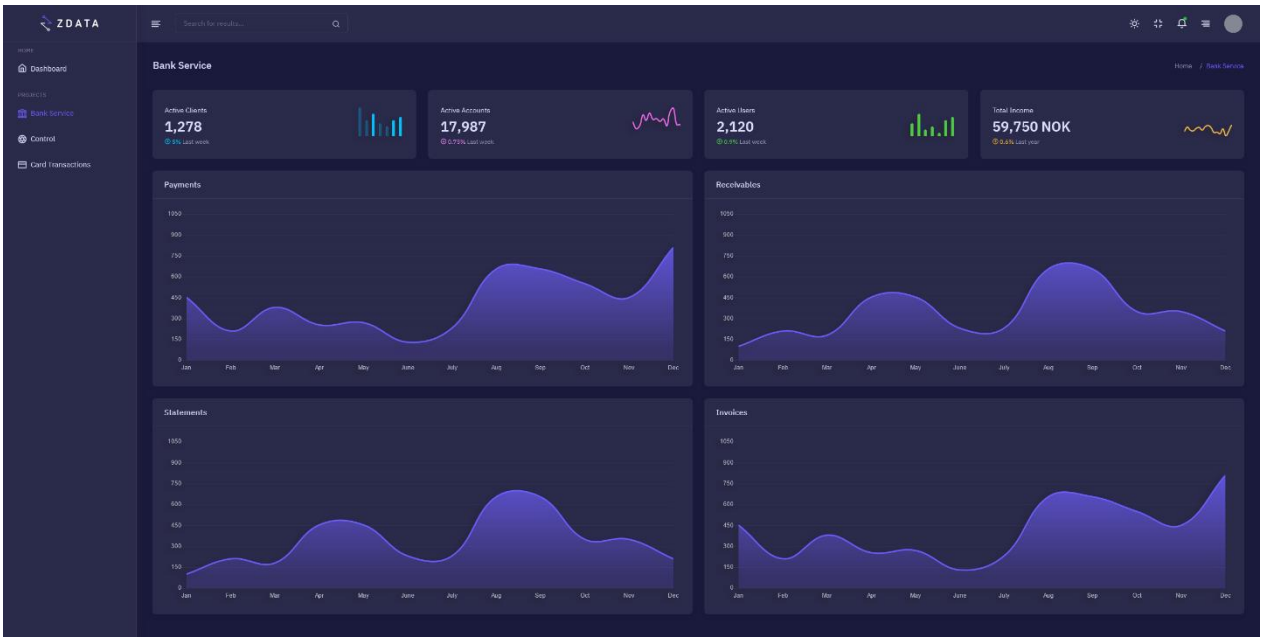
Control er en nettbasert avstemmingsløsning med direkte kobling til både bank og regnskapssystem. Transaksjoner hentes fortløpende fra begge sider, og avstemmes automatisk.

Settlement er en løsning som automatisk henter korttransaksjoner fra oppgjørsselskaper som Klarna eller Vipps, og bokfører de i regnskapssystemet.

Førsteutkastet av brukergrensesnittet fokuserte på selvstendige moduler for fremvisning av data, samt moderne, responsivt design. Figur 10 og figur 11 viser første utkast av Bank Service-siden i henholdsvis lys og mørk modus.



Figur 10: Førsteutkast av prosjektside i lys modus



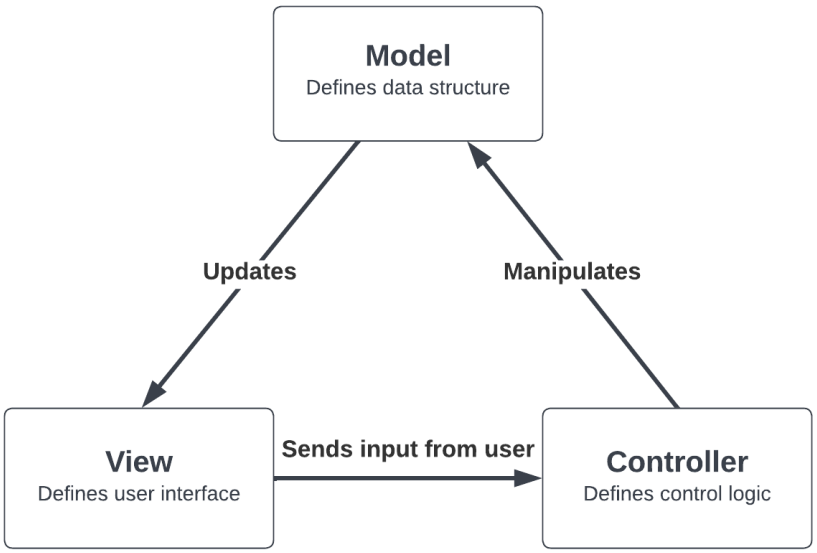
Figur 11: Førsteutkast av prosjektside i mørk modus

4.2. Arkitektur

Systemet skal kunne videreutvikles av oppdragsgiver etter at prosjektet er avsluttet. Det er derfor vesentlig for resultatet at systemarkitektur og kode holder høy standard. Dette sikres ved å utvikle gjenbrukbar kode, som gjør systemet skalerbart og lett å vedlikeholde. For å oppnå dette målet skal løsningen implementere Model-View-Controller-mønsteret.

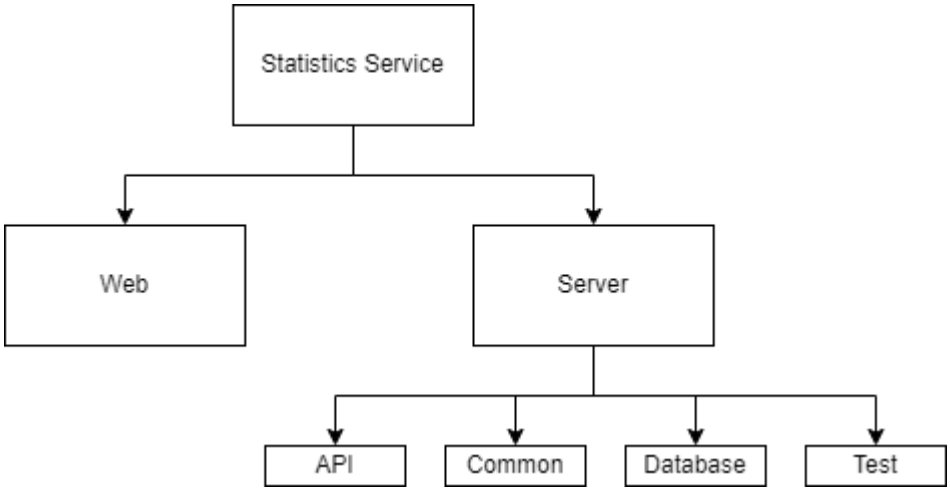
4.2.1. Model-View-Controller

Krasner og Pope (1988) beskriver Model-View-Controller-mønsteret (MVC) slik: "et treveis ansvarsskille, hvor objekter av forskjellige klasser overtar operasjonene knyttet til applikasjonsdomenet (modell), visningen av applikasjonens tilstand (visning) og brukerinteraksjonen med modellen og brukergrensesnittet (kontroller)". Mer konkret betyr dette at logikken bak brukergrensesnitt, datastruktur og kontroll er tydelig skilt og håndteres i egne komponenter med et gitt ansvarsområde. I tillegg til inndeling av ansvarsområder, definerer mønsteret samhandling mellom komponentene. Modellen er ansvarlig for datahåndtering, kontrolleren er megler mellom visningen og modellen, og visningen tegner en representasjon av modellen. Figur 12 beskriver hvordan MVC-mønsteret kan implementeres og hvordan komponentene samhandler.



Figur 12: Illustrasjon av MVC-arkitektur

Som nevnt skal prosjektet ha et tydelig ansvarsskille, dette gjenspeiles også i strukturen av løsningen som illustrert i figur 13. Brukergrensesnittet er implementert i Web-prosjektet og er ansvarlig for visning. Programmeringsgrensesnittet er implementert i Api-prosjektet og er ansvarlig for brukerinteraksjon med modellen. Common-prosjektet inneholder definisjoner av modeller og samhandlingslogikk. Database-prosjektet er ansvarlig for vedvaring av modeller. En mer detaljert beskrivelse av prosjektstrukturen finnes i systemdokumentasjonen (se vedlegg F, kapittel 3). Alle prosjektene i løsningen er uavhengig av hverandre, og samhandler kun gjennom gitte aksesspunkter, noe som blir nærmere beskrevet i kapittel 4.4.2.



Figur 13: Illustrasjon av løsningens prosjektstruktur

4.2.2. Dataoverføringsarkitektur

Som påpekt i kapittel 3.1, skal løsningen benytte seg av HTTP-protokollen for overføring av data. For å kunne tilby et forutsigbart programmeringsgrensesnitt, bør det benyttes en overordnet arkitektur eller metode for tilgjengeliggjøring av ressurser.

Et alternativ er Representational State Transfer (REST). REST ble foreslått av Roy Fielding i hans doktorgradsavhandling. Fielding (2000) definerer REST som “en arkitektonisk stil for distribuerte hypermediasystemer som tildeler et sett med arkitektoniske begrensninger som, når anvendt som en helhet, vektlegger skalerbarhet av komponentinteraksjoner, generalitet av grensesnitt, uavhengig distribusjon av komponenter, og mellomliggende komponenter for å redusere interaksjonsforsinkelse”. Mer konkret er REST en samling arkitektoniske begrensninger som legger til rette for forutsigbar dataoverføring mellom en klient og en tjener.

Et annet alternativ er teknologien GraphQL. GraphQL tillater klienter å spørre etter spesifisert data over ett enkelt endepunkt, lignende SQL. Hovedforskjellen er at REST er en arkitektur, mens GraphQL er en spesifikasjon og et spørringsspråk. Et annet viktig moment er at REST legger opp til flere ressursspesifikke endepunkter, hvor GraphQL tilbyr ett.

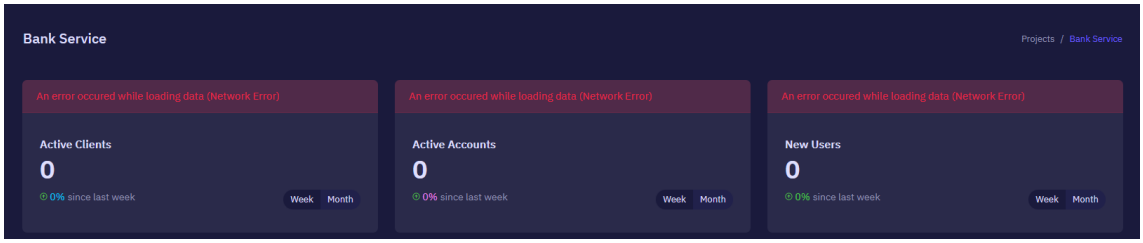
Oppdragsgiver ønsker REST, da programmeringsgrensesnittet skal kunne konsumeres av eksterne klienter og tjenester. I tillegg har oppdragsgiver ingen eksisterende tjenester som benytter seg av GraphQL, noe som gjør det unaturlig å tilføre denne teknologien til deres teknologistack.

