UNIVERSITETET I AGDER

Forside

IS-304: 2021

Tittel:

Emnekode	IS-304
Emnenavn	Bacheloroppgave i informasjonssystemer
Emneansvarlig	Hallgeir Nilsen
Veileder	Janis Gailis
Oppdragsgiver	Webstep (Avd KRS)

Studenter:

Fornavn	**Etternavn**
Gustav	Eikaas
Daniel	Lindalen
Tomas	Ryen
Mantas	Karulaitis

Jeg/vi bekrefter at vi ikke siterer eller på annen måte bruker andres arbeider uten at dette er oppgitt, og at alle referanser er oppgitt i litteraturlisten.	JA _X_	NEI__ _
Kan besvarelsen brukes til undervisningsformål?	JA X	NEI__
Vi bekrefter at alle i gruppa har bidratt til besvarelsen	JA X	NEI__

Forord

Denne rapporten er skrevet siste semester ved bachelorutdanningen innen IT og Informasjonssystemer ved Universitetet i Agder. I forbindelse med denne rapporten er det flere personer vi ønsker å takke.

Vi vil starte med å takke gruppens veileder Janis Gailis som har hjulpet oss og besvart spørsmål gjennom veiledningsmøter. Vi vil også takke Hallgeir Nilsen for hans gode jobb som foreleser i IS-304.

Videre vil vi rette en stor takk til vår oppdragsgiver Webstep. Vi vil spesifikt navngi Trond Kjetil Nilsen som har bistått med kodegjennomganger og teknisk veiledning gjennom prosjektet. Og ikke minst vil vi takke produkteier Anders Høibakk som gjennom ukentlige møter har hatt en sentral rolle i utviklingen av produktet og veiledningen av vår bacheloroppgave. Denne rapporten hadde ikke blitt til uten deres hjelp. Tusen takk!

Figurliste

1 Introduksjon

Bacheloroppgaven i IT og Informasjonssystemer ved Universitet i Agder innebærer at en gruppe studenter planlegger, gjennomfører og dokumenterer et utviklingsprosjekt hos en bedrift.

Den valgte bedriften er Webstep. Webstep er et teknologi- og rådgivningsselskap som opererer innenfor ekspertområder som arkitektur, integrasjon, utvikling og UX/UI design. Selskapet består av 9 avdelinger i Norge og Sverige. Dette prosjektet ble utført i samarbeid med avdelingen i Kristiansand.

Webstep ga oss som oppdrag å utvikle en web-applikasjon der selgere kan utføre deres arbeidsoppgaver. Vi ble gitt frie tøyler i utformingen av applikasjonen og ble oppfordret til å tenke nytt. Gjennom prosjektet hadde vi tett kontakt med produkteieren for å sikre at utviklingen gikk riktig retning.

I utførelsen av prosjektet tok vi i bruk ferdigheter vi hadde skaffet oss fra tidligere fag. Som del av studiet har vi utført flere programmeringsprosjekter med agil metodikk. Vi valgte å ta i bruk teknologiene vi mente passet best for prosjektet. Dette gjorde at vi måtte lære oss TypeScript, React, C# og .NET Core.

I denne rapporten beskriver vi prosjektet som ble utført våren 2021. Dette innebærer å beskrive dagens løsning, analysen vi utførte, planene vi skapte for prosjektstyring, prosjektgjennomførelsen og til slutt vår refleksjon på hvordan det gikk.

2 Prosjektet

Prosjektet går ut på å lage en web applikasjon der selgere i Webstep kan utføre sine arbeidsoppgaver. Applikasjonen skal være en SPA(Single Page Application) som støtter samarbeid i sanntid. Hensikten med overgangen til webapplikasjon er å gjøre det mulig for Webstep å integrere dette mot regnskap-, HRM(Human resource management)- og CRM-system(Customer relationship management).

2.1 Dagens løsning

Vårt prosjekt går ut på å utvikle en applikasjon som lar selgere i Webstep utføre diverse arbeidsoppgaver.

De fleste konsulentselskap har to typer ansatte, selgere og konsulenter. Selgere, som navnet tilsier, selger selskapets tjenester til andre bedrifter og organisasjoner. Konsulenter blir tildelt kontrakter av selgerne. Løsningen vi skal forbedre er et administrativt verktøy brukt av selgere. I dette verktøyet kan selgere holde oversikt på hvilke kontrakter de har, mulige kontrakter de kan få og konsulentenes kapasitet.

Dagens løsning er laget i Google Sheets. Den har 5 faner med hvert sitt arbeidsområde. Det er tatt i bruk avanserte formler for å automatisere beregninger. En styrke ved å ha en slik løsning i Google Sheets er at flere kan jobbe i dokumentet samtidig. Produkteier ønsket en høyere grad av automasjon i prosjektet, samt et høyere fokus på smidig og simpelt design.

Problemene selgerne har med løsningen er at den er ikke intuitiv, vanskelig å integrere med andre systemer, ikke fleksibel og lite brukervennlig. Vår oppgave er å lage en webapplikasjon som mangler svakhetene men beholder styrkene. Dette innebærere å implementere støtte for samarbeid i sanntid, automatisere og skjule alle operasjoner som ikke er direkte relevant for brukeren og lage en nettside der alle arbeidsoppgaver kan utføres. Brukergrensesnittet bør skjule unødvendig data, vise informasjon på et oversiktlig vis og være intuitiv å bruke. En annen svakhet er at regnearkløsninger er svært begrenset i designmuligheter.

I de neste kapitlene beskriver vi hver side av løsningen som selgerne bruker. I vår beskrivelse inkluderer vi hva som vises, hvilke arbeidsoppgaver som kan bli utført og hvordan vi skal forbedre den.

2.1.1 Prospekter

Betydningen av et prospekt i denne sammenhengen er en mulig kontrakt. Selgere i Webstep prøver regelmessig å selge sine tjenester til andre bedrifter. For å holde kontroll på alle tilbudene oppretter de prospekter. Et prospekt gis en sannsynlighet på 10, 30 eller 70% utifra hvor sannsynlig det er at de får oppdraget.

Betydningen av verdiene er:
10% = Vært i kontakt med - ser ut som behov
30% = Avklart behov, sendt over tilbud/CV
70% = muntlig aksept

Hensikten med denne siden er å holde styr på potensielle kontrakter hos kunder. For hvert prospekt føres det kundenavn, prosjektnavn, sannsynlighet for at det blir kontrakt og estimert arbeidsomfang som kreves på kontrakten.

Det er en rekke irritasjonsmomenter med denne siden som den er i dag. Det største problemet er at regnearket er fullpakket med informasjon. Det gjør det vanskelig å få et raskt overblikk for brukeren.

Kravene fra produkteier var å lage en oversiktlig og intuitiv løsning. Vi var nokså frie til å velge design selv, men han så for seg et slags Gantt-chart. Produkteier laget en grov sketch for å vise hans visjon for designet.

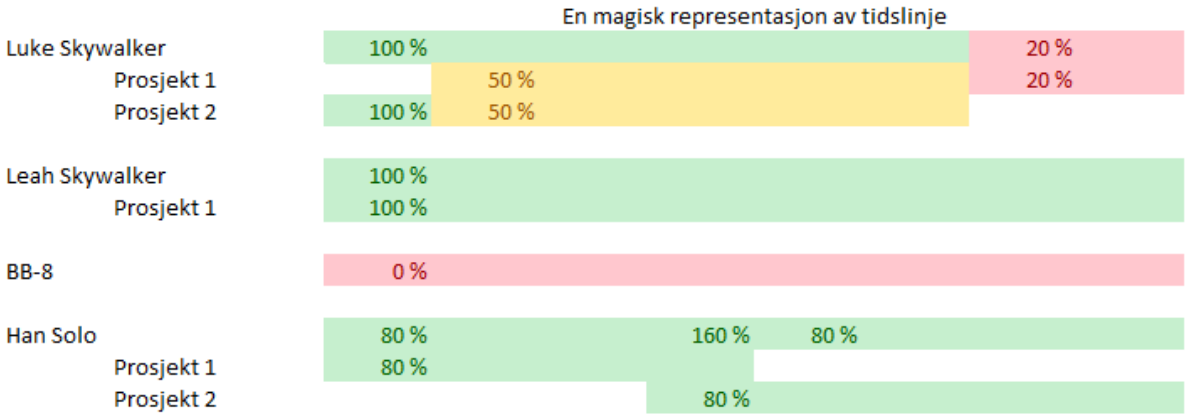


fig. 1 Initiell sketch for prospekter.

2.1.2 Belegg

Betydningen av belegg i denne sammenhengen er fakturerbar tid en konsulent jobber hos en kunde. I løsningen vi beskriver planlegger selgerne i Webstep kontraktsfestet tid for konsulentene. Det er derfor viktig at selgerne har oversikt over permisjon og sykdom hos deres konsulenter.

Hensikten med denne siden er å holde styr på hvilke konsulenter som jobber på hvilke kontrakter til enhver tid. Det er viktig for selgerne å balansere arbeidsmengden mellom alle konsulentene. Her kan selgerne føre belegg og fravær for konsulenter. De kan også naturligvis legge til nye kontrakter.

Problemene med denne siden er lik som for prospekt siden. Det er vanskelig å få oversikt over konsulentenes kontrakter. Produkteier synes også det var unødvendig å ha sider for både belegg og ledige ressurser, da begge kunne blitt slått sammen til èn side. Dette ville gjort det mye lettere å finne konsulentene med høyest ledighet.

Planen for denne siden var å først lage et design for prospekter, deretter kopiere og tilpasse den for belegg. Produkteier gjorde oss oppmerksomme på at vi kunne slå sammen ledige ressurser siden med belegg. Grunnen til dette er at hvis du reverserer belegg får du ledig tid. Hvis en konsulent har 40% belegg er han altså 60% ledig.

2.1.3 Ledige ressurser

Hensikten med denne siden er å få en oversikt over ledigheten til konsulentene i firmaet. Dette er viktig for selgere å vite, da det er de som tildeler oppdrag til konsulenter.

Problemet med denne siden er at den ikke trenger å være isolert fra belegg-siden. Dette er fordi sum dager på kontrakt er det inverse av antall ledige dager. Vår plan for forbedring var å

kombinere belegg-siden og siden for ledige ressurser. Målet var å beholde verdien til begge sidene etter de er kombinert.

2.1.4 Nøkkeltall

Nøkkeltall er en samling av grunndata og beregnede verdier som skal gi et generelt inntrykk av avdelingens tilstand. Kategoriene i nøkkeltall er omsetning, resultat, timepris, antall ansatte, forecast og på kontrakt. Fra nøkkeltall vil en selger kunne evaluere kapasitet, lønnsomhet og få et generelt innblikk i hvordan avdelingen ligger ann. Blant tallene så er "på kontrakt" og "forecast" to av de viktigste.

Tallet "på kontrakt" viser andel tid konsulentene har på kontrakt i forhold til deres kapasitet. 90% betyr altså at av tiden konsulentene har tilgjengelig så er 90% dedikert til en kontrakt. Denne verdien spiller en viktig rolle i evalueringen av avdelingens tilstand.

Tallet "forecast" er hvor stor andel tid de tror konsulentene vil ha i fremtiden. For å beregne verdien brukes summen av faktisk antall dager på kontrakt og antall dager i prospekter ganget med deres sannsynlighet.

Hensikten med nøkkeltall-siden er å gi et raskt overblikk over hele avdelingen. Siden gjør det mulig for selgere å kunne finne ut av avdelingens nåværende og fremtidige tilstand raskt. Dette er viktig for å kunne legge merke til og reagere på mulige problemer tidlig.

Problemer med denne siden er blant annet at tallene kan oppleves som vanskelig å forstå for nye selgere. Det er også muligheter til å erstatte tabeller med grafer. Utenom dette så er tidsrommet som vises begrenset til tre måneder for noen tall uten særlig grunn.

Planen for denne siden var å finne bedre måter å vise samme informasjon. Dette innebærer å utforske forskjellige kombinasjoner av grafer og tabell for fremvisningen. Tilbakemelding fra produkteier vil spille en sentral rolle i utførelsen av denne planen. Uten tilbakemelding vil vi ikke kunne evaluere subjektive krav som f.eks å vise informasjon på et bedre vis.

2.1.5 Grunndata

Grunndata i denne sammenhengen er økonomisk data hentet direkte fra økonomisystemet til bedriften. Grunndata blir brukt som basis for å regne ut andre celler i løsningen.

Hensikten med denne siden er å holde styr på og føre grunnleggende data i systemet. Grunndata består hovedsaklig av de samme tallene som nøkkeltall, med noen ekstra felt for avvik i omsetning, resultat og konsulent timepris. I tillegg er tabellen for ansatte utvidet. Måltall blir skrevet inn ved starten av året, mens de faktiske tallene skrives inn etterhvert når tallene fra økonomiavdelingen er inne.

Problemet med denne siden er at grunndata er åpen for redigering. De andre sidene trenger kun å lese disse tallene, så evnen til å redigere dem virker unødvendig. Tallene må føres for

hånd, som kan forårsake feilføringer og videre skape en feil bilde av avdelingens tilstand. Ideelt sett vil grunndata bli automatisk hentet rett fra økonomiavdelingen.