4.3. Brukergrensesnitt

Brukergrensesnittet ble utformet av gruppen i henhold til førsteutkast som ble definert i kravdokumentet (se vedlegg E, kapittel 4), samt tilbakemeldinger fra oppdragsgiver. For brukergrensesnittet er det vektlagt god brukeropplevelse samt et moderne og responsivt design. Brukeropplevelsen skal sikres ved å følge Nielsens heuristikker (Nielsen, 2020) for brukervennlighet. Det er da spesielt fem heuristikker som skal tilstrebes:

1. **Synlighet av systemstatus** – La brukeren vite hva som skjer i systemet, eksempelvis lastingsindikatorer.
2. **Gjenkjennelse fremfor minne** – Nødvendig informasjon skal være lett tilgjengelig ved behov, samt at handlinger og alternativer skal være synlig.
3. **Feilforebygging** – Begrense brukerens muligheter for å gjøre feil.
4. **Estetisk og minimalistisk design** – Brukeren skal kun vises relevant informasjon.
5. **Hjelp brukere å gjenkjenne, diagnostisere og gjenopprette etter feil** – Gi tydelige feilmeldinger, slik at brukeren kan bedre forstå feilen.

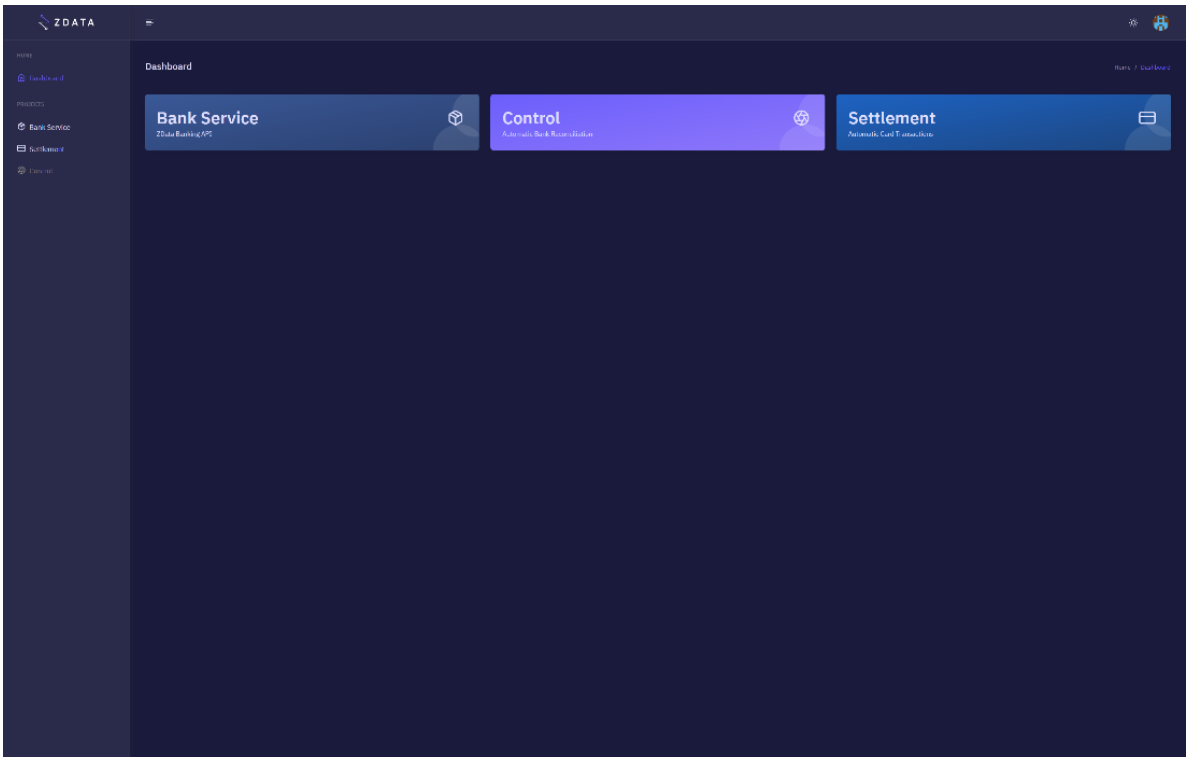
Figur 14 demonstrerer et eksempel av heuristikk 1, 2, 3 og 5. Applikasjonen viser tydelig at en feil har oppstått (1), brukeren ser et rødt felt som ikke var der tidligere (2), feilen krasjer ikke systemet (3), samt at feilmeldingen gitt til brukeren vil kunne forenkle diagnostisering av problemet (5).



Figur 14: Illustrasjon av feilmeldinger i separate moduler

4.3.1. Dashboard

Dashboard er landingssiden for applikasjonen og følgelig brukerens første møte med grensesnittet. Her vil brukeren få et overblikk over de tilgjengelige tjenestene, som illustrert i figur 15. I prototypen hadde landingssiden flere moduler. Det var tiltenkt at landingssiden skulle gi en oversikt over salg og andre referansepunkter på tvers av tjenestene. Da de tiltenkte modulene ikke var av høy prioritering ble det endret til en meny for å lett kunne navigere til de forskjellige prosjektene.



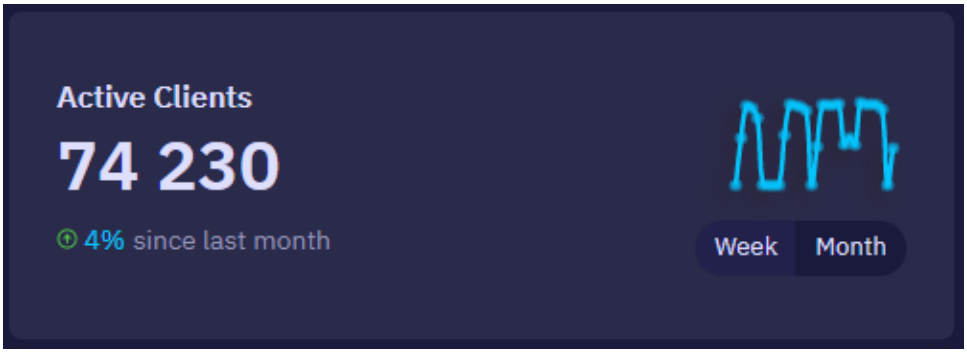
Figur 15: Landingsside for applikasjonen

4.3.2. Prosjekter

Som nevnt tidligere, skal brukergrensesnittet vise statistikk for tjenestene oppdragsgiver tilbyr sine kunder og partnere. Statistikken for hvert av tjenestene er i kontekst av brukergrensesnittet inndelt i to kategorier, henholdsvis aktivitet og tjenesteytelse.

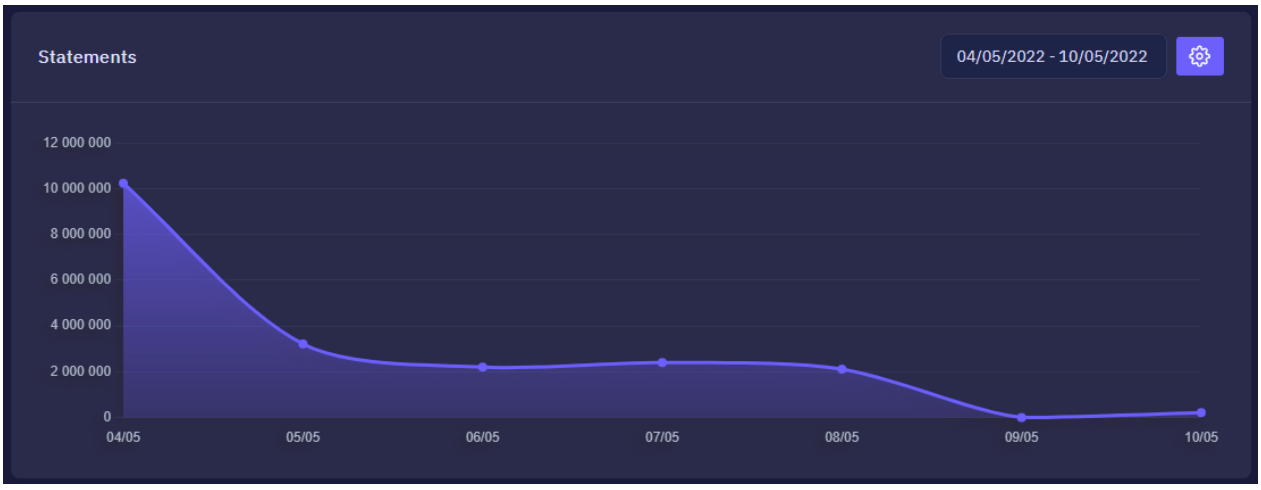
Når brukeren navigerer til en prosjektside vil overordnet statistikk bli vist. Øverst blant innholdet på siden vil små grafer vise informasjon om aktivitet i prosjektet. Videre var det et ønske fra

oppdragsgiver at en enkelt kunne sammenligne endringer fra foregående uke. Figur 16 viser hvordan informasjon presenteres i en liten graf.

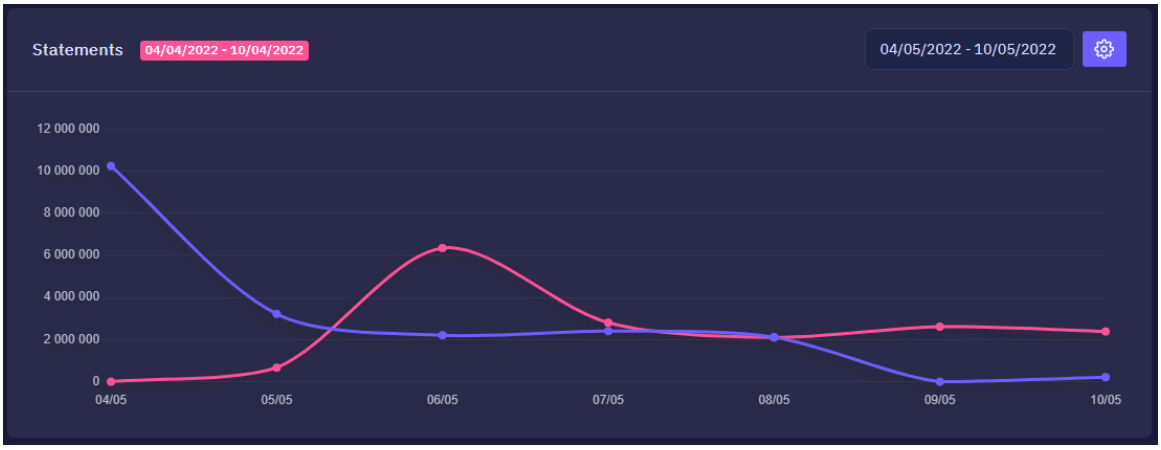


Figur 16: Liten graf med veksling mellom perioder

Statistikk for tjenesteytelse vises i store grafer som illustrert i figur 17. Brukeren kan for en gitt graf velge tidsintervall det skal vises statistikk for, samt sammenligne to perioder. Figur 18 viser et eksempel på sammenligning av perioder. Videre er det mulig for brukeren å se detaljert statistikk for et punkt ved å holde musepekeren over punktet.

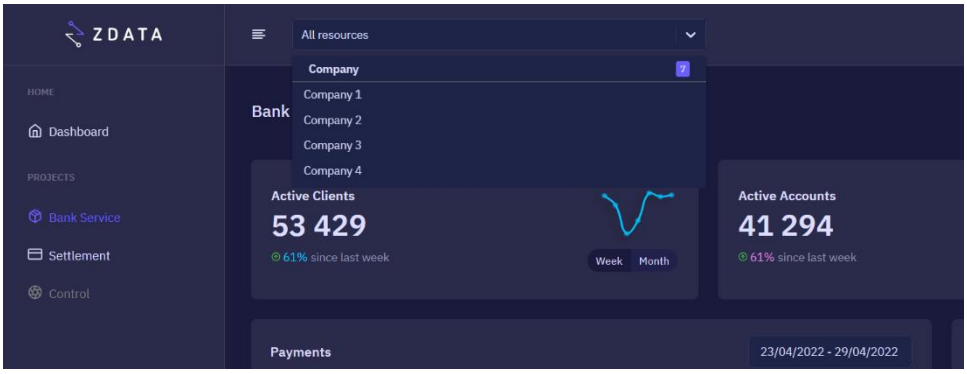


Figur 17: Stor graf med filtreringsmuligheter for periode



Figur 18: Stor graf med sammenligning mellom to perioder

Dersom brukeren ønsker å utforske en spesifikk klient kan dette velges i nedtrekksmenyen som illustrert i figur 19. I nedtrekksmenyen vil brukeren få en oversikt over alle klienter og partnere, samt kunne søke etter en ønsket klient (i kontekst av prosjekter er klient en kunde av oppdragsgiver). Dersom brukeren allerede er på siden til en klient vil en valgmulighet for alle ressurser være tilgjengelig øverst i menyen. Dette navigerer brukeren tilbake til siden med overordnet statistikk.



Figur 19: Åpen nedtrekksmeny med liste av anonymiserte klienter i Bank Service

4.3.3. Henting og lagring av data

Ettersom klientsiden skal motta store mengder data, er det nødvendig å implementere verktøy for lagring av tilstand og applikasjonsdata. NPM tilbyr metoder og biblioteker for å oppnå dette i React. Kravene for et lagringsbibliotek er liten størrelse, høy tilgjengelighet og dataintegritet.

Derfor er det foretrukket å ha en sentral lagringstjeneste som kan brukes globalt i løsningen, altså innad i og utenfor React-komponenter. Disse kravene kan oppfylles av biblioteket Zustand.

Zustand er et verktøy for å håndtere enkel lagring, og som blir benyttet for å holde på brukerinnstillinger som for eksempel å lagre brukerens ønsker om lys/mørk modus, brukerens autentiseringsbevis og lignende. Zustand ble valgt over mer populære biblioteker som Redux, da Zustand krever betydelig mindre gjentagende kode for å oppnå samme resultat. Videre ble det også behov for å kunne fortløpende hente, mellomlagre og oppdatere ikke-konstant applikasjonsdata, slik som statistikken som leveres i prosjektsidene. Det er mulig å gjøre dette med Zustand, men det vil kreve store mengder manuell konfigurasjon. Dette problemet ble derfor løst med biblioteket React-Query.

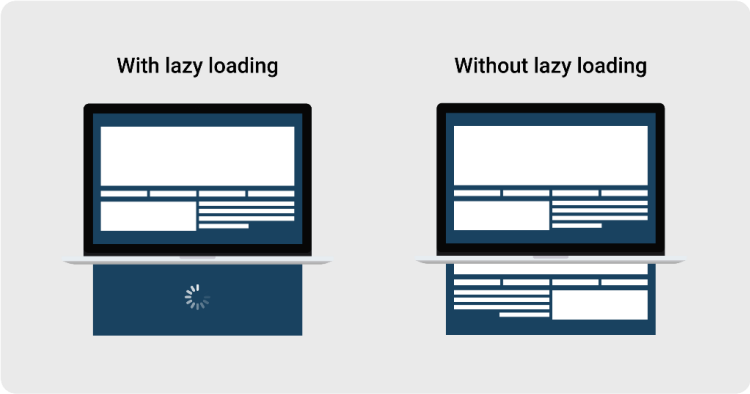
React-Query er et verktøy for å holde på mer avansert og asynkron data, og blir benyttet for å mellomlagre data som hentes ned fra programmeringsgrensesnittet. Dette gjør at data ikke må hentes på nytt før visse krav er møtt. Eksempelvis at ny data er tilgjengelig eller at intervall for henting av data er nådd. Verktøyet tilbyr også metoder for feilhåndtering som gir forutsigbarhet i feilsituasjoner.

For å hente data fra programmeringsgrensesnittet ble HTTP-biblioteket Axios valgt. Axios er et bibliotek som bygger på og forenkler eksisterende metoder i JavaScript. Axios oversetter automatisk dataobjekter til og fra JSON, tillater definisjon av gjenbrukbare klientinstanser, begrenser mengden gjentatt kode og har innebygd automatisk feilhåndtering. Dette gjør bruk og konfigurasjon av Axios langt lettere enn de innebygde HTTP-metodene i JavaScript.

4.3.4. Optimalisering av ytelse

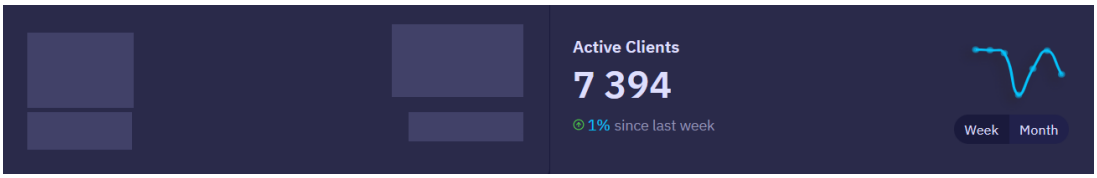
Det er begrensede muligheter for å optimalisere ytelsen i en CSR-applikasjon. Unntakene er å effektiviser kode, minimere statiske filer og implementere anbefalinger fra Google Lighthouse. Lighthouse har blitt brukt for å redusere innlastningstid ved å avdekke problemer som ikke sitter direkte i kodebasen, som ukomprimerte fonter og overflødige overføringen ved førstegangsinnlastning. Utover å optimalisere ytelsen kan det implementeres metoder som øker brukerens opplevde ytelse. En slik metode er *lazy loading*.

Lazy loading er en innlastingsmetode som gjør at applikasjonen kun laster inn innhold som er synlig for brukeren. Dette forhindrer innlasting av alt innhold ved første sideinnlasting, som reduserer innlastingstid. Figur 20 viser et eksempel på *lazy loading*.



Figur 20: Illustrasjon av sideinnlasting med *lazy loading* og uten *lazy loading* (Vats 2020)

En annen metode for økning av opplevd ytelse er skjelettinnlasting. Skjelettinnlasting er en innlastingsmetode som viser alt innhold på skjermen ved første mulighet, men ikke laster inn det faktiske innholdet før det er klart. Dette øker synligheten av systemet, ettersom brukeren kan se innholdet bli lastet inn, som illustrert i figur 21. Dette er en metode som er valgt over spinners (svart skjerm med spinnende hjul), da brukeren kan forvente hvor innholdet på siden vil være ved ferdig innlasting. Brukeren vil også kunne få en forventning om mengden innhold som eksisterer på siden.



Figur 21: Eksempel av skjelettinnlasting for liten graf. Ved innlasting vil skjelettet være animert.

4.3.5. Autentisering

Ettersom programmeringsgrensesnittet krever tilgangskontroll, er det nødvendig å autentisere brukere på klientsiden. Dette utføres av IdentityServer, en open-source implementasjon av OIDC-protokollen bygd på toppen av OAuth2. Innlogging blir gjort mot en frittstående instans av IdentityServer, som ved vellykket innlogging returnerer en autentiseringsnøkkel med relevant informasjon om brukerens rettigheter. Denne nøkkelen brukes deretter mot alle endepunkter som krever gyldig autentisering for tilgang.

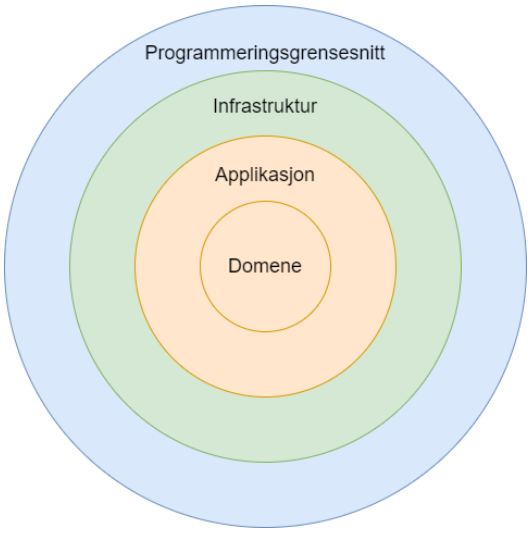
4.4. Programmeringsgrensesnitt

4.4.1. Clean Architecture

Clean Architecture er en designfilosofi, foreslått av programvareutvikleren Robert C. Martin, som vektlegger en lagdelt oppdeling av programvare og eksklusivt utgående avhengigheter, dvs. at et indre lag kan ikke være avhengig av et ytre lag.

En Clean Architecture-løsning starter alltid ved å definere et domene. Et domene består typisk av entiteter, grensesnitt og modeller som definerer løsningens problemstilling. Det følgende laget skal innkapsle domenet og definere forretningslogikk for applikasjonen. Deretter kan tilleggstjenester som databasetilkobling defineres, ofte som et infrastrukturlag. Ytterst i modellen er presentasjonslaget, som er ansvarlig for å implementere applikasjonens grensesnitt.

Domenet er definert av oppdragsgiver gjennom aggregeringstjenesten som leverer data til applikasjonsdatabasen. Resultatet av dette er at domenelaget i henhold til modellen er slått sammen med applikasjonslaget til én komponent som illustrert i figur 22.



Figur 22: Prosjektets implementasjon av Clean Architecture-filosofien

4.4.2. Mediator

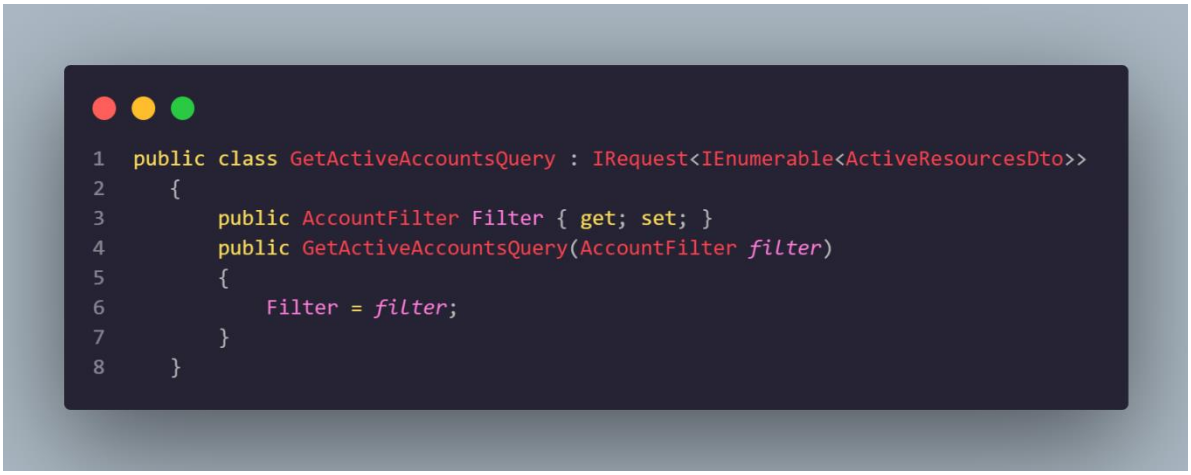
I større applikasjoner er sannsynligheten for å oppnå lenkede avhengigheter (rekursiv avhengighet mellom klasser over flere nivåer) stor. Dette kan føre til et uoversiktlig system med tett kopling.

Mediator-mønsteret løser dette ved at klasser ikke avhenger av hverandre, men kun mediatoren [formidleren]. Dette gjør at formidleren opptrer som det eneste aksesspunktet mellom lag i applikasjonen.