Planen for denne siden var å erstatte den med en database i backend. Dette legger opp til muligheter for fremtidig integrering med økonomisystemet. Det må fremdeles være mulig å føre grunndata så lenge økonomisystemet ikke er integrert. I enighet med produkteier nedprioriterte vi manuell føring av grunndata inntil videre.

2.2 Brukerhistorier

Ved oppstart ble vi enige med produkteier at vi utsetter systemutvikling til vi forstår systemet vi skal erstatte. Vår valgte metode for å skaffe denne forståelsen var oppretting av brukerhistorier. Vi lagde først brukerhistorier til vår beste evne ved å studere systemet og formulere hvilke funksjonaliteter som er innebygd. Historiene ble dannet uten hjelp fra produkteier og tok utgangspunkt i perspektivet til en selger.

Brukerhistoriene består av en beskrivelse og implementeringskrav. Beskrivelsen følger formatet *"Som en x, så vil jeg y, slik at jeg kan z"*. Implementeringskravene består av krav til lagret data, funksjonalitet og brukergrensesnitt. Hensikten med implementeringskravene er å danne tanker rundt hvordan historien kan implementeres. Vi regnet med å måtte skrive kravene på nytt under utviklingen.

Neste gang vi møtte med produkteier, gikk vi gjennom historiene sammen. Da pekte produkteier ut feil og misforståelser ved historiene. Tilbakemeldingene hjalp oss forstå systemet bedre siden det rettet ut feil i vår mentale modell av hvordan systemet fungerer. Over de neste dagene rettet vi opp i historiene etter produkteierens tilbakemelding.

Når det var gjort, møttes vi igjen for å se over historiene. Sammen med produkteier prioriterte vi historiene og gjorde de siste få nødvendige rettelsene. Vi la inn historiene i Azure DevOps. Dette ble gjort siden Azure DevOps lar oss gruppere oppgaver og systemkrav under brukerhistoriene i hver sprint.

Vi lagde totalt 34 brukerhistorier. Noen av brukerhistoriene gjaldt nye funksjonaliteter foreslått av oss, som da ble godkjent av produkteier. Denne prosessen hjalp oss forstå hvilke funksjonaliteter som skaper verdi for selgeren. Den hjalp oss også forstå hvordan funksjonalitetene bør fordeles mellom frontend og backend i vår applikasjon. Vi inkluderte noen brukerhistorier i appendix.

2.3 Design

Vi bestemte oss for å ikke ha en formell prosess for design. Kort, betyr dette at vi utvikler funksjonalitet først, for å få tilbakemelding på hvordan det bør se ut. Dette passer med React. React-komponenter er enkle å omplassere i systemet og gi definerte stiler. Argumentet for dette valget er at dersom vi må definere utseende før utvikling ville det ha bremset oss ned betydelig.

Sidene for ledige ressurser, belegg og prospekter i den eksisterende løsningen viser informasjon over tid gruppert etter temaer. Derfor bestemte vi at sidene skulle ha like design

og fungere på omtrent samme vis. Argumentet er at det vil spare oss tid og samtidig gi en konsistent brukeropplevelse. Produkteieren anbefalte å følge et design i likhet med det som kalles Gantt-charts.

Siden for nøkkeltall har ingen fellestrekk med resten av sidene. Gjennom samtale med produkteier fikk vi definert noen krav til designet. Det må være åpenbart hva tallene betyr. Det må være lett å få et raskt overblikk. Alt bør være synlig uten interaksjon eller scrolling.

Gjennom utviklingen gikk vi gjennom mange designs. For hver sprint som gikk fikk vi et klarere syn på hvordan sluttproduktet bør se ut. Noen av våre tidlige designs er i appendix.

2.4 Krav og Mål

I dette kapittelet går vi gjennom krav og mål som ble dannet i oppstartsfasen før utviklingen startet.

Vi hadde ingen krav før utviklingen starten utenom brukerhistoriene. Brukerhistoriene oppsummerer arbeidsoppgavene selgerne må kunne utføre samt noen forslag til ny funksjonalitet. Produkteier ønsket å definere videre krav gjennom diskusjoner og demoer hver sprint.

Vi ble enige med produkteier om at målet for prosjektet var å utvikle grunnfunksjonaliteten i hver side. Hensikten med dette målet var å holde seg realistisk mens man fokuserer på det essensielle. Vi ventet med å sette flere mål da omfanget og kompleksiteten til ønsket sluttprodukt var ukjent for oss og produkteier.

3 Sentrale avgjørelser

Denne delen av rapporten beskriver sentrale avgjørelser vi har tatt før og under utviklingen. Vi kommer til å ta for oss prosjektstyring først, for å gå gjennom verktøy og språk vi har brukt.

3.1 Prosjektstyring

Dette kapitlet beskriver diverse regler, tiltak og aktiviteter som ble valgt for å styre prosjektet.

3.1.1 Metodikk

Vi har brukt et utvalg av aktiviteter og prinsipper fra Scrum i utviklingen av dette produktet. Målet med våre valg er å sikre at vi jobber iterativt, endrer fremgangsmåte ved behov, starter på nytt ved behov, kommuniserer tett med produkteier og jobber med små definerte oppgaver.

En slik metodikk lar oss raskt teste idéer innen design og funksjonalitet. Dette er viktig for oss siden vi vil sikre at vi utvikler det produkteieren ønsker. Dette reduserer også sløst innsats ved å la “dårlige idéer dø raskt”.

Alle gruppemedlemmene hadde erfaring med å følge agil metodikk. Gjennom studiet har vi mye erfaring med SCRUM i tidligere programmeringsprosjekter.

3.1.2 Roller

Vi har hatt rollene prosjektleder, scrum master og utvikler i dette prosjektet. Alle medlemmer har vært utviklere. Gustav har hatt rollen som prosjektleder og Daniel har vært Scrum master. Ansvarene til rollene overlapper noe men det skapte ingen konflikter. Områdene der ansvarene overlapper ble regnet som delt ansvar.

Scrum Master ansvarsområder

- Administrere og holde daily standups.
- Assistere produkteier med produkt backlog.
- Hjelpe å fjerne hindringer i utviklingen.
- Holde kontroll over scope på oppgavene.
- Løse interne konflikter.
- Passe på at tidsestimering blir utført
- Passe på at arbeid blir definert som oppgaver i Azure DevOps.
- Administrere og holde sprint planning.
- Prøve å sikre at utviklingsprosessen er agil.

Prosjektleder ansvarsområder

- Kommunisere med gruppemedlemmer
- Løse interne konflikter
- Kommunikasjon med produkteier og stakeholders.
- Identifisere problemer i prosjektet
- Ha oversikten i prosjektet mtp. tid, fremgang, problemer osv.
- Planlegging

Som utviklere hadde vi forskjellige arbeidsområder. Vi begynte først med 2 i backend og 2 i frontend men gikk over til 3 i frontend og 1 i backend. De neste listene beskriver hovedansvarene til hver utvikler.

Gustav (Backend)

- Backend

Daniel (Frontend)

- Belegg-siden
- Prospekt-siden

Mantas (Frontend)

- Nøkkeltall
- Innstillinger

Tomas (Frontend)

- Nøkkeltall
- Diverse arbeid på tvers av sidene

3.1.3 Sprinter

Vi valgte å jobbe i sprints. Våre sprints varte i én arbeidsuke, og noen ganger to uker. Hver sprint ble definert i Azure DevOps, et verktøy som tilbyr en oversikt over hvilke brukerhistorier og oppgaver som skal gjøres. Vi valgte å starte hver sprint på mandag, og avslutte med sprint review på fredag sammen med produkteier.

3.1.4 Daily Scrum

Gjennom hele prosjektet møttes alle i gruppen et fast tidspunkt hver ukedag. Møtene varte mellom 10 og 30 minutter. Agendaen besto av at Scrum master stilte tre spørsmål til hvert medlem.

Dette er spørsmålene:

1. Hva gjorde du igår?
2. Hva skal/vil du gjøre idag?
3. Hva er potensielle hinder for din fremgang?

Etter spørsmålene var det rom for diskusjon, teknisk hjelp og demo av hva som ble gjort dagen før.

3.1.5 Sprint Planning

Det første møtet ved start av ny sprint gikk til planlegging av sprinten. Dette innebærer å velge brukerhistorier, danne oppgaver, estimere omfang og tildele ansvar. Våre valg tok utgangspunkt i innholdet av forrige Sprint Review.

Oppretting av oppgaver ble gjort sammen med utvikleren som skulle ha ansvaret for brukerhistoriene.

Tidsestimering ble i starten av utviklingen regnet som utviklerens ansvar. Dette hadde begrenset suksess da ikke alle gjorde dette. Senere ble estimering gjort i plenum under planleggingen og oppdatert ved daglige møter.

Ansvar ble tildelt etter interesse, kompetanse og erfaring.

3.1.6 Sprint Review

Vi hadde formelle møter i slutten av hver sprint sammen med produkteier. Møtene hadde en rimelig fast agenda:

1. Demo av produktet
2. Oppsummering av fremgang og utfordringer
3. Feedback fra produkteier
4. Diskusjon av fremgang og status
5. Planlegging av neste sprint

Under demoen av produktet ble synlig fremgang vist. Etter dette sto produkteieren fritt til å se kode og annen funksjonalitet etter ønske.

Gjennom prosjektet ble det også satt opp styringsgruppemøter sammen med veileder og produkteier. I disse møtene presenterte fremgangen og eventuelle problemer vi har hatt. Mot slutten av slike møter ble veien videre diskutert i plenum.

3.1.7 Azure DevOps

Azure DevOps tilbyr verktøy for å utvikle iht. Scrum-rammeverket. Dette innebærer Scrum-hjelpemidler som backlog, taskboard og burndown-charts. Disse ble brukt under Sprint Planning, utviklingen av produktet og Sprint Review. Produkteier hadde også tilgang til dette.

I tillegg tilbyr DevOps repository for koden. Dette gir oss versjonskontroll og mulighet for arbeid i branches.

Vi bruker også DevOps sin innebygde Continuous Integration. Når flere utviklere jobber samtidig med forskjellige ting, og pusher sine endringer til en branch så kan kodebasen bli ustabil. Uten CI vet du ikke at en build feiler og må manuelt sjekke. Sannsynligheten av dette problemet vokser med størrelsen på utvikler teamet.

Continuous Integration lar deg sikre at du alltid har stabil kode etter en endring i kodebasen og varsler deg dersom en build eller testene feiler. Dette lar deg identifisere og løse problemer raskt.

3.2 Verktøy og språk

Vi har benyttet oss av flere forskjellige verktøy og språk gjennom prosjektet. I de neste kapitlene skal vi forklare hva de forskjellige verktøyene og språkene er, og forklare hvorfor vi har valgt å ta de i bruk.

Valgene våre rundt verktøy og språk ble påvirket av produkteier og TK(utvikler fra Webstep). Vi valgte verktøy og språk vi visste enten produkteier eller TK hadde kjennskap til. Dette ga oss muligheten til å motta veiledning fra dem. Mange av språkene og verktøyene er også relevante for det lokale jobbmarkedet.

3.2.1 .NET Core

Vi valgte å ta i bruk .NET Core (med C#) for utviklingen av vår backend. Dette ble gjort etter anbefaling av produkteier. Vi visste at denne teknologien ble mye brukt i jobbmarkedet i Kristiansand, noe som videre motiverte valget.

3.2.2 React

Vi brukte React i utviklingen av nettsiden. Det er ingen spesiell grunn til at vi foretrakk React over Angular, Vue eller Svelte. Vi visste at React hadde god dokumentasjon og stor

brukerbase. Utenom dette så visste vi at teknologien var kjent for utvikleren vi kunne kontakte i Webstep.

3.2.3 Typescript
Grunnen til at vi valgte å bruke dette språket fremfor Javascript var muligheten til å bruke statiske type-definisjoner. Fordelen med type-definisjoner er at det gjør koden enklere å vedlikeholde og reduserer bugs relatert til typer. Vi fikk inntrykk av at kombinasjonen av TypeScript og React fungerte godt.

3.2.4 REST
Vi brukte REST API siden det var enkelt å implementere. Vi vurderte også GraphQL men vi fikk inntrykk av det var en brattere læringskurve. REST er et rammeverk som er kjent for å være enkelt å implementere.

3.2.5 GraphQL
Hovedgrunnen til at vi etterhvert gikk over til GraphQL var at produkteieren foreslo det. GraphQL lot oss utføre spørringer mot API slik at vi mottok kun relevant informasjon. Dette reduserte datamengden som ble sendt og stress på serveren.

3.2.6 Apollo Client
Vi valgte å bruke Apollo Client for kommunikasjon med vår GraphQL API. Apollo gjør det enkelt å håndterte spørringer og state management relatert til spørringene. En av grunnene til at det er enkelt er fordi de tar i bruk Hooks for spørringer. Ved å bruke deres Hooks slapp vi ta hensyn til kompleksitetene som kan oppstå under spørringer. Apollo var rimelig greit å lære da verktøyet er godt dokumentert gjennom konsise avsnitt og illustrerende kodesnutter på deres nettside. Verktøyet har også støtte for datasynkronisering. Datasynkronisering spiller en stor rolle i støtte for sanntid samarbeid i vår applikasjon.