I prosjektet benyttes formidleren hovedsakelig for å koble endepunkter mot datahentingsmetoder i applikasjonslaget. Figur 23 viser et kodeutdrag for mottak og behandling av en spørring mot et endepunkt. Her vil informasjonen bli innkapslet i en klasse, som blir brukt for å instansiere spørringsklassen som sendes av formidleren. Figur 24 viser konstruktøren til spørringsklassen. Denne klassen arver fra *IRequest*-grensesnittet fra *MediatR*-pakken (en alternativ formidler-pakke i .NET). Alle nye klasser av denne typen som sendes av formidleren vil automatisk kartlegges til en håndteringsmetode. Håndteringsmetoden, som illustrert i figur 25, står ansvarlig for å returnere informasjon som er blitt forespurt av formidleren. Dette kan være returnering av konstanter direkte i metoden, eller å sende videre kall til en repository-metode. *Repository* vil bli beskrevet i neste kapittel.

```
1  [HttpGet]
2  public async Task<ActionResult<ActiveResourcesDto>> GetActiveAccounts([FromQuery] DateTime? from,
3                                                                         [FromQuery] DateTime? to,
4                                                                         [FromQuery] int? take,
5                                                                         [FromQuery] int skip = 0)
6  {
7      var filter = new AccountFilter(from, to, take, skip);
8      var widgets = await mediator.Send(new GetActiveAccountsQuery(filter));
9
10     if (widgets == null)
11         return NotFound();
12
13     return Ok(widgets);
14 }
```

Figur 23: Kodeutdrag fra en kontroller-klasse i programmeringsgrensesnittet



```
1  public class GetActiveAccountsQuery : IRequest<IEnumerable<ActiveResourcesDto>>
2      {
3          public AccountFilter Filter { get; set; }
4          public GetActiveAccountsQuery(AccountFilter filter)
5          {
6              Filter = filter;
7          }
8      }
```

Figur 24: Kodeutdrag fra en query-klasse i programmeringsgrensesnittet

```
1  public async Task<IEnumerable<ActiveResourcesDto>> Handle(GetActiveAccountsQuery query, CancellationToken cancellationToken)
2  {
3      return await accountsRepository.GetAllActiveAccounts(query.Filter);
4  }
```

Figur 25: Kodeutdrag fra en handler-klasse i programmeringsgrensesnittet

Bruk av en formidler kan fjerne avhengigheter mellom klasser, noe som resulterer i lavere kopling og øker muligheten for gjenbruk av kode. Videre følger mønsteret *single responsibility*-prinsippet (SRP) som sier at en modul eller klasse bare skal ha ansvar for én enkelt del av applikasjonens funksjonalitet. Ved å følge SRP kan applikasjonen sikres mot uventede bivirkninger av fremtidige endringer i en vilkårlig komponent.

En mulig risiko ved å bruke en formidler er at den kan bli så avgjørende for løsningen at den opptrer som et Gud-objekt. Et Gud-objekt er et objekt som holder en altfor sentral rolle i løsningen. Følgen av dette er tett kopling, noe som strider med hensikten av Mediator-mønsteret.

4.4.3. Repository-mønster

Repository-mønsteret er et mønster som kan anvendes for å danne et abstraksjonslag over operasjoner mot en databasetilkobling. Dette betyr at all samhandlingslogikk mot databasen holdes samlet på ett sted, noe som kan øke løsningens ryddighet. I løsningen er alle repository-klasser definert gjennom et grensesnitt. Alle klasser som er avhengig av et repository er avhengig

av grensesnittet for klassen. Dette gjør at hele repository-klassen kan veksles ut, eksempelvis ved skifte av database, uten at dette krever endringer i avhengige klasser. Mønsteret forenkler også testing, da databasekonteksten i repository-klassene er konfigurerbar ved instansiering. Dette gjør at man kan veksle ut koblingen mot hoveddatabasen for en kobling til en test-spesifikk database ved utførelse av tester. Dette sikrer at metodene som utføres i testing er identisk til de som brukes i kjøring.

4.4.4. Object-relational mapper (ORM)

En vanlig utfordring for applikasjoner med vedvarende data er dataoverføring mellom server og database. Verdier må kunne hentes ut av et dataobjekt og lagres i databasen. Det må også være mulig å hente ut verdier fra databasen og sette dem inn igjen i et dataobjekt. Dette er tungt å utføre manuelt, følgelig er det en sjelden strategi å bruke i moderne utvikling. En løsning til dette problemet er en ORM.

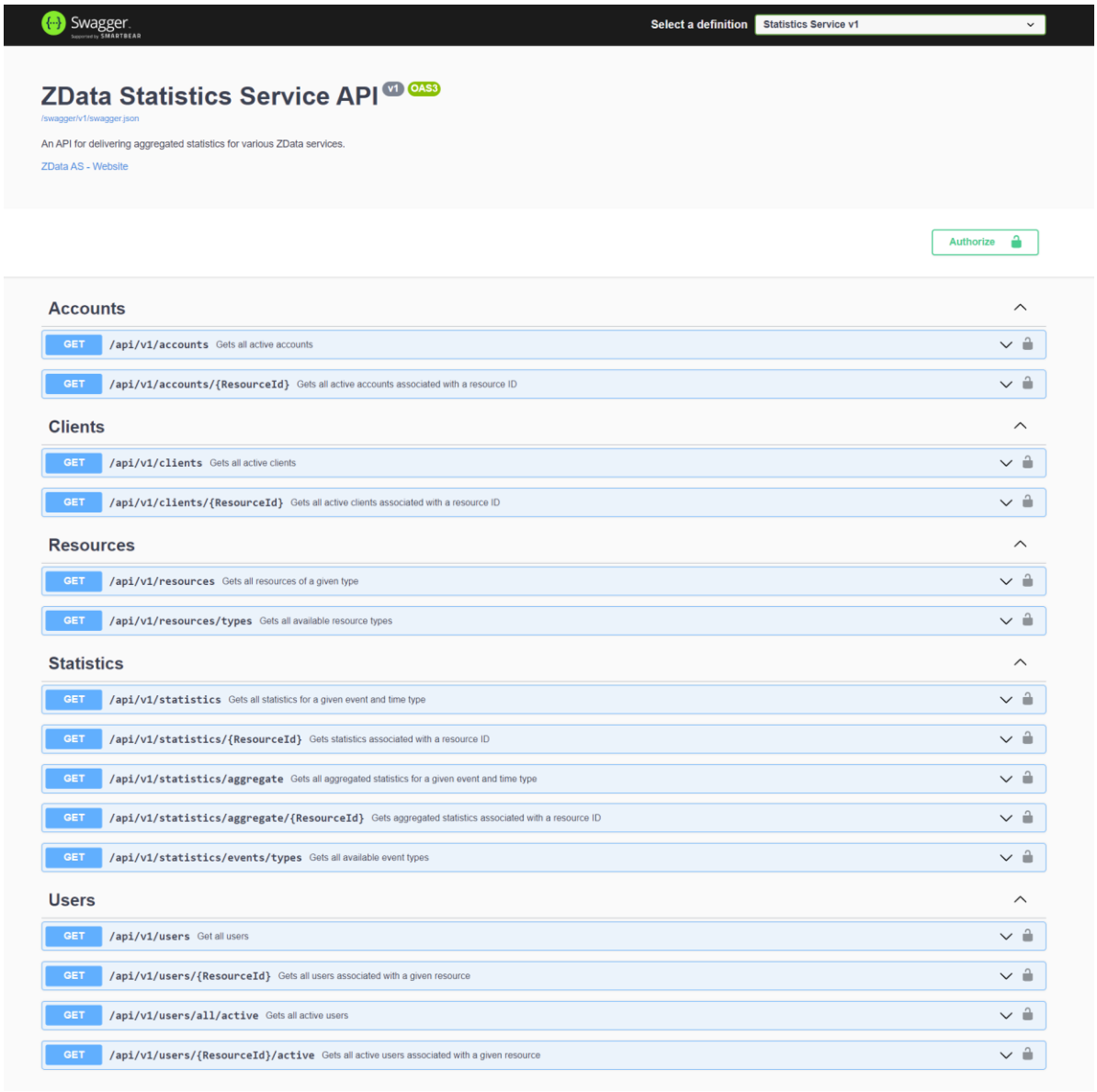
En ORM vil automatisk overføre data i en relasjonsdatabase til dataobjekter i et objekt-orientert system. For .NET Core-applikasjoner som trenger en databasetilkobling, er det normalt å bruke Entity Framework Core (EF Core). I EF Core brukes en modell for å få datatilgang. Denne modellen består av entiteter og en databasekontekst (som representerer en sesjon med databasen). Databasekonteksten tilgjengeliggjør spørringer og lagring av data ved hjelp av Language-Integrated Query (LINQ), som tillater skriving av spørringer i C#.

LINQ fjerner behovet for eksplisitte SQL-spørringer (manuelt skrevne SQL-spørringer hvor variabler settes inn virkårlig) blir systemet betraktelig mindre utsatt for SQL-injeksjonsangrep. Videre vil EF Core sjekke og rense inngående data for uønskede tegn eller strenger. Dette vil ytterligere sikre systemet mot angrep. Videre kan benyttelse av en ORM kan resultere i et kortere utviklingsløp ettersom behovet for å kartlegge spørringsresultater til et objekt, og omvendt, elimineres.

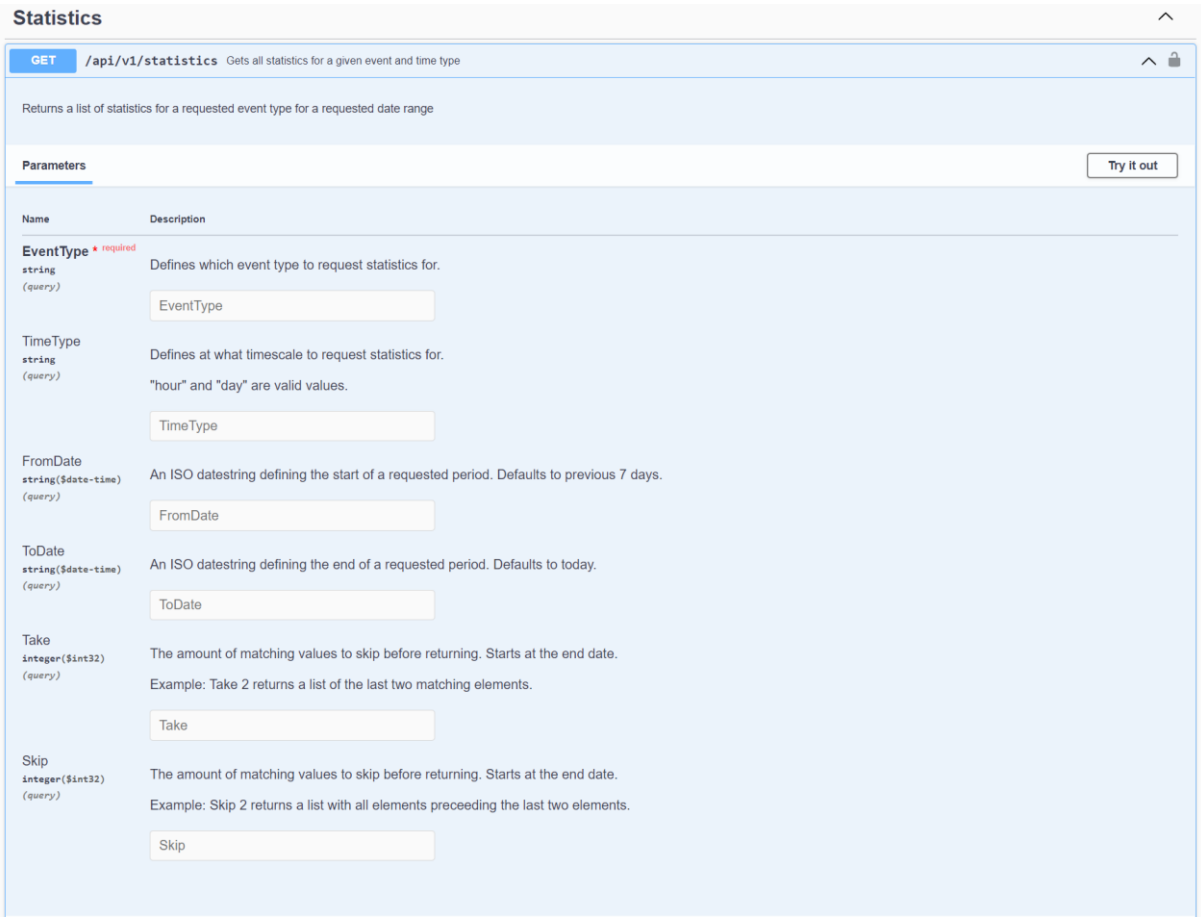
4.4.5. Endepunkter

Ettersom programmeringsgrensesnittet følger REST-prinsippene, vil all kommunikasjon gjøres gjennom ressursspesifikke endepunkter. Alle endepunktene benytter seg av versjonering. Dette gjør at grensesnittet kan oppdateres senere, uten å gi uforutsette konsekvenser for tjenester og klienter som avhenger av en gitt versjon av et endepunkt.

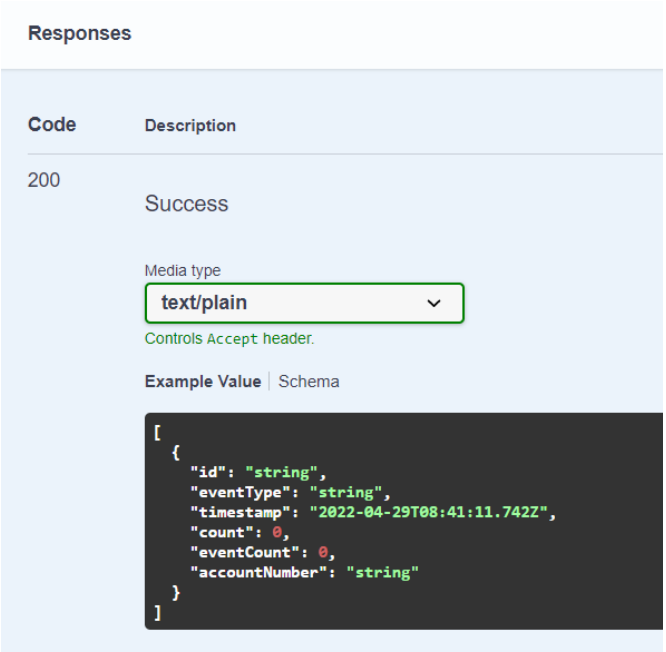
Programmeringsgrensesnittet er dokumentert både i kode og gjennom dokumentasjonsverktøyet Swagger, som illustrert i figur 26. Swagger gir et overordnet blikk over tilgjengelige endepunkter, samt forventet inngående data og utgående datatyper for ethvert endepunkt, som vist i figur 27 og 28. Swagger gir også mulighet for å gjøre forespørsler mot grensesnittet fra dokumentasjonssiden. Endepunktenes egenskaper er nærmere beskrevet i systemdokumentasjonen (se vedlegg F, kapittel 6).



Figur 26: Tilgjengelige endepunkter i Swagger-dokumentasjonen



Figur 27: Eksempel på forespørsel i et endepunkt for å hente ut statistikk i Swagger-dokumentasjonen



Figur 28: Eksempel på vellykket respons fra et endepunkt for å hente ut statistikk i Swagger-dokumentasjonen

4.4.6. Innkapsling av forespørselsparametere

Ettersom alle endepunkter i prosjektet skal levere data i lignende format, ble det naturlig å benytte de samme basisparameterne på alle endepunkter. Disse parameterne består av *FromDate* og *ToDate*, som definerer tidsperioden man ønsker å filtrere for. Samt *Skip* og *Take*, som blir videre brukt av *Skip*- og *Take*-funksjonene i LINQ som begrenser antall objekter returnert fra forespørselen.

For å implementere disse parameterne på en forutsigbar og enkel måte, ble parameterne innkapslet i en klasse (*BaseFilter*), som definerer hvilke parametere som skal inkluderes, samt hvordan de skal defineres ved manglende input fra brukeren. Denne klassen vil da sette *ToDate* til dagens dato, hvis ingen *ToDate* er supplert, og *FromDate* vil settes til en uke før *ToDate*, dersom ingen *FromDate* er supplert i forespørselen. Videre har *Skip* standardverdi på 0, og *Take* er udefinert (dvs. den returnerer alle objekter om ikke *Take* er supplert).

En slik innkapslingsklasse tillater også dokumentasjon av alle parametere på ett sted, noe som gjør at beskrivelsen av for eksempel en *ToDate* blir lik over alle endepunkter i Swagger-dokumentasjonen. Dette er gjort ovenfor å skrive individuell parameterbeskrivelse for hvert endepunkt. Et eksempel på dokumentasjonen av parametere kan sees i figur 29.

```
1  public class BaseFilter : IBaseFilter
2  {
3
4      /// <summary>
5      /// An ISO datestring defining the start of a requested period. Defaults to previous 7 days.
6      /// </summary>
7      public DateTime FromDate { get; set; }
8
9      /// <summary>
10     /// An ISO datestring defining the end of a requested period. Defaults to today.
11     /// </summary>
12     public DateTime ToDate { get; set; } = DateTime.Now;
13
14     /// <summary>
15     /// The amount of matching values to skip before returning. Starts at the end date.
16     /// <br/><br/> Example: Take 2 returns a list of the last two matching elements.
17     /// </summary>
18     public int? Take { get; set; }
19
20     /// <summary>
21     /// The amount of matching values to skip before returning. Starts at the end date.
22     /// <br/><br/> Example: Skip 2 returns a list with all elements preceeding the last two elements.
23     /// </summary>
24     public int Skip { get; set; } = 0;
25
26     public BaseFilter()
27     {
28         FromDate = ToDate.AddDays(-6).Date;
29     }
30 }
```

Figur 29: Kodeutdrag fra filterklasse i programmeringsgrensesnittet

For endepunkter som krever flere parametere som ressurs-id eller *EventType*, er det utviklet flere klasser som arver fra BaseFilter-klassen, samt et grensesnitt for filteret. Dette gir gjennomgående forutsigbarhet i parameterdefinisjon og forventninger rundt standardoppførsel.

4.4.7. Autentisering

Som nevnt tidligere krever programmeringsgrensesnittet autentisering. Da autentisering er gjort gjennom en ekstern tjeneste, skal programmeringsgrensesnittet verifisere gyldigheten av en brukers autentiseringsnøkkel. All verifisering av nøkkelen gjøres med innebygde metoder fra ASP.NET, noe som forenkler utviklingen og sikrer mot feil.

4.4.8. Bruk av eksterne programmeringsgrensesnitt

Det blir benyttet et eksternt programmeringsgrensesnitt fra oppdragsgiver, kalt Resource Service API. Dette grensesnittet anvendes for å få ut informasjon om tilgjengelige ressurser (dette er ofte klienter og partnere til oppdragsgiver). Denne informasjonen blir lagt inn i nedtrekksmenyen i brukergrensesnittet (figur 19). Et eksempel på en slik ressurs er illustrert i figur 30.

Figur 30: Eksempel på respons fra forespørsel mot Resource Service API for en gitt ressurs

ZData Access Management API er et programmeringsgrensesnitt som behandler tilgangskontroll for oppdragsgivers klienter og ansatte i henhold til deres produkter. I prosjektet blir dette brukt til å hente ut partner-ressurser. Dersom en kundeportal blir utviklet kan dette grensesnittet også benyttes for å identifisere en gitt brukers tilgangsnivå.

For loggføring blir det benyttet Sentry, et open-source system som støtter sanntidssporing av feil. Sentry tilbyr, i tillegg til loggførings- og feilsporingsverktøy, et brukergrensesnitt som forenkler å finne den relevante feilmeldingen til enhver tid. All kommunikasjon med Sentry gjøres over ett enkelt endepunkt.

4.4.9. CI/CD

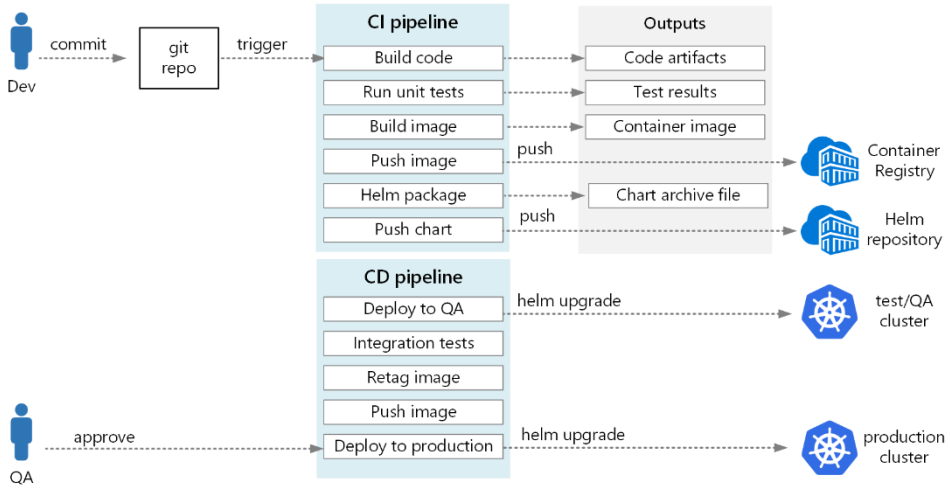
Prosjektet blir automatisk bygget ved pull-forespørsler til hovedgrenen i Git, etter hvert som nye funksjoner blir ferdigkodet. Utvikleren som produserte koden lager en pull-forespørsel som blir lastet opp for gjennomgåing. Når koden er gjennomgått og godkjent vil endringene overføres til hovedgrenen. Etter koden er integrert i hovedgrenen starter en automatisert bygge- og utrullingsprosess. Denne prosessen står ansvarlig for å bygge, teste, pakke og levere koden til relevante Azure-tjenester.

Under utviklingen brukes to forskjellige miljøer. Et utviklingsmiljø som alltid har den nyeste programvaren, og et produksjonsmiljø som bare har stabil og testet programvare. For omverdenen finnes bare produksjonsmiljøet, ettersom utviklingsmiljøet bare er tilgjengelig internt. Etter hvert som ny programvare i utviklingsmiljøet blir godkjent av oppdragsgiver vil den rulles ut til produksjonsmiljøet.

Det skal også skrives enhetstester med mål om å oppnå størst mulig testdekning. Dette gjøres for å sikre at produktet som leveres til utviklings- og produksjonsmiljøet er kjørbart, da feilede tester vil avbryte utrullingen.

Terraform har blitt benyttet for å få satt opp de ulike miljøene i Azure, dette er grunnet krav fra oppdragsgiver at all infrastruktur skal være i kodeformat. For pakking av programvare er det benyttet Docker, en teknologi basert på containerisering. Containerisering er pakking av programvarekode med nødvendige operativsystembiblioteker og avhengigheter for at applikasjonen skal kunne kjøres. Dette skaper en kjørbar fil (en kontainer) som kjører konsekvent på enhver infrastruktur (IBM Cloud Education, 2021). Docker blir brukt for å produsere identiske instanser av tjenesten, som kan vilkårlig instansieres i skyen ved behov. For levering av Docker-

kontainere brukes Kubernetes. Dette er en plattform for automatisering av prosessene involvert i utrulling, vedlikehold og skalering av containeriserte applikasjoner. Figur 31 beskriver prosessen med bygging og levering av koden for nevnte miljøer.