3.2.7 MS-SQL
Grunnen til at vi valgte MS-SQL var fordi vi planla å gå over til Azure SQL når produktet skulle deployeres i skyen. MS-SQL er nokså likt Azure SQL, noe vi antok ville forenkle prosessen.

3.2.8 EF Core
EF core lar deg skrive SQL queries ved bruk av LINQ. Den konverterer automatisk LINQ til SQL kode. Den sikrer deg også det siste innen beskyttelse mot SQL injection og andre farlige angrep. Vi regnet med at dette verktøyet kom til å spare oss en stor mengde koding og testing.

Eksempel på LINQ til SQL konvertering er
dbContext.Consultants.ToList() blir oversatt til *SELECT * FROM Consultants*

3.2.9 Discord
Hovedgrunnen til at vi valgte å bruke Discord var at alle i gruppen hadde mye erfaring med plattformen. Bruk av kanaler lot oss strukturere kommunikasjonen etter temaer som frontend, backend, bugs og notater.

3.2.10 Slack
Vi brukte Slack for å kommunisere med Anders (produkteier) og Trond Kjetil (utvikler) fra Webstep. Det ble opprettet en kanal gjennom Slack Connect. I denne kanalen kunne vi stille spørsmål, vise bilder og planlegge møter.

3.3 Kvalitetssikring

Vi regner at produktet har høy kvalitet dersom det oppnår følgende mål:

1. Oppfyller kravene for grunnfunksjonalitet
2. Støtter samarbeid i real-time
3. Fungerer i stor likhet med et realistisk scenario
4. Mangler svakhetene ved den eksisterende løsningen

De neste seksjonene beskriver aktivitetene og tiltakene vi har tatt for å sikre kvaliteten til koden, produktet og prosessen.

3.3.1 Kode
Vi sikret høy kodekvalitet gjennom støtte fra våre valgte verktøy og ved å følge kjente kode-prinsipper.

I backend-utviklingen forsøkte vi å følge god kodeskikk. Dette innebærer å følge prinsipper som SOLID og KISS. Samt ha fokus på høy “Cohesion”, lav “Coupling”, og lite kode duplikasjon. Vi hadde noen gjennomganger med produkteier hvor han så gjennom koden sammen med oss. Siden han er tidligere utvikler hadde han gode innspill. ReSharper er en utvidelse man kan installere i Visual Studio som gir deg gode tips og hint til hvordan å skrive bedre kode. Vi klarte å holde en god struktur og ryddig kode. Vi fikk også gode tips fra produkteier som også er/var utvikler.

Valget av TypeScript over JavaScript i frontend bidro til økt kodekvalitet. Vi valgte å bruke en “streng” versjon av TypeScript, som vil si at koden ikke kompilerer med mindre reglene definert av TypeScript følges. Reglene har som hensikt å hindre bugs og feil relatert til typer. Vi brukte typer på nesten alle variabler i kildekoden.

Vi regner kvaliteten til logikken i koden som en del av kodekvaliteten. Vi valgte å sikre denne kvaliteten gjennom testing. Backend delen av applikasjonen prøvde å bruke mange velutviklede rammeverk som mulig og derfor var det lite kode å teste. Vi brukte NUnit tester for all egen kode som inneholder en viss mengde logikk. Vi rådførte oss med produkteier som også er utvikler og han var enig i avgjørelsene rundt testing i backend.

I frontend brukte vi Jest, et testverktøy for React og Javascript. Verktøyet blir brukt til unit-testing av deler av logikken i nøkkeltall-siden. Vi rakk ikke legge til unit testing i logikken til de andre sidene.

3.3.2 Produkt
Vi definerte ikke produktkvalitet på tradisjonelt vis gjennom akseptkriterier, brukerhistorier og feedback fra brukere. Samarbeid med produkteier under utviklingen spilte en stor rolle i å definere produktkvalitet. Våre brukerhistorier fungerte som definisjoner på hva brukeren vil kunne se og gjøre i applikasjonen.

Kravene for funksjonalitet og design var under endring gjennom hele utviklingsprosessen. De overordnede kravene endret seg lite, men måten de skulle implementeres på ble revurdert ofte. Dette kunne bidra til at deler av fremgangen i en sprint måtte gjøres på nytt, men da som regel på et bedre vis. På denne måten sikret vi at produktet hadde høy kvalitet og stemte overens med produkteierens ønsker.

Prosessen for å utvikle en funksjonalitet kan oppsummeres i 5 steg:

1. Definere ønsket funksjonalitet, design og krav med produkteier.
2. Utvikle kode som utfører ønsket funksjonalitet etter beste evne
3. Få feedback fra produkteier
4. Rette utviklet funksjonalitet etter produkteiers ønsker
5. Gjenta til funksjonaliteten regnes som ferdig av produkteier.

Hvert møte med produkteier ble dokumentert gjennom referater. Referatene oppsummerte nye krav, ønsker om endringer, planen videre og tilbakemeldinger. Slike referater ble da brukt i planleggingen av neste sprint.

3.3.3 Prosess

Med prosesskvalitet mener vi kvaliteten til utviklingsprosessen. Denne seksjonen beskriver tiltakene vi tok for å sikre godt samarbeid, jevn fremgang og redusert social loafing.

For at utviklere skal kunne lese og forstå hverandres kode er det viktig at alle følger like regler. Derfor valgte vi tidlig å lage et dokument for kodestandard. Dette dokumentet inneholdt blant annet hvordan vi skulle navngi variabler, funksjoner, klasser og konstanter, og hvordan kommentarene skulle se ut.

En essensiell del av samarbeid i utviklingsprosessen er bruk av versjonskontrollsystem. Derfor lagde vi også et dokument med retningslinjer for bruk av Git. Denne inneholdt nyttige kommandoer og regler for hvordan vi skulle skrive en commit-message. I vår kommunikasjonsplattform, Discord, la vi til steg-for-steg veiledning for mer kritiske handlinger som forberedelser før en pull request. Dette ble gjort for å effektivt utnytte versjonskontrollsystemet og hindre feilbruk.

Vi separerte ansvarsområder i egne branches. Dette bidro til at utviklere ikke unødvendig forstyrret hverandre. Når et definert stykke arbeid var utført innen en branch ble en pull request åpnet til hoved-branchen. Hver pull request måtte ha en beskrivelse av sine endringer og en tildelt “reviewer”. Dette gjorde at koden måtte godkjennes av en annen utvikler. Vi mener dette bidro til økt kvalitet i vår utviklingsprosess.

Mye av vår jevne fremgang skyldes daily standups. Som del av møtene må alle beskrive hva de har gjort, hva de skal gjøre og hva som er i veien for fremgang. Dette lot oss fange problemer tidlig, endre fokus dersom nødvendig og se andres bidrag til prosjektet. Dette var effektivt fordi alle var ærlige om deres evne til å gjennomføre deres tildelte oppgaver. Det lot oss hindre at utviklere setter seg fast i en oppgave de ikke kan fullføre.

Vi tok i bruk “transparency”, egne ansvarsområder og klart definerte oppgaver i forsøk på å redusere social loafing. Med transparency mener vi at arbeidet til hver utvikler er synlig i sin mengde, kvalitet og hastighet. Hver utvikler hadde et ansvarsområde med variert omfang i hver sprint. Ansvarsområdene kunne være en hel side, en del av siden eller en funksjonalitet.

Klart definerte oppgaver gjør det lett å vite når man mislykkes. Vi tror social loafing ofte oppstår når individets bidrag ikke kan tydelig kobles til gruppens resultater/fremgang. Med våre tiltak blir hvert individs bidrag synlig og betydelig for deres ansvarsområde. Gjennom DevOps blir antall commits, opprettede arbeidsoppgaver, fullførte arbeidsoppgaver og opprettede pull requests synlig for hvert medlem. Under er bilder av total antall fullførte arbeidsoppgaver for hvert medlem.

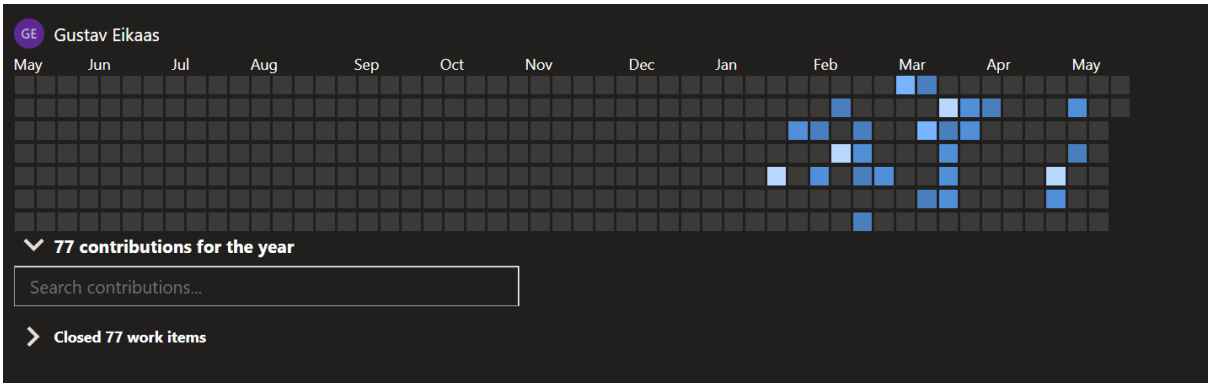


fig 2. Contributions graph for Gustav.

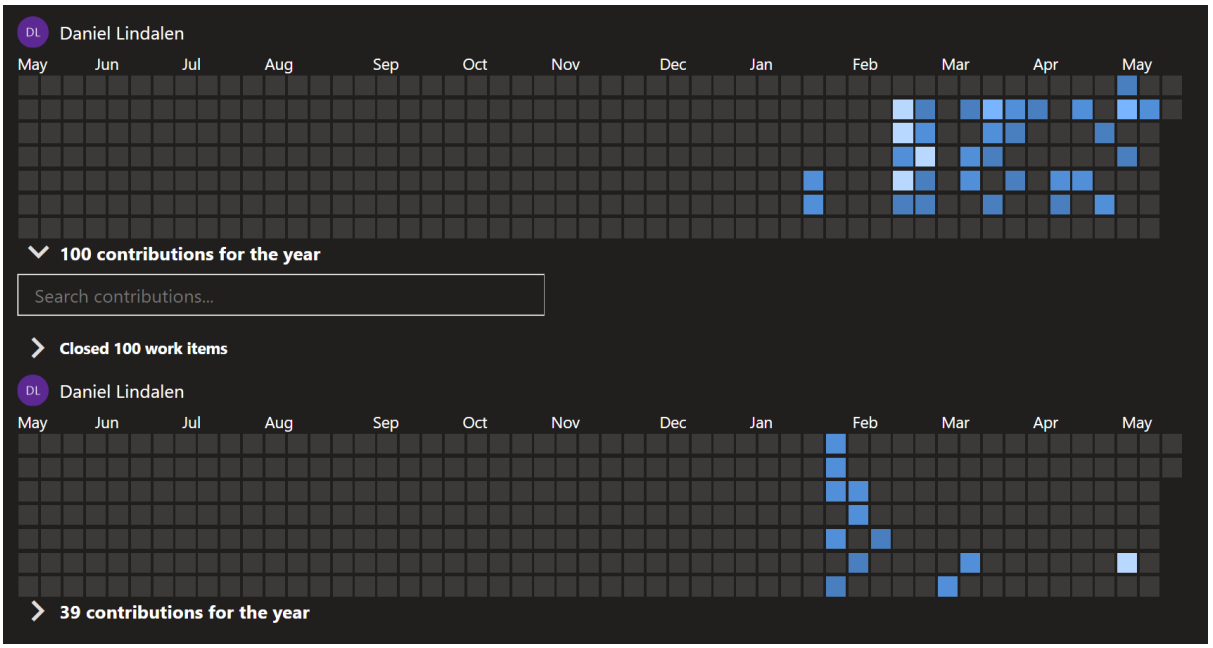


fig 3. Contributions graph for Daniel. Daniel hadde to brukere p.ga et problem med emailen han brukte først.

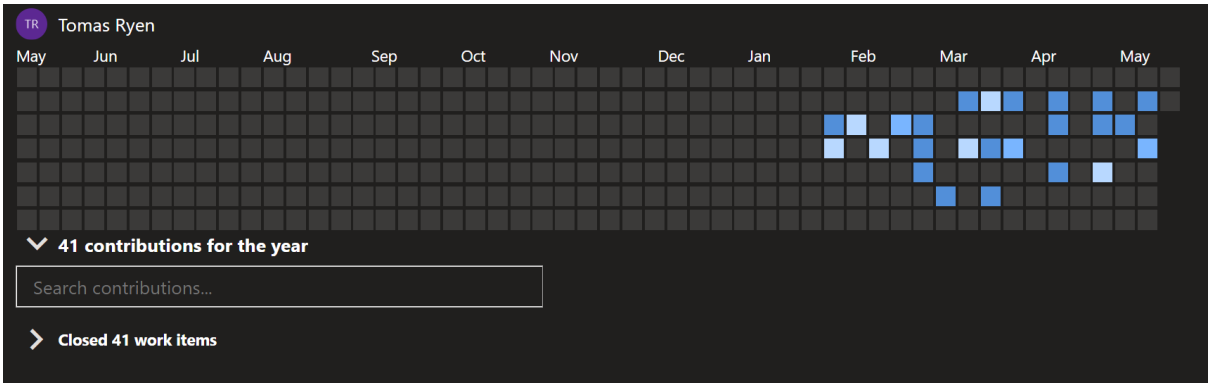


fig 4. Contributions graph for Tomas.

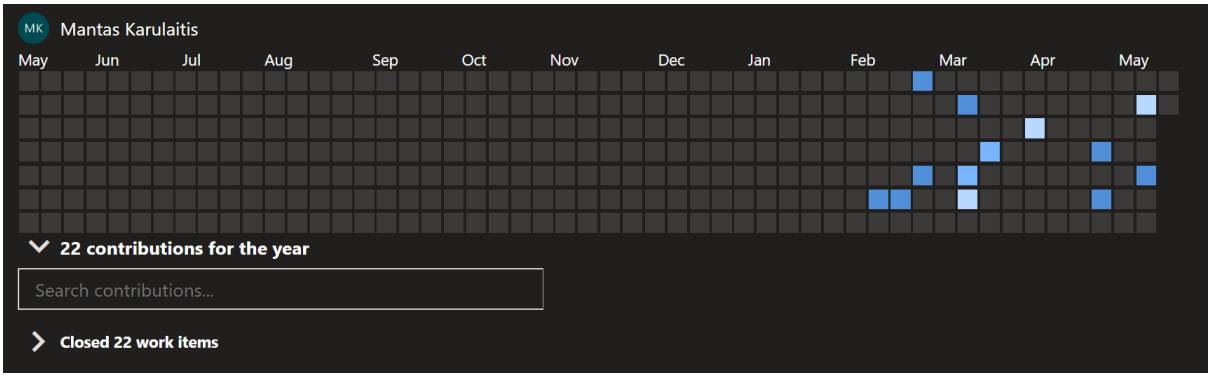


fig 5. Contributions graph for Mantas.

Selv om statistikken i DevOps hjelper oss synliggjøre bidrag, så er det ikke en perfekt løsning. Nøyaktigheten til tallene er sterkt avhengig av at alle medlemmer bruker DevOps på samme vis. Det bør også være likheter i størrelsen på commits og omfanget til arbeidsoppgavene.

4 Produktet

Link til video med forklaring av produktet:
https://drive.google.com/file/d/1E__ZA9A6SchBR41_KBdnG-NfvquAIxjn/view

5 Prosjektgjennomførelse

I dette kapittelet har vi delt opp prosjektgjennomførelsen etter store temaer i utviklingen. I de neste kapitlene beskriver vi stegene vi tok under utviklingen, utfordringene vi møtte på og tiltakene vi tok.

Det er verdt å nevne at utviklingen vi beskriver i dette kapitlet dekker bare store deler av funksjonaliteten vi implementerte. Vi valgte å legge fokus på utviklingen som bød på interessante utfordringer og unike problemstillinger.

I kapitlene som følger brukes ordet “vi” som subjekt i setningene. Det kan antas at “vi” betyr personen, eller personene, som hadde hovedansvaret for ansvarsområdet det snakkes om.

5.1 Database

I dette kapittelet beskriver vi delene av prosjektet relatert til databasen. Databasen er en nødvendig del for lagring av data over tid.

Ved bruk av et verktøy kalt EF Core i backend fikk vi et tett samarbeid mellom kode og SQL som gjorde at vi ikke trengte å skrive noe SQL kode. EF Core lar deg bruke LINQ for å interagere med databasen og oversetter dette automatisk til den mest effektive SQL queryen.

Vi gikk for en "code-first approach" ved bruk av EF Core. Første steg var å identifisere og lage klasser for alle objekter. Vi fulgte de 3 første normalformene i databasemodellering.

Vi fikk ikke alltid svar fra produkteier på hvordan forholdene mellom objektene bør være. Derfor måtte vi ofte gjøre antagelser basert på egen fornuft. Gjennom utviklingen gjennomgikk datamodellen mange endringer. Endringene var som regel knyttet til en ønsket endring i hvordan objektene skulle fungere sammen. Et eksempel er innføringen av sub-prospekter. Et prospekt kan ha mange sub-prospekter. Endringen skyldes at produkteieren fant ut at han ville kunne lage flere prospekter hos samme kunde og prosjekt.

EF Core gjorde det enkelt å endre forhold og felter i objektene, det var bare å endre objektene og apply migration.

5.2 API

I dette kapitlet beskriver vi delene av prosjektet relatert til API. API spilte en sentral rolle i all funksjonalitet, da det var måten vi skapte samarbeid mellom frontend og backend.

5.2.1 REST

Ved starten av prosjektet gikk vi først for REST programmeringsgrensesnitt. Vi valgte REST siden det er raskt å implementere og virket tilstrekkelig for å dekke kravene fra oppdragsgiveren. Vi la vekt på å kunne raskt implementere API fordi en kobling mellom backend og frontend er en viktig del av vår applikasjon.

I sprint 1 fikk vi implementert noen API endepunkter, typisk struktur på de var 127.0.0.1:5001/{Klassenavn}. Det ble implementert CRUD funksjonalitet for dem i løpet av de første sprintene.

I frontend tok vi i bruk endepunkter tidlig slik at brukergrensesnittet kunne basere seg på reelle data fra backend. REST fungerte godt med TypeScript, da det var enkelt å lage interfaces til objektene vi mottok fra endepunkter. For dette brukte vi en nettside som oversetter Json format til TypeScript interface.

Etterhvert fikk vi endepunkter for klasse-relasjoner som: '/selger/prospekter'.

Dette endepunktet lot deg hente en selger og alle tilhørende prospekter. Dette endepunktet støttet kun GET. Det blir etterhvert veldig mange endepunkter å holde styr på.

5.2.2 GraphQL

I sprint 5 begynte vi overgangen fra REST til GraphQL i backend. Årsaken til dette byttet var at vi hadde en del problemer med REST API. Det ble blant annet mange endepunkter å holde styr på og vanskelig å holde en ryddig stil. Produkteier tipset oss om at GraphQL kanskje kunne løse noen av problemene for oss. Noen av problemene vi opplevde med REST viser seg i ettertid å skyldes uerfarenhet med diverse verktøy og språk.

Det første steget i overgangen til GraphQL i backend var å finne et rammeverk som passet. Vi valgte .Net Core-rammeverket HotChocolate GraphQL. Vi var klar over at dette ikke var et ferdig utviklet rammeverk og risikoene dette kunne medføre. Oppsett av rammeverket i koden var rimelig enkel.

For å separere de interne klassene i koden fra de klassene du eksponerer i API'et lager man en form for kopi av klassen. Da kan du velge å skjule felter og definere hvordan feltene løses, dersom du har en databaserelasjon mellom de. Vi valgte å implementere "soft delete" i prosjektet vårt som vil si at data aldri blir slettet fra databasen, men kun flagget som slettet. Dette er et felt du ikke vil at API'et skal eksponere.

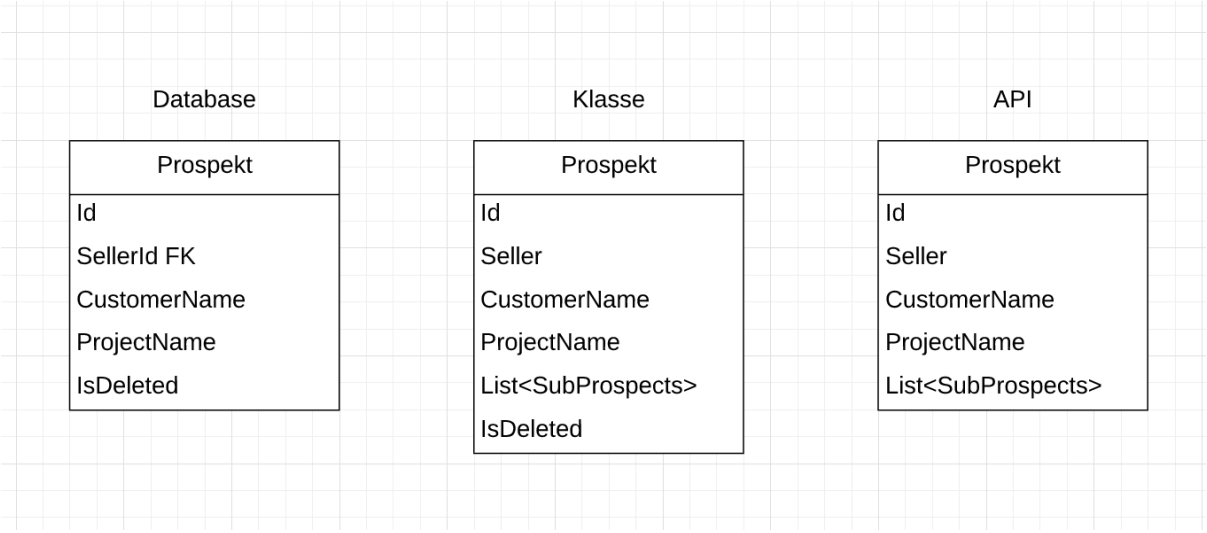


fig 6. Tre like objekter med varierende felter for hvor i "pipelinen" de brukes

Etter vi hadde satt opp klassene begynte vi å deklarere queries. Dette innebar å definere en metode som returnerer et resultatsett fra databasen. Ved bruk av EF Core kunne man returnere en såkalt "*Queryable*" istedenfor et faktisk resultatsett, som vil si at den henter data fra databasen som matcher din query, istedenfor å hente alt fra databasen og kun sortere ut etter din query.

For å kunne slette, endre og opprette objekter måtte det deklareres såkalte mutations. Dette er metoder som tar imot et objekt som input, og returnerer det nye/endrede objektet som output. Denne klassen ble fort rotete da man bare kunne definere en mutation-klasse, men dette ble løst senere når vi fant ut at det var mulig å splitte opp i flere klasser ved hjelp av en løsning i rammeverket.

Vi var usikre på hva som var den beste måten å sørge for at informasjon holder seg oppdatert på i frontend, spesielt når flere personer jobber på samme side. Det ble implementert subscriptions i GraphQL, som eksponererer en form for "event listener" hvor klienter kan lytte etter spesifikke endringer. Vi valgte å dele de inn i 3 former for endringer per objekt: On{objekt}Added ,On{objekt}Edited, On{objekt}Deleted. Dette ble aldri implementert i frontend.

Et kjent problem i alle GraphQL servere er det vi kaller N+1 problemet. Ifølge utviklerne bak GraphQL-implementasjonen vi brukte var løsningen å implementere "batch data loaders". Data loaders samler opp queries til databasen og utfører de i batch, det vi desverre ikke visste var at EF Core allerede kunne løst dette problemet for oss enkelt. Mye unødvendig tid gikk til implementasjon av "data loaders" for så å fjerne det i etterkant.

I frontend måtte vi gå over til Apollo Client for å håndtere spørringer til vår GraphQL API. Dette innebar en del "strukturelle" endringer i våre React komponenter da Apollo Client også har state management. Vi beskriver hvordan vi brukte verktøyet i de neste kapitlene.

5.3 Kalender

I dagens løsning bruker sidene for belegg, prospekter og ledige ressurser et kalendersystem. Alle sidene viser informasjon over tid gruppert etter tema. I siden for belegg vises dagene konsulenter har på kontrakt over tid. I siden for prospekter vises prospekter selgere jobber med over tid. I siden for ledige ressurser vises antall ledige dager til hver konsulent over tid. I dette kapitlet beskriver vi hvordan vi implementerte et kalendersystem i produktet.

Tidlig i prosjektet hadde vi et møte med en utvikler i Webstep for å diskutere problemstillingen. Fra diskusjonen kom det frem at gjenbrukbarheten til koden bør nedprioriteres. I stedet bør det prioriteres å lage fungerende kalendere på prospekt-siden og belegg-siden. Dette innebærer å duplisere kode, for deretter å gjøre den egnet et nytt sted. Argumentet var at det vil være lettere å gjøre koden generell når et kalendersystem har blitt implementert begge steder. Dette var nytt for oss, men vi valgte å følge rådet.

Utviklingen av kalendersystemet begynte i sprint 1. Vi la vekt på å gjøre kalenderen enkel i starten. Kravene besto av å kunne vise prospekter for hver selger langs en tidslinje på 52 uker basert på startdato og sluttdato. For å redusere kompleksitet omdannet vi datoene til uketall, som da kunne brukes som kolonnenummer. Altså, dersom et prospekt starter i uke 4 og slutter i uke 7, skal elementet dekke kolonne 4-7. Vi lagde en demo som oppfylte kravene iløpet av den første sprinten. Under er et bilde av hvordan det så ut.