Figur 31: Beskrivelse av CI/CD-prosessen med Kubernetes i Azure (Price, 2022)

4.5. Database

4.5.1. Valg av database

Valget av database ble gjort av oppdragsgiver, da de var ansvarlige for å sette opp infrastruktur rundt løsningen, samt å aggregere statistikk inn i databasen. I oppstartsmøtene med oppdragsgiver ble det nevnt at MongoDB var en aktuell database å ta i bruk. Derimot ble det senere (omtrent halvveis) i prosjektløpet bestemt at oppdragsgiver skulle sette opp en PostgreSQL-database med en utvidelse kalt TimescaleDB.

4.5.2. Design

Databasen ble designet av oppdragsgiver og inneholder to tabeller (se figur 32) gruppert på henholdsvis dags- og timesbasis. Dataene er gruppert på *topic_id* (gjenspeiler hver klient) og på *event_type* (som er for eksempel utbetalinger, innbetalinger, kontoutskrifter og lignende).

aggregated_events_1_day	
id	**int**
event_time	timestamp
topic_id	varchar
event_type	varchar
count	float
event_count	float
resource_type	varchar

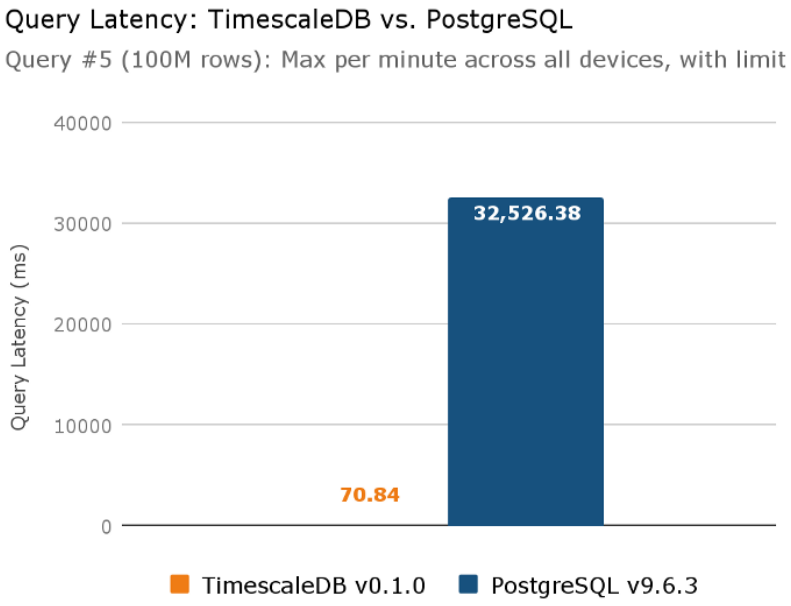
aggregated_events_1_hour	
id	**int**
event_time	timestamp
topic_id	varchar
event_type	varchar
count	float
event_count	float
resource_type	varchar

Figur 32: Illustrasjon av tabeller i databasen

4.5.3. TimescaleDB

TimescaleDB ble valgt av oppdragsgiver grunnet at databasetypen passer godt til oppgavens formål. Det er en database som har god støtte for tidsrekke-data, som er hva strukturen til statistikken består av, samt at den skalerer godt med store mengder data.

Sammenlignet med PostgreSQL, har TimescaleDB vesentlig bedre ytelse på store datasett. Basert på en undersøkelse gjort av Timescale, er forsinkelsen på en spørring for å hente ut 100 mill. rader med data hele 450 ganger raskere i TimescaleDB sammenlignet med PostgreSQL (se figur 33).



Figur 33: Forsinkelsen på en spørring i TimescaleDB vs. PostgreSQL på datasett med 100 millioner rader (Kiefer, 2017)

5. RESULTATER

5.1. Evalueringsmetode

Som nevnt skal prosjektet hovedsakelig benytte seg av tre evalueringsformer; systemtesting, kontinuerlig evaluering gjennom sprintmøter og en avsluttende akseptansetest utført av oppdragsgiver. Sammen skal disse metodene kunne gi et overblikk over prosjektets gjennomførelse i henhold til planlagt funksjonalitet og mål.

5.1.1. Systemtesting

Systemtesting består av automatiserte tester som blir utført som en del av CI/CD. I prosjektets kontekst har dette hovedsakelig bestått av enhetstester, da applikasjonen ikke har mye kompleks forretningslogikk. De viktigste funksjonene å implementere enhetstester for er alle *repositories*, hvor all samhandlingslogikk mot databasen er definert. Dette er for å sikre at all data som blir hentet ut blir levert på riktig format, samt i riktige mengder.

De overnevnte testene ble ikke skrevet før avsluttende fase av prosjektet, noe som tillot en håndfull logiske feil å slippe gjennom til de tidlige utrullingene av prosjektet. Dette ble derimot korrigert fortløpende.

Alle enhetstester som tester samhandlingslogikk mot databasen starter ved å sette opp en lokal og unik database, kjørt i systemminnet, som injiseres inn i repositoriet som skal testes. Dette sikrer at ingen endringer utført under testing vil reflekteres til utviklingsdatabasen. Test-databasen destrueres ved endt test for å sikre at all testdata blir fjernet, samt frigjøre det brukte minnet.

Videre er alle tester satt opp i AAA-mønsteret (Arrange, Act, Assert). Dette mønsteret gir tydelige skillelinjer for hvor man setter opp forventet og faktisk data (Arrange), hvor de faktiske dataene blir mutert (Act) og hvor man sjekker at de faktiske dataene stemmer overens med forventet data (Assert). I figur 34 representerer *expectedData* de aggregerte dataene per dato og *expectedMeta* representerer det totale antallet unike klienter over den testede perioden. Listen som testes inneholder verdier for de siste to ukene, samt duplikater av samme klient ID-er. Dette gjør at summen av unike klienter per dag ikke samsvarer med totalt antall unike klienter over perioden. Dersom selskap A og selskap B gjør én transaksjon hver, per dag, i en uke, vil dette gi to unike aktive klienter per dag, men også bare to aktive klienter for perioden.

```
[TestMethod]
public async Task GetAllClients()
{
    //Arrange
    var options = new DbContextOptionsBuilder<StatisticsServiceDbContext>()
        .UseInMemoryDatabase(databaseName: "test-get-all-clients")
        .Options;

    var context = new StatisticsServiceDbContext(options);

    await context.AggregatedEventsDay.AddRangeAsync(TestEvents());
    await context.SaveChangesAsync();

    var clientsRepository = new ClientsRepository(context);
    var filter = new BaseFilter();

    var expectedData = new List<ClientDto>()
        {
            new ClientDto(now.AddDays(-3), 1),
            new ClientDto(now.AddDays(-2), 1),
            new ClientDto(now.AddDays(-1), 2),
            new ClientDto(now, 2)
        };

    var expectedMeta = 4;

    ActiveResourceDto expected = new ActiveResourceDto(expectedData, expectedMeta);

    //Act
    var actual = await clientsRepository.GetAllActiveClients(filter);

    //Assert
    actual.Should().NotBeNull().And
        .BeOfType<ActiveResourceDto>().And
        .BeEquivalentTo(expected);

    //Cleanup
    context.Database.EnsureDeleted();
}
```

Figur 34: Kodeutdrag fra testklasse med testmetode for uthenting av alle klienter i databasekontekst

5.1.2. Kontinuerlig evaluering

I prosjektløpet har oppdragsgiver utført kontinuerlig evaluering på løsningen ved sprintmøter. Dette har gitt oppdragsgiver mulighet til å gi tilbakemeldinger på produktet i daværende tilstand.

Samtidig kunne oppdragsgiver få tilbakemelding fra gruppen på uklarheter eller manglende ressurser for å kunne fortsette i utviklingsløpet.

Denne evalueringsmetoden har hatt som mål å sikre at sluttproduktet stod til den forventingen oppdragsgiver hadde da prosjektet startet. Avslutningsvis har metoden gitt gruppen mulighet til å tilpasse arbeidet, samt å endre fokusområder etter behov.

5.1.3. Akseptansetest

Som nevnt bestod akseptansetesten av en brukertest og et tilhørende evalueringsskjema. Akseptansetesten har som mål å teste at løsningen tilfredsstiller oppdragsgivers ønsker og om de funksjonelle kravene er møtt. Brukertesten ble utført av oppdragsgiver etter avsluttende sprintperiode, hvor oppdragsgiver da hadde tilgang til produktet i produksjonsmiljøet.

Evalueringsskjemaet kan sees i vedlegg B. Skjemaet ble utformet av gruppen, og består av 8 påstandsspørsmål hvor svarene er rangert mellom 1 og 5 følgende av Likert-skalaen (Malt og Grønmo, 2020), samt 3 kortsvarspørsmål. Oppdragsgiver hadde skjemaet tilgjengelig ved brukertesten.

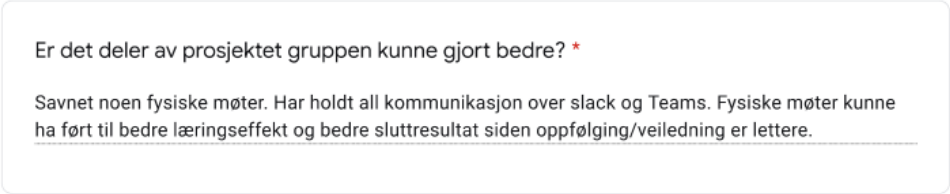
5.2. Evalueringsresultat

Systemtestene har bidratt i sikring av systemets kodekvalitet, samt å avdekke logiske feil i databasespørringer. Videre har testene hindret utrullinger med feil i koden. Dette har resultert i at programmeringsgrensesnittet opptrer som forventet.

Som nevnt ble visjon og kravspesifikasjonen for prosjektet endret tidlig. Opprinnelig var hensikten å lage et program som ville automatisk fakturert oppdragsgivers kunder basert på tjenesteforbruk. Derimot ble dette endret tidlig i utviklingsfasen, til en ren statistikkgenererende tjeneste. Ettersom en slik endring kom etter prosjektet allerede var i gang, måtte forventningene tilpasses deretter. Kontinuerlig evaluering viste seg dermed å være verdifullt for å sikre at produktet som ble utviklet møtte forventingen til oppdragsgiver.

Oppdragsgiver utførte akseptansetesten etter avsluttende sprintmøte. Av de 8 påstandsspørsmålene var 7 svar rangert svært høy (5). Det eneste oppdragsgiver rangerte lavere (4) var gruppens kommunikasjon og arbeidsflyt. Dette skyldes antakeligvis lite kommunikasjon utover de fastsatte

sprintmøtene. Videre, i kortsvarseksjonen, ga oppdragsgiver uttrykk for at det var mangel på fysiske møter, da de kan gi større læringseffekt og bedre sluttresultat (se figur 35).



Figur 35: Utdrag fra svar på evalueringsskjema (vedlegg B)

Oppdragsgiver ga også uttrykk for at de hadde manglende kapasitet for oppfølging, som resulterte i forsinket start på serversiden av løsningen. Til tross for den lave kapasiteten mener oppdragsgiver at resultatet er svært bra og skal bygges videre på i framtiden (se figur 36).

Figur 36: Utdrag fra kortsvar på evalueringsskjema (vedlegg B)

Samlet sett, vil kombinasjonen av systemtestingen, den kontinuerlige evalueringen og akseptansetesten, gitt til oppdragsgiver i sluttfasen av prosjektets utvikling, tyde på at prosjektet leverte til oppdragsgivers forventning.