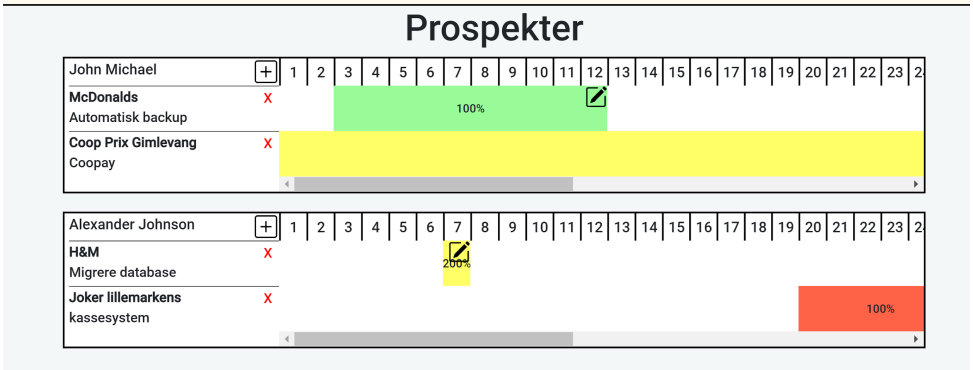


fig 7: Første versjon av kalendersystemet

Fra et designperspektiv er denne løsningen svært manglende. Et åpenbart problem er hvordan dette skal skaleres til 100+ selgere. Et annet problem er at elementer, som linjen med uketall, gjentas uten at de skaper verdi. Likevel var det et nyttig utgangspunkt for diskusjon med produkteier om veien videre.

I de tre neste sprintene fokuserte vi på handlingene som skulle kunne utføres på prospekter. Kalendersystemet var ikke direkte knyttet til dette. Det er fordi systemet sin rolle i dette stadiet er å holde seg oppdatert med den nyeste versjonen av prospektene. Kalenderen var altså bare en view.

I sprint 5 ble fokuset skiftet tilbake til kalendersystemet. Fokuset skulle være på å utvide tidslinjen og å gjøre elementer flyttbare langs tidslinjen. Dette innebærer å håndtere dato, uketall og event-lytting.

Ved starten av sprint 5 tok ikke uketallene hensyn til ISO 8601, som er vår valgte standard for tid. Målet i denne sprinten var å utvide tidslinjen til 104 uker, ta hensyn til at noen år har 53 uker og begynne tidslinjen fra nåværende uke. Dette krevde funksjonalitet som kunne gi oss antall uker i året og nåværende uketall. Vi valgte å utvikle dette selv med hjelp av kodesnutter fra nettet i stedet for å bruke en pakke som Moment.js.

Det var to store utfordringer med å lage flyttbare prospekter. Den første var å utvikle logikk for flytting mellom år. Den andre var å dekke kildene til bugs og unntakstilfellene som kunne oppstå ved flytting. Detaljene rundt hvordan selve flyttingen fungerer er beskrevet i kapitlet om Inline Edit. Kort, måles avstanden musen beveger seg mens elementet er valgt for å beregne antall kolonner elementet skal flyttes. Kolonner tilsvarer uker.

ISO uker kan begynne og slutte i forskjellige år. Dette gjør at det ikke er en klar kobling mellom uketall og år. Ved flytting av prospekter til neste år må slutt-uketallet settes tilbake til for eksempel 1. Dette er ikke nok å vite, siden det kan være 3 steder i tidslinjen der uketallet er 1. Vår løsning er å konvertere uketall og år til dato, legge til 7 dager, for å finne ut hvilket uketall den datoen har. Vi tok utgangspunkt i ISO 8601, der uker begynner på mandager.

Derfor ble uketallet prospekter starter konvertert til mandag mens uketallet prospekter slutter ble konvertert til søndag. Dette gjorde at start og slutt måtte ha egen logikk, da de har forskjellige unntakstilfeller å ta hensyn til.

Vi unngår å beskrive i særlig detalj de forskjellige unntakstilfellene logikken til både endring av startdato og sluttdato måtte håndtere. I brukergrensesnittet måtte vi hindre at uker kunne flyttes utenfor den første og siste uken i tidslinjen. For flytting mellom år måtte det sjekkes om datoen uketallene representerer er fra siste uke i forrige år eller første uke i neste år. Det måtte hindres at man kunne dra startuke forbi sluttuken. Det ble også gjort innsats for å hindre at ugyldige uketall ble sendt i API kallet for redigering.

Det oppsto en del bugs når vi håndterte datoer i frontend. En av de var at backend alltid konverterte datoene vi sendte inn til forrige dag. Det viste seg at det skyldtes en "timezone offset" som JavaScript legger til i sine Date objekter. Problemet ble løst ved å fjerne "timezone offset" før vi sendte inn datoene. Under er bilde av funksjonen vi lagde for dette.

```
export function toISOStringNoTimezoneOffset(d: Date): string {
    return new Date(d.getTime() - d.getTimezoneOffset() * 60000).toISOString();
}
```

fig 8: Viktig funksjon for å sikre at backend og frontend håndterer dato på samme vis

Etter konsultasjon med produkteier i sprint 7 fant vi ut at datohåndtering burde gjøres i backend. Produkteieren hadde opplevd at håndtering av dato i frontend ofte kunne skape uforventede bugs og problemer. Vår opplevelse stemte overens med hans mistanker.

Overgangen til at backend skulle håndtere datoer forenklet frontend-siden av kalendersystemet betydelig. Mye av logikken kunne fjernes da den var kun nødvendig dersom uketall måtte konverteres til dato. Datoer ble erstattet med "WeekYear" objekter, som er et objekt med felt for uketall og år. Vi måtte fremdeles håndtere logikk rundt antall uker i hvert år, da frontend fremdeles hadde ansvaret for å endre uketall ved flytting.

```
export interface WeekYear {
    year: number;
    week: number;
}
```

fig 9: Interface som ble brukt i stedet for datoer etter ansvaret for datohåndtering ble flyttet til backend.

I backend ble API endret til å ta imot startuke, startår, sluttuke og sluttår istedenfor dato. Ved mottakelse av slik informasjon ble det konvertert til dato før det ble lagret. For å håndtere dato i backend brukte vi en pakke kalt NodaTime. NodaTime håndterte logikken og valideringen som frontend tidligere gjorde manuelt.

I sprint 9 skulle kalendersystemet implementeres i belegg-siden. Som tidligere nevnt ble vi rådet til å utvikle et kalendersystem per side. Første versjon av kalendersystemet i belegg-siden ble derfor veldig likt som i prospekt-siden. Forskjellene besto for det meste av at komponentene tok imot kontrakter i stedet for prospekter.

Kort tid etter vi hadde implementert et grunnleggende kalendersystem for belegg-siden begynte vi å utvikle et system som begge sidene kunne bruke. Dette lot seg gjøre ved gjennom å oppfylle fire krav. De neste avsnittene beskriver endringene i dataflyt og komponent-hierarkiet som måtte til for å oppfylle kravene. Et viktig begrep å kjenne til før man leser videre er "props". Props er dataen en komponent forventer å motta.

Det første kravet er at kalenderen må kunne vise både kontrakter og prospekter. En viktig endring vi gjorde for å støtte dette var å endre interfacet som elementer i tidslinjen forventer å motta. Tidligere var det enten en kontrakt eller et prospekt. Vi erstattet dette med et interface som kun krever feltene id, startår, startuke, sluttår og sluttuke. Under er et bilde av dette interfacet.

```
interface Eventable {
    id: number;
    startWeek: number;
    startYear: number;
    endWeek: number;
    endYear: number;
}
```

fig 10: Generelt interface for alle plasserbare objekter i tidslinjen

Det andre kravet er at innholdet til elementene i tidslinjen må kunne defineres dynamisk gjennom props. Dette er fordi vi ønsket å separere logikken relatert til posisjonering i kalendersystemet fra definisjonen til synlig innhold. Tidligere ble innholdet definert i samme kode som inneholdt logikken for flyttbare elementer. Vi løste dette med å bruke en variasjon av en React-teknikk som heter "render props". Ideen er at en komponent bruker en prop, som regel en funksjon, til å bestemme hva den skal vise (ReactJS, 2021). I vårt tilfellet lagde vi en prop kalt render som forventer å motta HTML elementene som skal vises. Dette gjorde at vi kunne dynamisk plassere ønsket HTML i elementet. Under er et bilde som viser at kontrakter og prospekter har forskjellig innhold selv om de begge er basert på samme komponent.

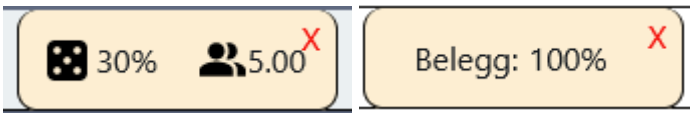


fig 11: Elementet for prospekter i tidslinjen (venstre) og kontrakter i tidslinjen (høyre).

Det tredje kravet er at ansvaret for å definere API kallet nødvendig for å redigere posisjon måtte flyttes "opp" et nivå. Dette er fordi hvert objekt har forskjellige API kall for å endre

sine felt. Hvis koden for vising av objektene skal være generell, må slik objekt-spesifikk funksjonalitet flyttes til komponenten som implementerer kalendersystemet.

Det andre og tredje kravet gjorde at vi tok i bruk "Container Component"-mønsteret hver gang en ny type objekt skulle vises i tidslinjen. Vi lagde slike komponenter for kontrakter, prospekter og sykdom/fri. En "Container Component" inneholder all kode som ikke er direkte knyttet til visning (ReactPatterns, 2021). Dette mønsteret lot oss skille API kall, logikk og HTML elementer spesifikt for hvert objekt fra koden for visning i tidslinjen. Totalt må det å lages to komponenter hver gang et nytt objekt skulle bli mulig å vise i kalendersystemet. Disse er *{objekt navn}EventContent* og *{objekt navn}EventContainer*. Under er et eksempel på hvordan *ContractEventContainer* er satt sammen.

```
export const ContractEventContainer: React.FC<ContractEventContainerProps> = ({ contract, consultantId }) => {
    // Hook for calling the mutation for editing contracts
    const [editContract] = useMutation<EditContractPayload, { input: EditContractInput }>(EDIT_CONTRACT, {···
    });

    // API call for editing time-related fields specifically
    const editPlacement = (input: Eventable) => {···
    };

    // API call for editing any field on the contract
    const editContractWrapper = (c: Contract) => {···
    };

    let color = 'rgb(255, 239, 213)';
    return (
        <CalendarEvent
            eventObj={contract}
            color={color}
            render={<ContractEventContent contract={contract} editCallBack={editContractWrapper} />}
            editPlacement={editPlacement}
            deleteMutationString={DELETE_CONTRACT}
        />
    );
};
```

fig 12: Koden til container-komponenten for kontrakt som hendelse i kalenderen. (innhold av funksjoner minimert)

Det fjerde kravet er at knappen for sletting, som befinner seg i hjørnet av hvert element i tidslinjen, må bli gjort generell. Tidligere ble koden for sletting duplisert slik vi hadde egne knapper for sletting av prospekter og kontrakter. Vi gjorde det generelt ved å lage en komponent som mottar definisjonen på mutasjonen for sletting og IDen til objektet som skal slettes. Dette gjør at hvilket som helst objekt blir slettbart i tidslinjen så lenge props fylles med gyldig data.

```
interface DeleteCornerButtonProps {
    id: number;
    deleteMutationString: DocumentNode;
}

export const DeleteCornerButton: React.FC<DeleteCornerButtonProps> = ({ id, deleteMutationString }) => {
    const [deleteSelf] = useMutation<number, { input: { id: number } }>(deleteMutationString);
```

fig 13: Bilde som illustrerer hvordan knappen for sletting er generell

Til slutt hadde vi et kalendersystem som fungerte uavhengig av objektene som skulle vises. Vi tok i bruk kjente mønstre og teknikker i React for å oppnå gjenbrukbarhet. Strategien anbefalt fra utvikleren i Webstep fungerte godt for oss. Vi hadde ingen erfaring med React da prosjektet startet, så det ville vært urealistisk å skape et generelt system fra starten.

```
interface CalendarEventProps {
    eventObj: Eventable;
    render: JSX.Element;
    color: string;
    deleteMutationString?: DocumentNode;
    editPlacement: (input: Eventable) => void;
}
```

fig 14: Dataen hvert element i tidslinjen mottar. Reflekterer de fire kravene vi nevnte i de forrige avsnittene.

5.4 Belegg-siden

I dette kapitlet beskriver vi problemstillingene og utfordringene vi møtte på under utviklingen av belegg-siden. Vi har valgt å fokusere på det som er urelatert til kalendersystemet og inline-editing mønsteret.

5.4.1 Kapasitet og ledige ressurser

Siden i dagens løsning for ledige ressurser viser antall ledige dager over tid for hver konsulent. Sammen med produkteier kom vi frem til at denne siden burde slås sammen med belegg-siden. Argumentet for å kombinere dem var at antall ledige dager var det inverse av antall dager på kontrakt.

Vi gikk gjennom flere design-ideer før vi begynte utviklingen av ledige ressurser. Målet var å finne en måte å inkludere ledige ressurser i tidslinjen til belegg-siden. Etter et par diskusjoner med produkteieren kom vi frem til at det holdt å vite prosenten av total kapasitet hver konsulent har på kontrakt. Altså, det vil ikke vises antall ledige dager, men istedet hvor mye av arbeidsuken konsulenten har på kontrakt.

På prospekt-siden er det et tomrom i brukergrensesnittet i tidslinjen etter selgerens navn. Vi bestemte oss for at dette tomrommet kunne fylles med kapasitet på belegg-siden. For å vise kapasitet må frontend beregne, eller motta beregninger, på sum antall dager på kontrakt for hver uke. Vi bli enige med produkteier om å sette 5 dager i uken som 100%. Vi endte opp med å gi backend ansvaret for beregningen av kapasitet. Backend fikk ansvaret siden vi ville unngå å håndtere datoer i frontend.

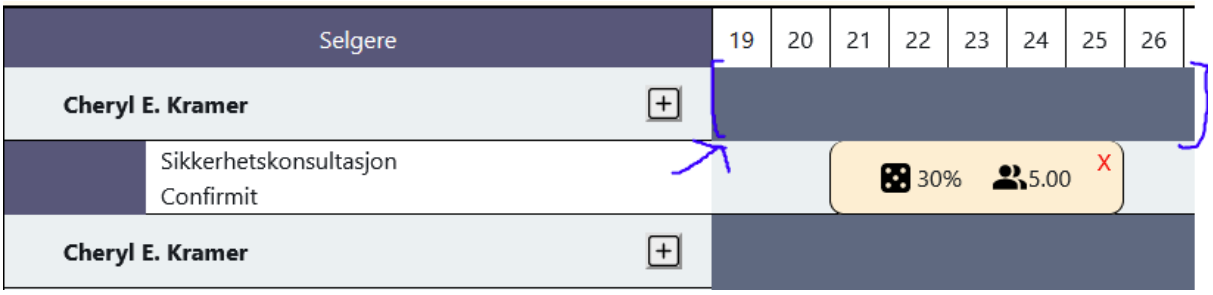


fig 15: Tomrommet på prospekt-siden i tidslinjen etter selgerens navn

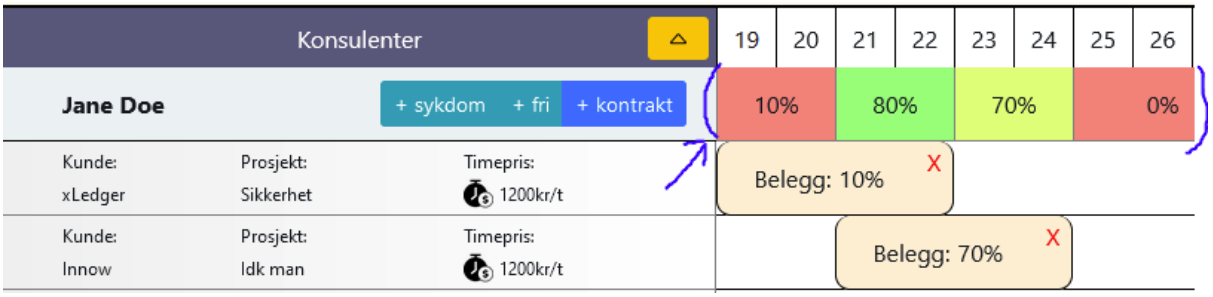


fig 16: Kapasiteten til konsulenten vises

5.4.2 Føring av sykdom og fri

Selgere må ha en måte å føre antall dager en konsulent er syk eller har fri i hver uke. I dagens løsning fylles en celle for hver uke med et tall som representerer antall dager. Vi innså at den enkleste måten å implementere slik føring var med å lage "spesielle" kontrakter som kunne flyttes langs tidslinjen. I likhet med vanlige kontrakter vil man da kunne fylle inn antall dager, men dagene skal ikke behandles som dager på en normal kontrakt. Selv om dette ville fungert, valgte vi å droppe denne ideen. Vi likte ikke at urelatert funksjonalitet skulle utnytte kontrakter til en annen hensikt.

Vi endte opp med en løsning som tar lite plass i grensesnittet, krever ingen forandring i eksisterende funksjonalitet og har ingen sideeffekter på urelaterte verdier. Første steg var å lage en ny klasse som representerer tid borte. Vi valgte å kalle denne klassen Vacancy. Vi kom frem til klassen etter vi lærte fra produkteier at det kun skilles mellom to typer fravær; planlagt og uplanlagt. I bildet under er interface for denne klassen i frontend.

```
export interface Vacancy {
    id: number;
    planned: boolean;
    daysOfWeek: number;
    startYear: number;
    startWeek: number;
    endYear: number;
    endWeek: number;
}
```

fig 17: Interface for Vacancy

Det andre steget var å finne ut hvordan fravær skulle vises i tidslinjen. Det første instinktet var å vise dem blant kontraktene. Problemet med dette var at brukergrensesnittet var tilpasset visning av kontrakter. Ved å vise fravær og kontrakter på samme sted på samme vis risikerer vi at brukergrensesnittet blir uoversiktlig. Vi ville unngå at helt urelaterte ting vises på samme måte.

Vi endte opp med å vise fravær langs linjen for kapasitet. Fraværet skulle ikke påvirke kapasiteten på noen vis. En viktig detalj er at kapasitet ikke er relevant for selgeren i tidsrom der konsulenter er borte eller syk. Derfor kan det være akseptabelt at kapasitet er blokkert i tidsrommet konsulenter er borte eller syk. For å la to tidslinjer vises på toppen av hverandre brukte vi to like CSS grids med forskjellige z-indekser. Vi måtte skille mellom tidslinjen til kapasitet og tidslinjen for fravær siden kun den ene skulle ha flyttbare elementer. Vi måtte ikke skrive noen ny kode for å gjøre fravær flyttbar langs tidslinjen, da flyttbarheten til prospekter og kontrakter var blitt gjort gjenbrukbar. Produkteieren godkjente løsningen etter å ha sett en demo. Under er et bilde av sluttversjonen av det vi beskrevet i disse avsnittene.

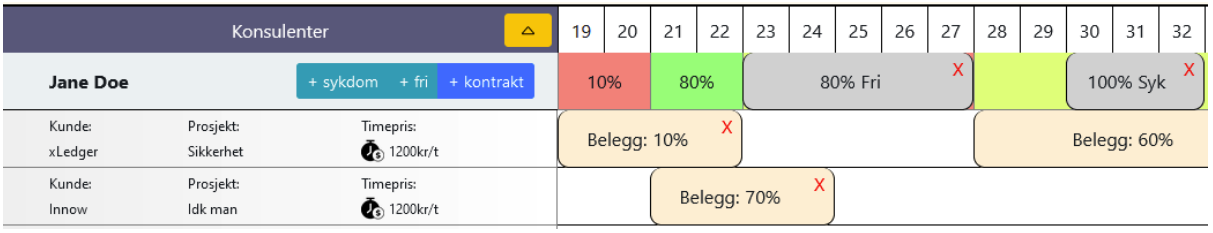


fig 18: Eksempel på konsulent som har fri i uke 23-27 og er syk i uke 30-32.

5.5 Inline edit

Inline edit er et design-mønster der innhold er redigerbart uten å måtte forlate siden (PatternFly, 2021). Vi utvidet denne definisjonen til at redigering skal kunne utføres uten å blokkere innhold eller fryse oppdateringer. Vi kom frem til at dette var en god ide etter vårt første styringsgruppemøte med veileder og oppdragsgiver. I dette møtet ble det nevnt at modalene vi brukte for å utføre handlinger blokkerte innhold, hindret andre handlinger og var generelt forstyrrende.

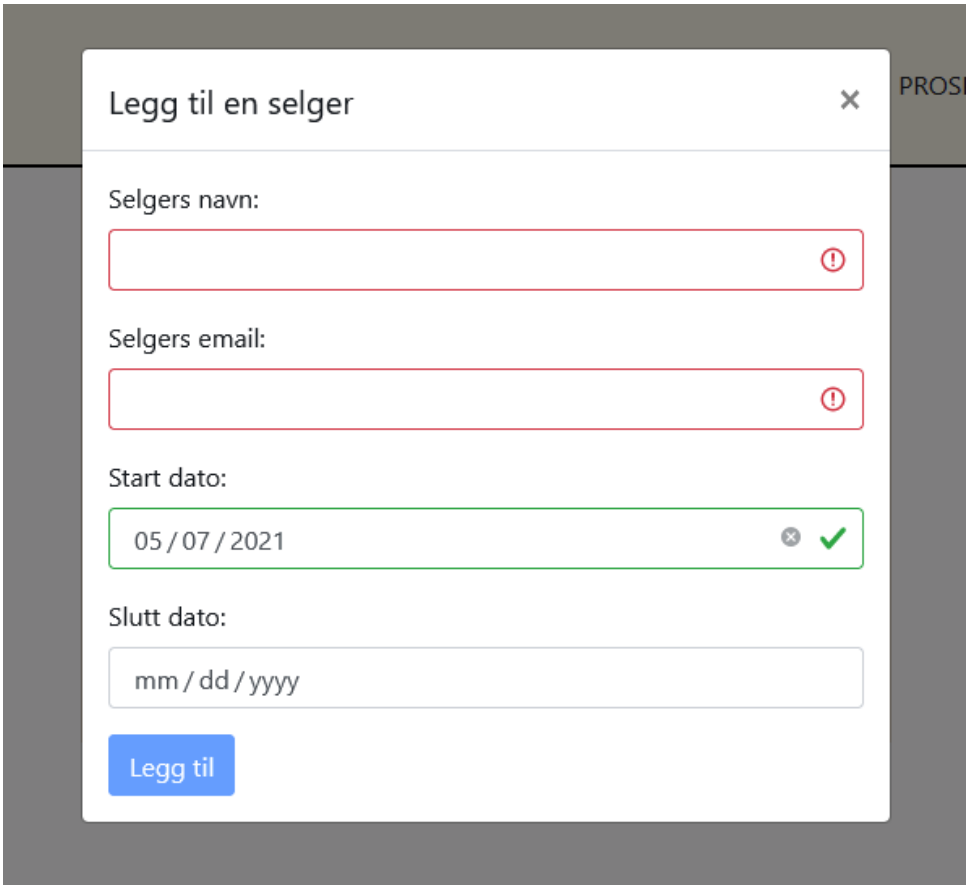


fig 19: Modal for oppretting av selgere.

Vi kom frem til at et bedre alternativ er at redigering følger inline edit mønsteret. Etter å ha diskutert hvordan dette bør implementeres med produkteieren ble det satt som fokus for sprint 5. Sluttmålet var at alle modaler skulle bli erstattet. Vi regnet med at dette ville ta flere sprints, der modaler blir gradvis erstattet.

Planen for implementering varierte etter handlingen som skulle utføres. Ordet handling i dette tilfellet referer til en type redigering. Felles for alle handlingene er at det skulle være færrest mulig steg i utføringen av dem. I de neste avsnittene beskriver vi hvordan alle handlingene ble implementert. Måten handlinger ble implementert på var lik på tvers av sider og objekter. Detaljene rundt implementeringen av handlingene for flytting og endring av lengde befinner seg i kapitlet om kalendersystemet.

5.5.1 Planen

For redigering av synlig data var planen at tekst erstattes med et redigerbart felt ved musetrykk. Dette feltet vil da fungere som det ville gjort i et skjema, der innholdet av feltet erstatter teksten som var etter fullført endring. En utfordring med slik løsning er hvordan brukeren skal bekrefte at endringen er ferdig. Vi kom frem til at endringen bør utføres dersom feltet ikke lengre blir fokusert på. Altså, dersom brukeren endrer fokus etter å ha redigert informasjonen, sendes et API kall for redigering.

For oppretting var planen å lage objekter med standardverdier etter knappetrykk. Objektene vil da kunne redigeres videre etter oppretting. Slik implementering sikrer at brukeropplevelsen er uforstyrret. Under er et eksempel på hva som skjer i brukergrensesnittet ved oppretting av ny kontrakt.