5.3. Prosjektresultat

Resultatet av prosjektet er en løsning som forenkler måten bedriften henter ut statistikk for de ulike tjenestene sine. Resultatet er en direkte implementasjon av løsningsdesignet beskrevet i kapittel 4. Som nevnt inneholder løsningen et generisk programmeringsgrensesnitt som er enkelt

for andre tjenester å ta i bruk, samt at det blir konsumert av et brukergrensesnitt som visualiserer statistikken.

Prosjektresultatet blir til dels målt opp mot de ønskede funksjonelle og ikke-funksjonelle egenskapene (krav) som er beskrevet i visjonsdokumentet (se vedlegg C). De funksjonelle egenskapene er delt i to kategorier, henholdsvis *frontend* (klientsiden) og *backend* (serversiden).

På klientsiden er 8 av 9 funksjonelle egenskaper dekket:

1. Innloggingsfunksjonalitet – implementert med ZDatas IdentityServer
2. Tilgangskontroll - håndtert med ZData Access Management API
3. Funksjon(er) for å hente ned data fra endepunkter i API – implementert med Axios
4. Funksjon for å visualisere data i visningen – implementert med kodebiblioteket Chart.js
5. Lagre data i cache og synkroniser ved ny tilgjengelig data – håndtert med React Query
6. Funksjon(er) for synkronisering av data ved melding fra serversiden – ikke utført
7. Funksjon for filtrering av data i visningene – implementert ved funksjon som filtrerer data på datointervall
8. Funksjon for å søke i historikk
9. Systemtesting – utført ved tester i Google Lighthouse

På serversiden er 9 av 13 funksjonelle egenskaper dekket:

1. Implementere autentisering av bruker – implementert med ZDatas IdentityServer
2. Implementere tilgangskontroll – håndtert med ZData Access Management API
3. Tilkobling til database – tilkoblet med funksjonalitet fra Entity Framework
4. Funksjoner for henting av brukerdata fra database – utført
5. Funksjoner for henting av statistikk fra database – utført i alle statistikk-relaterte endepunkter
6. Aggregere data til riktige/nyttige formater – utført før levering av data fra endepunkter
7. Implementere endepunkter for uthenting av data basert på REST - utført
8. Sett opp API dokumentasjon - utført ved hjelp av Swagger
9. Implementere SignalR – ikke utført
10. Integrere ZData EventHub – ikke utført
11. Utvikle enhetstester - utført ved hjelp av MSTest
12. Utvikle integrasjonstester – ikke utført
13. Utvikle ende-til-ende tester – ikke utført

Videre måles prosjektresultatet mot målene som er satt for prosjektet. Hovedmålet var som nevnt å utvikle en applikasjon som leverer statistikk på tvers av tjenestene til oppdragsgiver. Dette målet ga to delmål. Første delmål var å utvikle et programmeringsgrensesnitt som aggregerer statistikk og tilgjengeliggjør for andre tjenester. Andre delmål var å utvikle et brukergrensesnitt som konsumerer programmeringsgrensesnittet og fremstiller uthentet data.

Figur 37: Utdrag fra svar på evalueringsskjema (vedlegg B)

Prosjektmålene ble postulert fra oppdragsgivers forventninger og visjon for løsningen. I oppdragsgivers svar på evalueringsskjema, vist i figur 37, ga de utrykk for at deres tilbakemeldinger påvirket prosjektet mot ønsket visjon. Dette tyder på at prosjektresultatet står sterkt mot denne visjonen. Videre tyder tilbakemelding fra oppdragsgiver (se figur 36) på at de er fornøyd med den foreslåtte løsningen. Dermed kan det antas at løsningsforslaget står som et formålstjenlig svar til problemstillingen.

5.4. Prosjektgjennomføring

Prosjektet ble gjennomført i perioden januar 2022 til mai 2022. Det har blitt forsøkt å fordele arbeidsoppgaver jevnt over perioden mellom medlemmene. I prosjektets startfase var mengden tilgjengelige arbeidsoppgaver begrenset, derimot ble flere parallelle oppgaver tilgjengelig utover i løpet. For arbeidsoppgavene er det satt et timebudsjett på 540 timer per medlem, altså et total budsjett på 2160 timer. I løpet av prosjektperioden har det blitt dedikert 1090 timer. Gjenstående aktiviteter etter prosjektslutt består av refleksjonsnotat, sluttpresentasjon og EXPO. Prognosen for tidsbruk for nevnte aktiviteter er 112 timer totalt, altså 28 timer per gruppemedlem. Dette resulterer i at totalt dedikerte timer er 1202, som er omtrent halvparten av det totale budsjettet.

Ukentlige timelister og statusrapporter er tilgjengelig i prosjekthåndboken (se vedlegg D, kapittel 4).

Fremdriften over prosjektperioden har vært jevn, derimot noe begrenset tidlig i løpet grunnet manglende ressurser (databasetype, dataformat) fra oppdragsgiver. Dette er et syn som også er delt av oppdragsgiver (se figur 34). Den initielle planen var å påbegynne serversidefunksjonalitet i tidlig mars, men dette ble utsatt til månedsskiftet mars-april.

Forsinkelser i planen har ledet til at enkelte funksjoner og krav ikke ble innfridd. Videre ledet det til at ønsket funksjonalitet, eksempelvis kundeportal, ikke ble utforsket.

6. DISKUSJON

6.1. Sluttprodukt

Som diskutert i kapittel 5.3 har løsningen oppnådd delmålene som ble satt i kapittel 1. Løsningen fungerer også som et godt svar på problemstillingen, da løsningen tilgjengeliggjør bedriftsstatistikk og bruksdata over to forskjellige grensesnitt. Dette muliggjør bruk av dataene i videreutviklinger av produktet og andre applikasjoner i oppdragsgivers systemer.

Det er som nevnt enkelte krav som ikke ble møtt, samt at den originale løsningsideen ikke ble fullstendig implementert grunnet tidsmangel. Løsningen skulle ideelt inneholdt en kundeportal for henting av egen bedrifts bruksdata. Dette ble nedprioritert ettersom det ikke lenger var et krav, men ønskelig dersom det var mulig innenfor tidsrammen.

Prosjektet startet i januar med design av skisser av brukergrensesnittet med HTML, og implementasjon av dette etter valg av rammeverk som diskutert i kapittel 3. Arbeidet med programmeringsgrensesnittet ble ikke startet før månedsskiftet mars-april, grunnet forsinkelser fra oppdragsgivers side. Dette er ettersom kritiske tjenester, som database og levering av data, ikke var implementert før den tid.

Videre var det satt krav til at programmeringsgrensesnittet skulle være frittstående og enkelt å ta i bruk av andre tjenester. Med andre ord skulle endepunktene være generiske. De første iterasjonene av programmeringsgrensesnittet tilbydde endepunkt som var høyst implementasjonsspesifikk mot klientsideapplikasjonen. Ettersom dette ikke møtte kravet for programmeringsgrensesnittet ble det nødvendig å utvikle nye endepunkter for generelle bruksområder.

Som nevnt skulle løsningen ha god testdekning, noe som ble delvis nådd i henhold til initiell løsningsidé. I startfasen av prosjektet var det et forventet behov for integrasjons- og ende-til-ende-testing. Dette viste seg å ikke være nødvendig ved sluttfasen. Årsaken til dette er at modulene i prosjektet samhandler svært lite med hverandre, og følger et strengt ansvarsskille. Dersom løsningen skulle innehatt en faktureringstjeneste eller en kundeportal ville det vært aktuelt med integrasjonstesting, ettersom dette hadde ledet til økt samhandling mellom komponenter. Videre ble ende-til-ende testing heller ikke nødvendig, da de reelle dataenes opprinnelsessted er ute av prosjektets rekkevidde. Disse faktorene gjør at enhetstesting av kritiske moduler er nok for oppgavens omfang og kontekst. Til tross for disse manglene, tyder oppdragsgivers sluttevaluering på at det leverte produktet er av høy kvalitet og et solid fundament å bygge videre på.

6.2. Prosess

I prosjektet har det vært tydelig til enhver tid hvilke oppgaver som er relevante, grunnet Scrum. Ved sprintmøtene har gruppen kommet til enighet med oppdragsgiver om hva som skal fokuseres på i en gitt periode. Til tross for god planlegging har det vært tilfelle ved flere sprintavslutninger at oppgaver må overføres til påfølgende periode. Dette kan være resultat av at gruppen har hatt vanskelig for å estimere omfanget av oppgaver, noe timeføring viser tydelig med stor variasjon i antall timer per sprintperiode. Et annet negativt aspekt ved bruken av Scrum er den overbærende posisjonen til sprintmøtene. All kommunikasjon mellom gruppen og oppdragsgiver har hovedsakelig tatt sted under disse møtene. Dette resulterte til tider i at mindre problemstillinger og spørsmål ble til tider utsatt til neste sprintmøte, ovenfor å ta det opp fortløpende.

Til tross for de negative aspektene, har Scrum i all hovedsak hatt positiv innvirkning på prosessen. Ved å følge sprintformatet har endringer i omfanget av oppgaven, samt nye ønsker fra oppdragsgiver, vært enklere å tilpasse seg til. Dette har ledet til høy fleksibilitet både for gruppen og oppdragsgiver. De nevnte endringene hadde antakeligvis hatt større konsekvenser dersom prosjektet hadde fulgt en lineær og sekvensiell utviklingsmetodikk, eksempelvis fossefallsmetoden. Dette er grunnet at med en slik metodikk ville man jobbet mot klare mål satt innledningsvis i prosjektet.

Videre ble det tydelig at arbeidsutbytte av skriveøkter sank betraktelig med økt sesjonslengde. Den initielle strategien var å sette av én til to dager i uken for dokumentasjon. Dette førte til lange dager med skriving, og det ble fort åpenbart at lengre skriveøkter ga redusert effektivitet. Gruppen ble oppmerksom på dette, og endret strategi til daglige skriveøkter med kortere intervall. Denne endringen ga langt større arbeidsutbytte, sammenlignet med de lengre sesjonene.

6.3. Konsekvenser

Som beskrevet i kapittel 6.2 ble det meste av kommunikasjonen med oppdragsgiver gjort under sprintmøter. Som følge av oppgavens frihet har gruppen forsøkt å løse de fleste problemer som har oppstått på egenhånd, uten å konsultere oppdragsgiver, for så å vise den foreslåtte løsningen ved sprintmøter. Denne arbeidsflyten fungerte godt tidlig i løpet, men med begrenset effekt senere i løpet. Årsaken til dette var at enkelte avgjørelser kunne være lite optimale eller langt fra oppdragsgivers visjon, noe som tydet at kommunikasjonen fra gruppen til oppdragsgiver har vært

mangelfull gjennom prosjektet. Dette ble forsøkt løst ved å benytte Slack for kommunikasjon med oppdragsgiver utover sprintmøtene, noe som fungerte godt.

En konsekvens av manglende ressurser fra oppdragsgivers side var sen bestemmelse av databasetype, samt tabellstruktur i databasen. Ettersom prosjektet avhenger av disse for å kunne utføre uthenting av data som skal behandles og fremvises, var det ikke mulig å starte på serversidefunksjonaliteten før dette var fastsatt. Fram til slutten av mars var det forstått at databasen skulle være en dokumentbasert database, men dette ble endret i slutten av mars. Følgelig startet ikke utvikling av serversidefunksjonalitet før denne endringen var endelig.

Forsinket start på serversideutvikling resulterte i at all planlagt funksjonalitet ikke ble implementert. Synkronisering av sanntidsdata med SignalR og EventHub er blant funksjonaliteten som ikke har blitt ferdigstilt. Resultatet av dette er at ny data kan bare hentes ved å laste inn siden på nytt, noe som kan forstyrre en økt og redusere brukeropplevelsen.