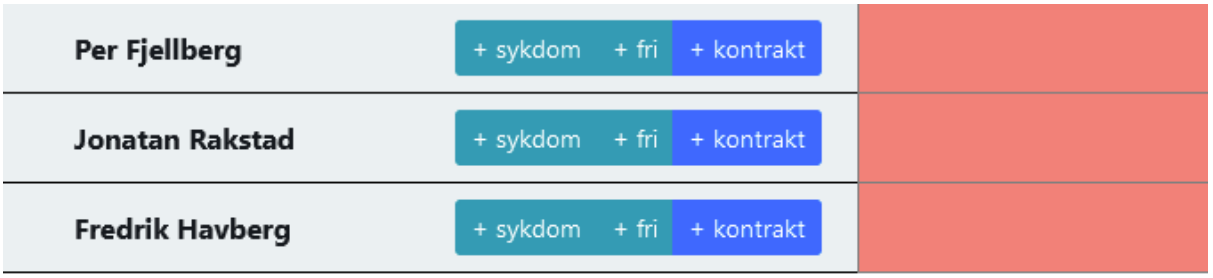


fig 20: Før oppretting av ny kontrakt

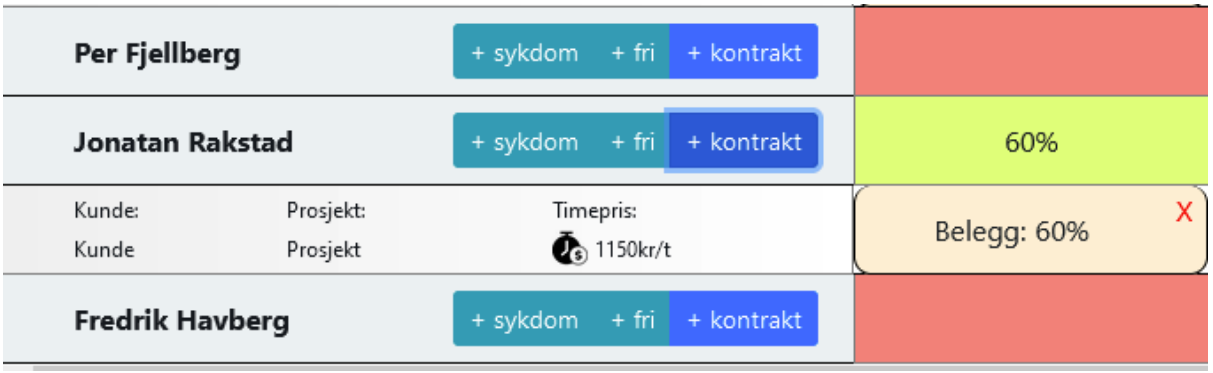


fig 21: Etter oppretting av ny kontrakt

Som vi kan se fra bildene blir en kontrakt opprettet med standardverdier for kundenavn, prosjektnavn, timepris, belegg og tidsrom.

For flytting langs kalenderen var planen å la brukeren kunne "dra-og-slippe" elementer. Dette innebærer at man kan trykke på elementet, holde inne musepekeren og dra elementet til ønsket posisjon. Dette lar brukeren redigere startdato og sluttdato med samme mengde i én handling. Når brukeren "slipper" elementet vil et API kall for endring av startdato og sluttdato bli sendt. Før dette så ble posisjonen til elementer endret gjennom utfylling av et skjema.

For endring av lengde var planen å kunne dra i sidene til elementer i tidslinjen. Her regnet vi med å kunne gjenbruke en del logikk fra flytting langs tidslinjen. Med denne løsningen skal brukeren kunne "dra-og-slippe" en kant av elementet til ønsket posisjon. Når brukeren "slipper" kanten, blir et API kall sendt for endring av tilhørende felt. Kanten til venstre tilsvarer startdato. Kanten til høyre tilsvarer sluttdato.

For sletting av elementer beholdt vi bruken av modaler. Dette var fordi produkteier ønsket at sletting måtte bekreftes før det ble utført.

5.5.2 Implementeringen

Vi prioriterte først utvikling av kjernefunksjonaliteten til redigering av synlig data. Kjernefunksjonaliteten er at verdier kan redigeres ved å trykke der de vises. Vi valgte å utvikle dette som en gjenbrukbar React-komponent. Komponenten mottar et input-element og teksten du vil vise (som regel verdien til et felt). Hvis komponentens innhold blir fokusert på, vises input-elementet, hvis ikke, vises teksten.

Vi brukte denne komponenten i implementeringen av redigering av tall, tekst og prosent. Det måtte også tas hensyn til data som skal redigeres gjennom en liste av gyldige verdier. Totalt ble komponenten implementert på fire forskjellige vis. Det holder å vite at hver implementering hadde egen validering, måte å vise feltets verdi og input-element.

For oppretting av objekter fant vi et gjenbrukbart mønster. Mønsteret går ut på at en React-komponent mottar ID til forelder-objektet som skal bli tilføyet et nytt objekt. For eksempel mottar knappen for å legge til prospekter IDen til selgeren som prospektet skal legges til. Denne IDen vil bli brukt som parameter i mutation for å legge til nytt objekt. Utenom dette må et objekt med standardverdier tilføyes i parametrene til GraphQL mutasjonen. Under er et eksempel på hvordan oppretting av prosjekter bruker dette mønstret.

```
addProject({
    variables: {
        input: {
            consultantId: consultantId,
            customerName: 'Kunde',
            projectName: 'Prosjekt',
            hourlyRate: 1150
        }
    }
})
```

fig 22: Eksempel på hvordan oppretting av prosjekter implementeres

Dette fungerte godt når det bare skulle opprettes et objekt uten “nestede objekter”. For objekter med slike objekter måtte mønsteret brukes rekursivt. Med dette mener vi at mønsteret implementeres på nytt for hvert nivå av “nesting” objektet har i sine nestede objekter. Under er et eksempel på hvordan dette ble gjort for oppretting av prospekter.

```
let defaultProspect = getDefaultProspect(sellerId);

addProspect({ variables: { input: defaultProspect } })
    .then((res) => {
        let newProspectId = res.data?.addProspect.prospect.id;

        if (newProspectId !== undefined) {
            let defaultSubProspect = getDefaultSubProspect(newProspectId);

            addSubProspect({ variables: { input: defaultSubProspect } });
        }
    });
```

fig 23: Eksempel på hvordan oppretting av nytt prospekt med ett "sub-prospekt" implementeres (kode forenklet for leselighet)

For flytting av elementer langs tidslinjen, altså "dra-og-slipp", fant vi også et mønster for implementeringen. Først lytter elementene etter å ha blitt trykket på. I tidsrommet elementet er trykket på måles avstanden musen beveger seg i horisontal retning. Dersom musen beveger seg tilsvarende bredden på en kolonne, økes eller reduseres startuken og sluttuken basert på retning. Detaljene rundt logikken bak endring av uketall og dato blir beskrevet i kapitlet om kalendersystemet. Under er et bilde av logikken vi beskrev i dette avsnittet, forenklet for leseligheten.

```
if (distance > columnSize) {
        incrementStartWeek();
        incrementEndWeek();
} else if (distance < -columnSize) {
        decrementStartWeek();
        decrementEndWeek();
}
```

fig 24: Eksempel på logikk for flytting av elementer langs tidslinje (kode forenklet for leselighet)

For endring av lengde brukte vi noe som kalles dragbars. En dragbar er et element festet ved en kant som lar deg endre lengden på objektet den er festet på. Dette viste seg å være den mest utfordrende handlingen å implementere.

Utviklingen av dragbars begynte i sprint 7. Vi antok at det ville fungere i stor likhet med flytting. Den eneste forskjellen ville være at kun startdato eller sluttdato endrer seg. Dette viste seg å være både riktig og feil. Første versjon av dragbars ut til å fungere som de skulle, der mye av koden kunne lånes fra koden relatert til flytting. En av grunnene til at dette fungerte var fordi vi kunne fortsette å dra elementet selv om musepekeren forlot elementet. Vi oppdaget en stund etter dette at dragbars ikke fungerte i andre nettlesere enn Mozilla Firefox. Årsaken viste seg å være at Mozilla Firefox har en bug som lar såkalte MouseMove events fyre selv etter musepekeren har forlatt elementet (BugZilla, 2021).

Fordi første versjon av dragbars var avhengig av en bug valgte vi å starte på nytt. For å forstå hvordan dragbars implementeres uten å la musepekeren forlate elementet undersøkte vi Google Calendar. I Google Calendar fant vi at dersom musen forlater dragbaren mens man drar i den, så vokser elementet en kolonne i samme retning. Elementet vokser såpass raskt at musepekeren ikke rekker å forbli utenfor elementet mens man drar.

Mye av kompleksiteten bak å lage dragbars er orientert rundt hva som skal skje dersom musen forlater dragbaren. Kompleksiteten kommer fra de forskjellige scenarioene koden må håndtere.

Prosessen for å håndtere diverse scenarioer kan brytes opp i krav. Det første kravet er at det lyttes etter at musen forlater dragbar mens den blir trykket på. Dette lar oss knytte logikk til hendelsen. Det andre kravet er at vi beregner nærmeste kant musen forlot elementet. Nærmeste kant bestemmer retningen elementet skal vokse. Dersom musen forlater bunnen eller toppen av elementet bør flyttingen stoppe, da vi har begrenset lengde-endring til den horisontale retningen. Det tredje kravet er å kalle metodene for å vokse og krympe basert på kanten dragbar er festet til. Hvis dragbar er festet til venstre kant, vil forlating i retning venstre tilsvare voksing og høyre krymping. Det fjerde kravet er å legge inn tidsrom det er tillat at musen forlater dragbar. Vi satt grensen til 300 millisekund. Årsaken til dette er at det må regnes noen millisekund der elementet ikke har tatt igjen musepekeren. Hvis tiden går ut uten at musepekeren kommer inn i elementet igjen stopper flyttingen. Bildet under illustrerer mye av koden vi beskrev i dette avsnittet.

```
let { edge, distance } = getClosestEdgeAndDistance(dragBarDivRef, e);

if (distance < 5 && (edge === 'top' || edge === 'bottom')) {
    setIsDragging(false);
    setIsDraggingParent(false);
    return;
}

if (direction === 'left') {
    setTimeoutId(startTimeoutToStopDragging(300));

    if (distance < 3 && edge === direction) {
        growInDirection();
        stopTimeoutToStopDragging(timeoutId);
        return;
    } else if (distance < 3 && edge === 'right') {
        shrinkAgainstDirection();
        stopTimeoutToStopDragging(timeoutId);
        return;
    }
}
```

fig 25: Utdrag av koden til logikken til dragbars (forenklet for leselighet)

Til slutt hadde vi erstattet alle handlinger for redigering med funksjonalitet som følger inline edit mønsteret. Applikasjonen ble betydelig enklere å bruke på grunn av valget om å følge dette mønsteret. Nesten all kode relatert til redigering måtte skrives på nytt. Vi fant mønstre for hver handling, noe som betydelig reduserte tiden det tok å erstatte modalene.

5.6 Støtte for sanntid samarbeid

Vår applikasjon måtte støtte at flere selgere kan jobbe med samme data samtidig. Vi hadde ingen erfaring med utvikling av verktøy som støtter slikt samarbeid ved oppstart. Vi brukte en utvikler i Webstep som rådgiver tidlig i utviklingsfasen for å bedre forstå hvordan sanntid samarbeid kan implementeres.

I møtet med utvikleren lærte vi at det var viktig å ha kun én kilde til sannhet. Dette betyr kun én del av systemet har ansvaret for å ha den "riktige" versjonen av dataen. Vi valgte å gjøre databasen til vår kilde til sannhet. Dette gjør at brukergrensesnittet sin måte å holde seg oppdatert er gjennom å hente data. I de neste avsnittene beskriver vi stegene vi tok og utfordringene vi møtte på i implementeringen av sanntid samarbeid.

Selv om vi ikke vet nøyaktig hvordan dagens løsning skaper støtte for sanntid samarbeid, ser det ut som hyppige små API kall ved hver brukerhandling spiller en stor rolle. Vi baserer dette på å ha undersøkt Google Sheets gjennom utviklerverktøy i nettleseren som lar oss lese nettverkstrafikken. Under er antall API kall etter å ha fylt inn seks celler i Google Sheets.

fig 26: Antall requests i Google Sheets ved utfylling av seks celler i løpet av 6.67 sekunder

Tidlige versjoner av produktet klarte å holde brukergrensesnittet synkronisert med databasen uten særlig forstyrrelser for brukeren. Vår logikk for å holde brukergrensesnittet oppdatert var enkel. For hver spørring til backend om å endre data ble det også sendt en forespørsel om å hente all relevant data. Dette gjorde at brukergrensesnittet ble oppdatert ved hver brukerhandling som endret dataen. Problemet med denne løsningen er at den ikke tar hensyn til andre brukere sine handlinger. Altså, brukergrensesnittet blir kun oppdatert ved handlinger brukeren gjør selv, ikke andres handlinger.

Et annet problem var måten vi utførte spørringer. I prospekt-siden ble det sendt en spørring for alle selgeres prospekter. Fra denne dataen ble komponenter i kalendersystemet tegnet. Hver gang vi hentet samme data på nytt reagerte React som om all dataen var endret. Når React-komponenter mottar ny data gjennom props blir de tegnet på nytt. Komponenter som brukte deler av denne dataen kunne ikke skille mellom endring i egen data og endring i data generelt. Når en komponent tegner seg selv på nytt blir handlinger på komponenten avbrutt. Dette er et problem siden handlinger bør kun forstyrres dersom dataen handlingen skal utføres på ikke lengre er nøyaktig.

Overgangen til GraphQL fra REST gjorde det enklere å støtte sanntid samarbeid. I overgangen til GraphQL tok vi i bruk verktøyet Apollo Client. Apollo Client har to måter man kan holde data synkronisert; polling og refetching. Polling kjører spørringer i gitte tidsintervall. Refetching kjører spørringer etter brukere utfører handlinger (ApolloDocs, 2021). Vi valgte først å ta i bruk polling for våre spørringer, da det er den letteste måten å holde data fra spørringer synkronisert. Dette løste problemet med at andre brukers handlinger ikke ble umiddelbart synlig for andre brukere. Likevel gjensto problemet at vi gjorde store spørringer som tvang React til å regne nesten alle komponenter på nytt. Det var heller ikke ønskelig å gjenta store spørringer veldig ofte uten god begrunnelse.