En overbærende konsekvens av forsinket oppstart på serversideutviklingen er tap av kalendertid. Serversiden ble startet sent i månedsskiftet mars-april, noe som tillot én måned med utvikling på denne delen av produktet, samt kobling mot brukergrensesnittet. Dette gjør at en økning i timebruk ikke nødvendigvis ville resultert i flere implementerte funksjoner, derimot kunne tidligere oppstart ha resultert i innfrielse av alle krav.

6.4. Forbedringer

De største forbedringsfaktorene som er blitt identifisert er effektivt samarbeid og kommunikasjon. Noe som leder til de følgende fire forbedringene;

Dersom prosjektet skulle gjennomføres på nytt, med samme tidsspenn, er det klart at tidsestimering og overordnet plan bør endres. Oppgaven hadde hatt større mulighet for ferdigstilling dersom gruppen kunne startet med serversiden i starten av februar, i stedet for slutten av mars. Dette hadde vært en mulighet, dersom gruppen hadde satt opp egen database med testdata og jobbet etter den. Et mulig problem for denne fremgangsmetoden er mengden informasjon fra oppdragsgiver om databasemodell, og endringer som skjedde i perioden mellom februar og mars, eksempelvis endringer av tabellmodell og valg av database.

En annen mulig forbedring er ajourhold av gruppemedlemmer. Det ble forsøkt å holde ukentlige stand-up møter mellom gruppemedlemmene, for å holde alle oppdatert på status og plan for uken. Disse møtene falt vekk i startfasen, da det ikke var store endringer fra uke til uke, og ble ikke

gjeninnført senere i prosjektløpet. I etterkant ble nødvendigheten av slike møter tydeligere, for å sikre kjennskap til ukens agenda innad i gruppen.

Videre burde gruppens kommunikasjon med oppdragsgiver være mer aktiv. Som nevnt tidligere, samt i oppdragsgivers sluttevaluering av prosjektet, var gruppens kommunikasjon utover fastsatte sprintmøter manglende. Gruppen forsøkte å motta status angående implementasjon av oppdragsgivers del av prosjektet, men svar var manglende. Når dette var et faktum, burde gruppen kontaktet oppdragsgiver for å ordne omprioriteringer av ansvarshavers arbeidsoppgaver. Dette ble senere gjort etter respons ved et sprintmøte, men for sent for å påvirke prosjektets gang.

Som en siste forbedring burde gruppen ha sikret generell kompetanse i prosjektets teknologistack. Ansvaret for å lære seg verktøyene og teknologiene falt på enkeltindividene selv i gruppen. I konteksten av prosjektet kunne antakeligvis flere produktive timer bli dedikert til utvikling, dersom språk og rammeverk var kjent av alle gruppemedlemmer ved utviklingsstart. Dette kunne eksempelvis bli gjort ved å delta på kurs om teknologiene før oppstart.

7. KONKLUSJON OG VIDERE ARBEID

7.1. Konklusjon

Hovedmålet for prosjektet var å utvikle en applikasjon som leverer statistikk på tvers av produktene til oppdragsgiver. Det utviklede systemet tilbyr et programmeringsgrensesnitt som tilgjengeliggjør statistikk fra produktene gjennom generiske endepunkter. Dette programmeringsgrensesnittet blir konsumert av et brukergrensesnitt som, gjennom respektive visninger, viser relevant statistikk for produktene.

Prosjektgruppen mener at den foreslåtte løsningen kan gi oppdragsgiver et bedre innblikk i sine produkter, samt forenkle prosessen ved å automatisere de manuelle internrutinene som fortsatt er i bruk. Prosjektgruppen mener også at det utviklede produktet er en god kandidat for videre utvikling og fremtidig bruk, da de mest sentrale funksjonene er ferdigstilt og testet i et produksjonsmiljø.

Dersom produktet blir sett i lys av problemstillingen,
"Hvordan kan man utvikle et system som tilgjengeliggjør bedriftsstatistikk og bruksdata?", så ansees den endelige løsningen å stå som et godt løsningsforslag, samt at målene som er satt har blitt nådd i høy grad. Til tross for at enkelte krav ikke ble møtt, har oppdragsgiver gitt svært positiv tilbakemelding på det leverte produktet. Dette tyder på at den kvalitative delen av prosjektet er opprettholdt. Tilbakemeldingen tyder også på at oppdragsgivers ønsker og interesser har blitt ivaretatt gjennom prosessen.

7.2. Videre arbeid

Den foreslåtte løsningen har oppnådd de overbærende målene som var satt for prosjektet, men det var enkelte krav som ikke ble oppfylt. Følgelig bør videre arbeid forsøke å innfri de resterende kravene. Videre arbeid kan også ta basis i mulighetene skapt av det utviklede produktet.

Det viktigste kravet som ikke ble innfridd innen prosjektperioden var implementering av synkronisert sanntidsdata. Dersom sanntidsdata blir synkronisert kan applikasjonen brukes uten manuelle oppdateringer av siden. Med slik synkronisering kan applikasjonen for eksempel stå som en direktesending på en TV, og kontinuerlig gi tilbakemelding om vekst og muligheter.

Et krav som burde innfris dersom behovet skulle oppstå ved videreutvikling, er skriving av integrasjons- og ende-til-ende-tester. Dette kan bli nødvendig dersom applikasjonen øker i kompleksitet eller blir videreutviklet til å inneholde en kundeportal.

Videre burde de gjenstående produktsidene i brukergrensesnittet, som nå viser testdata, bli implementert. Disse sidene avhenger av datapunkter som ikke har blitt integrert i tabellene i databasen enda. Det utviklede produktet har tilrettelagt for at implementasjonen av de nye datapunktene krever lite endring på serversiden. Videre kan det også implementeres flere moduler, som for eksempel fremstiller trender og vekst basert på gitt statistikk. Landingssiden kan inneholde moduler som viser et sammendrag av statistikk for hvert produkt, og eventuelt sammenligner de mot hverandre for å se hvilket produkt som gir best avkastning.

Utvikling av kundeportal er en god kandidat for videre arbeid, ettersom dette var ønsket av oppdragsgiver. En slik kundeportal vil gi klienter tilgang til statistikk på lik måte som oppdragsgiver, men begrenset til sin bedrift.

Videre er et annet alternativ for videreutvikling en automatisert faktureringstjeneste. Denne tjenesten vil konsumere programmeringsgrensesnittet for å hente ut informasjon om kunders tjenestebruk. Dette kan da brukes som faktureringsgrunnlag av tjenesten. En slik tjeneste vil videre bidra til å fjerne manuelle rutiner hos oppdragsgiver.

Avslutningsvis, dersom lignende oppgaver skal foretas i framtiden, ønsker gruppen å anbefale mer aktiv og kontinuerlig kommunikasjon med oppdragsgiver. Det er også sterkt anbefalt å starte arbeidet med alle deler av løsningen tidligere. Ideelt bør alle nødvendige ressurser fra oppdragsgiver være klar ved prosjektstart.

8. LITTERATURLISTE

Digital.ai (2021) *15th Annual State Of Agile Report*, s. 13. Tilgjengelig fra: https://digital.ai/resource-center/analyst-reports/state-of-agile-report (Hentet: 29. april 2022).

Fielding, R.T. (2000) *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral. University of California. Tilgjengelig fra: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

GitHub (uten år) *Dependency graph*, *Network Dependents*. Tilgjengelig fra: https://github.com/vercel/next.js/network/dependents?package_id=UGFja2FnZS0xNDIzMDMwOA%3D%3D (Hentet: 9. mai 2022).

Høgstrand, J. (2019) Hva er Kanban?, *Prosjektbloggen*. Tilgjengelig fra: https://www.prosjektbloggen.no/hva-er-kanban (Hentet: 29. april 2022).

IBM Cloud Education (2021) *Containerization*, *Containerization Explained*. Tilgjengelig fra: https://www.ibm.com/cloud/learn/containerization (Hentet: 24. april 2022).

Kiefer, R. (2017) TimescaleDB vs. PostgreSQL for time-series, *Timescale Blog*. Tilgjengelig fra: https://www.timescale.com/blog/timescaledb-vs-6a696248104e/ (Hentet: 3. mai 2022).

Krasner, G.E. og Pope, S.T. (1988) A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System, *Journal of object oriented programming*, 1(3), s. 26–49. Tilgjengelig fra: https://www.researchgate.net/profile/Stephen-Pope-2/publication/239452280_A_Description_of_the_Model-View-Controller_User_Interface_Paradigm_in_the_Smalltalk80_System/links/02e7e5297a33b7b70b000000/A-Description-of-the-Model-View-Controller-User-Interface-Paradigm-in-the-Smalltalk80-System.pdf.

Krausest (2022) *Results for js web frameworks benchmark*, *Interactive Results*. Tilgjengelig fra: https://krausest.github.io/js-framework-benchmark/2022/table_chrome_99.0.4844.51.html (Hentet: 9. mars 2022).

Malt, U. og Grønmo, S. (2020) Likert-skala, *Store norske leksikon*. Tilgjengelig fra: http://snl.no/Likert-skala (Hentet: 5. mai 2022).

Microsoft (uten år) *Build client web apps with C#*, *Build client web apps with C#*. Tilgjengelig fra: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor (Hentet: 25. februar 2022).

Nasjonal kommunikasjonsmyndighet (2021) *Bredbåndsdekning 2021*, *Ekomstatistikken*. Tilgjengelig fra: https://ekomstatistikken.nkom.no/#/article/dekning_regionalt2021#virk (Hentet: 20. april 2022).

Next.js (uten år) *Showcase*. Tilgjengelig fra: https://nextjs.org/showcase (Hentet: 9. mars 2022).

Nielsen, J. (2020) 10 Usability Heuristics for User Interface Design, *Nielsen Norman Group*. Tilgjengelig fra: https://www.nngroup.com/articles/ten-usability-heuristics/ (Hentet: 19 May 2022).

NPM (uten år) *About the public npm registry*. Tilgjengelig fra: https://docs.npmjs.com/about-the-public-npm-registry/ (Hentet: 25. februar 2022).

Price, E. (2022) Microservices CI/CD pipeline on Kubernetes, *Azure Architecture Center*. Tilgjengelig fra: https://docs.microsoft.com/en-us/azure/architecture/microservices/ci-cd-kubernetes (Hentet: 18. mai 2022).

Proff (uten år) *Zdata AS - Regnskap*. Tilgjengelig fra: https://www.proff.no/regnskap/zdata-as/laksev%C3%A5g/dataprogramvare-og-utvikling/IF9IIRQ009O/ (Hentet: 5. mai 2022).

Schwaber, K. and Sutherland, J. (2020) *Scrum Guide | Scrum Guides*, *Scrum Guide*. Tilgjengelig fra: https://scrumguides.org/scrum-guide.html (Hentet: 9. mars 2022).

Scrum.org (uten år) *What is Scrum?*, *What is Scrum?* Tilgjengelig fra: https://www.scrum.org/resources/what-is-scrum (Hentet: 25. februar 2022).

Stack Overflow (2021) *Stack Overflow Developer Survey 2021*, *Stack Overflow*. Tilgjengelig fra: https://insights.stackoverflow.com/survey/2021 (Hentet: 25. februar 2022).

Vats, H. (2022) How to lazy load images in html with pure Javascript?, *How to lazy load images in html with pure Javascript?*. Tilgjengelig fra: https://dev.to/harshvats2000/how-to-lazy-load-images-in-html-with-pure-javascript-13m7 (Hentet: 20. april 2022).

ZData AS (uten år) Om ZData, *Om ZData*. Tilgjengelig fra: http://www.zdata.no/om-zdata/ (Hentet: 18. mai 2022).

9. Vedlegg

Alle vedlegg er tilgjengelig som eksterne dokumenter.