Til slutt landet vi på en løsning som utnytter både refetching og polling for å synkronisere brukergrensesnittet. Det første, og kanskje viktigste, var at vi delte den store spørringer opp i flere små spørringer på tvers av komponenter. Et eksempel er at hver selger har en egen komponent som kun hentes den selgerens prospekter. Antall spørringer går betydelig opp med denne løsningen. Vi valgte å følge rådet om "Colocation" fra GraphQL sin nettside (Savona, J, 2021). Dette innebærer å plassere spørringer, i vårt tilfelle *useQuery* og *useMutation* hooks, nærme komponentene som skal bruke dem. Dette løser vårt problem der komponenter tegner seg selv på nytt selv om deres data ikke har endret seg. Med denne nye løsningen vil den kun tegne seg selv på nytt dersom egen data endrer seg. Vi satt pollingen til å være hvert tredje sekund, da vi mente refetching kunne dekke resten av synkroniseringen. Neste avsnitt beskriver hvordan vi brukte refetching. Koden i bildet under viser hvordan komponenten *ConsultantContractList* henter dataen den trenger selv.

```
export const ConsultantContractList: React.FC<ConsultantContractListProps> = ({ consultantId }) => {
    const { loading, error, data } = useQuery<GetConsultantContractsPayload, GetConsultantContractsInput>(
        GET_CONSULTANT_CONTRACTS,
        {
            variables: { id: consultantId },
            pollInterval: 3000,
        }
    );
```

fig 27: Komponenten ConsultantContractList sin useQuery hook

Vi valgte å utnytte refetching til å sikre at handlinger brukere utfører reflekteres i brukergrensesnittet umiddelbart. Dette ble gjort gjennom et parameter som *useMutation* hooks mottar; *refetchQueries*. *RefetchQueries* lar deg spesifisere hvilke queries som skal kjøres etter mutation er utført, og med hvilke parametere de skal kjøres. Vi valgte å inkludere en refetch for all data som blir påvirket av mutasjonen. Dette gjøres gjennom å inkludere ID i spørringene i *refetchQueries*.

```
const [addContract] = useMutation<AddContractPayload, { input: AddContractInput }>(ADD_CONTRACT, {
    refetchQueries: [
        {
            query: GET_CONSULTANT_CONTRACTS,
            variables: { id: consultantId },
        },
        {
            query: GET_CONSULTANT_CAPACITY,
            variables: { startYear: currentYear, endYear: currentYear + 2, id: consultantId },
        },
    ],
    awaitRefetchQueries: true,
});
```

fig 28: Eksempel på refetchQueries til mutasjon for å legge til kontrakt

Bildet over viser spørringene som kjører etter mutasjonen for å legge til en kontrakt er utført. Både kontraktene og kapasiteten til konsulenten som mottar den nye kontrakten vil bli påvirket av mutasjonen. Siden vi inkluderer konsulentens ID i spørringen vil kun komponenter avhengig av denne konsulentens data tegne seg selv på nytt.

Vi endte med en implementering som tillater uavbrutt sanntid samarbeid gjennom en rimelig mengde spørringer. Vi valgte å prioritere synkronisering i seksjonene av brukergrensesnittet som brukeren utfører handlinger på. Seksjoner som brukeren ikke interagerer med holder seg rimelig synkroniserte gjennom polling hvert tredje sekund. Sammen gir dette brukeren muligheten til å samarbeide med andre brukere uten at brukergrensesnittet mister sin nøyaktighet.

5.7 Innstillinger

Utviklingen av innstillingssiden startet i sprint 4. Siden skulle håndtere oppretting og sletting av konsulenter og selgere. Dette var funksjonalitet systemet trengte dersom det skulle håndtere endring i selgere og konsulenter. Valget om å inkludere en side for instillinger ble gjort sammen med produkteier. Vi ble enige om at siden kunne regnes som en “nice-to-have”, da det ikke er strengt tatt nødvendig for kjernefunksjonaliteten.

En kilde til utfordringer var at både konsulenter og selgere har et forelder-forhold til andre objekter i databasen. Selgere har prospekter som har sub prospekter. Konsulenter har kontrakter. Dette gjør at man bør være klar over konsekvensene av å slette objektene. Planen var å ha mulighet for å flytte prospekter fra en selger til en annen ved sletting, men det fikk vi ikke implementert. Vi valgte i stedet å inkludere antall “barn” hvert objekt har når det forsøkes å slette dem.

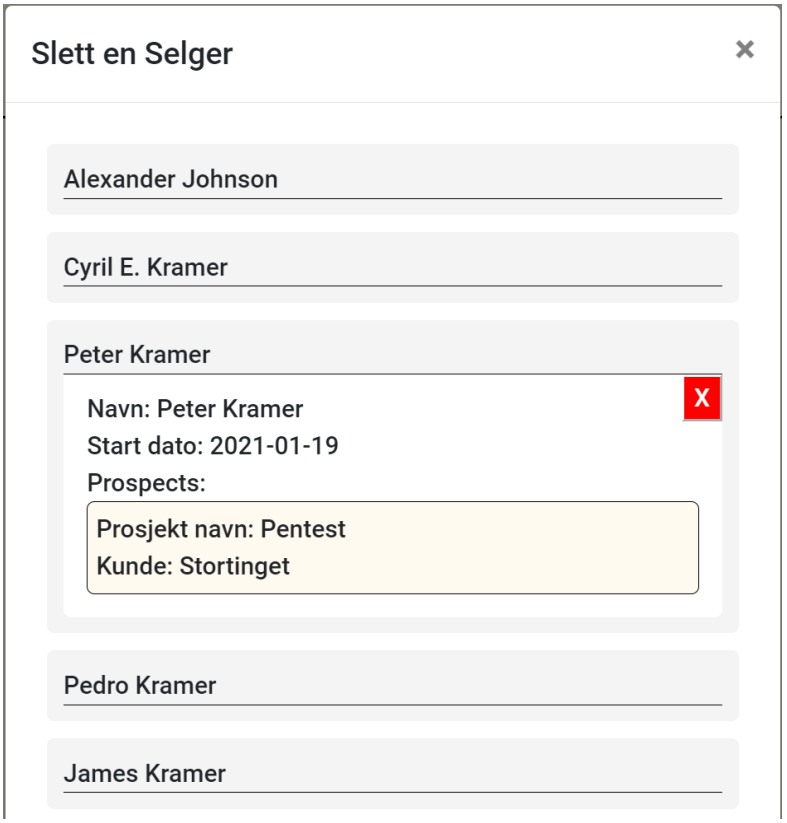


fig 29: Skjermbilde av modalen for å slette selgere

En del tid gikk til å prøve å redusere henting av data vi allerede har hentet. Ved sletting ville vi helst slippe å hente alt på nytt. Likevel, ville vi sikre at listen av konsulenter/selgere stemte overens med dataen i backend. Vi prøvde å finne en løsning der vi kun hentet nødvendig data på nytt, men dette viste seg å være tilsvarende umulig. Dette er fordi man kan ikke vite om lokal data stemmer med backend uten å sammenligne dem, noe som krever å hente nyeste versjon av dataen. Vi endte opp med å hente alt på nytt etter mye innsats dedikert til å unngå det.

Vi tok i bruk modaler med skjemaer for oppretting av selgere og konsulenter. Skjemaene er rimelig normale, der hvert felt har et eget input element i skjemaet. Vi implementere grunnleggende validering for å hindre at ugyldige verdier blir sendt til backend. Backend implementerte også logikk for å ignorere alle ugyldige requests.

Vi ble ikke helt ferdig med innstillingene. Dette skyldes at vi måtte skifte fokus til nøkkeltall i de siste sprintene. I ettertid ser det ut som om vi kan ha overkomplisert implementeringen, da mye tid gikk til å optimalisere en spørring som ikke skapte seriøse problemer.

5.8 Nøkkeltall

Vi begynte å utvikle siden for nøkkeltall i sprint 8. Implementeringen av nøkkeltall bød på flere utfordringer. Som beskrevet i kapittel 2.1.4 skulle nøkkeltall være en side som gir

oversikt over de viktigste tallene for avdelingen. I dagens løsning vises nøkkeltall gjennom tabeller. Vi innså tidlig at den største utfordringen med å vise nøkkeltall er å skape en god layout. På grunn av dette valgte vi å vente med utvikling til produkteieren godkjente en av våre mockups.

Det første som ble slått fast var at kategoriene omsetning og resultat skulle vises som grafer. Måten tallene for forecast, PåKontrakt, ansatte og timepris skulle visualiseres på ble diskutert flere ganger. Det sto her mellom tabeller eller grafer. De fleste tabeller og grafer skulle vise maksimum seks måneder tilbake i tid. Det skulle også være mulig å endre tidsperioden du ser data for, både året og måneden.

Omsetning og resultat hadde tre viktige tall hver; budsjett, faktisk og avvik. I tillegg skulle både månedsomsetning og resultat vises akkumulert. Vi ble først enige i at akkumulerte tall skulle regnes ut i backend og kunne hentes via API. Etterhvert viste det seg å være vanskelig å få til. Vi løste problemet med å gjøre den akkumulerte utregningen i frontend. Det var en utfordring, men de akkumulerte grafene funket som de skulle. Likevel førte dette til en del utestet logikk i frontend.

Forecast og PåKontrakt var først fremstilt som to tabeller, men etterhvert ombestemte produkteieren seg og vi endret dem til grafer. Vi utviklet noen få unit tester for nøkkeltall-logikken i frontend for å sikre at funksjonaliteten ble som forventet i videreutvikling og eventuell refactoring. Til slutt, etter mange iterasjoner, endte alle tallene opp som grafer.

Alle grafene ble utviklet ved hjelp av javascript-biblioteket CanvasJS. Dette biblioteket har komponenter med grafer som er enkle å implementere. Vi brukte to graftyper; Range Area Chart og Dashed Line Chart. Ranged Area Chart ble brukt for å visualisere omsetning, resultat, forecast og PåKontrakt. Dashed Line Chart ble brukt til timepris og antall ansatte. Vi valgte å bruke Range Area Chart grunnet behovet for å vise avvik. Timepris og antall ansatte hadde ikke dette behovet som gjorde at vi valgte å bruke en annen type graf.

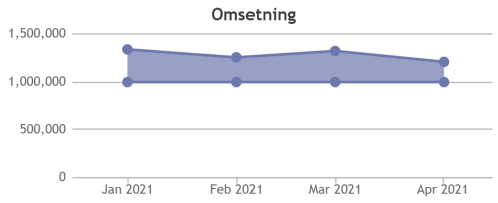


fig 30: Range Area Chart

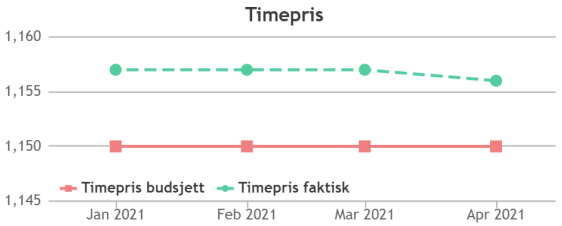


fig 31: Dashed Line Chart

Første versjon av layout brukte en Accordion komponent fra Reactstrap biblioteket. Accordions er en trekkspill-lignende måte for å vise og skjule informasjon. Komponenten skjuler sitt innhold med mindre den trykkes på. Vi gjorde det slik at man kunne se en beskrivelse av tallet det gjaldt, f.eks omsetning, og ved musetrykk vises grafen for det tallet. Dette tenkte vi kunne være nyttig for å komprimere informasjonen slik at man slapp å bla ned

på siden og lete etter informasjonen. Denne planen ble likevel raskt skrotet da produkteier ikke var fornøyd. Produkteieren likte ikke at det ble introdusert et ekstra steg i prosessen av å finne informasjon, nemlig at man må trykke. Det var heller ikke ønskelig at det ikke var plass til å se alle grafene samtidig. Det ble da heller bestemt at vi skulle gjøre alle grafene mindre og vise all informasjonen på en gang uten å måtte trykke på noe eller bla ned på siden.

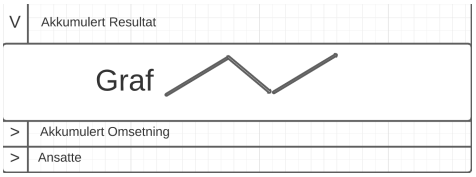


fig 32: mockup av nøkkeltall med bruk av accordion

For å få plass til grafene på samme side brukte vi en kombinasjon av CSS Grid og VW(view-width). View-width i CSS lar deg spesifisere bredde relativt til skjermens størrelse. For å støtte valg av måned og år brukte vi vanlige inputs. Verdiene til disse inputs ble brukt som parametere i API kallet for å hente nøkkeltall. Videre ble resultatet fra API kallet sendt som props til alle grafene. Nøkkeltall ble vist for maks seks måneder av gangen etter produkteierens ønske. Dersom det ikke var nok data for det inneværende året, ble kun månedene vi hadde data for vist.

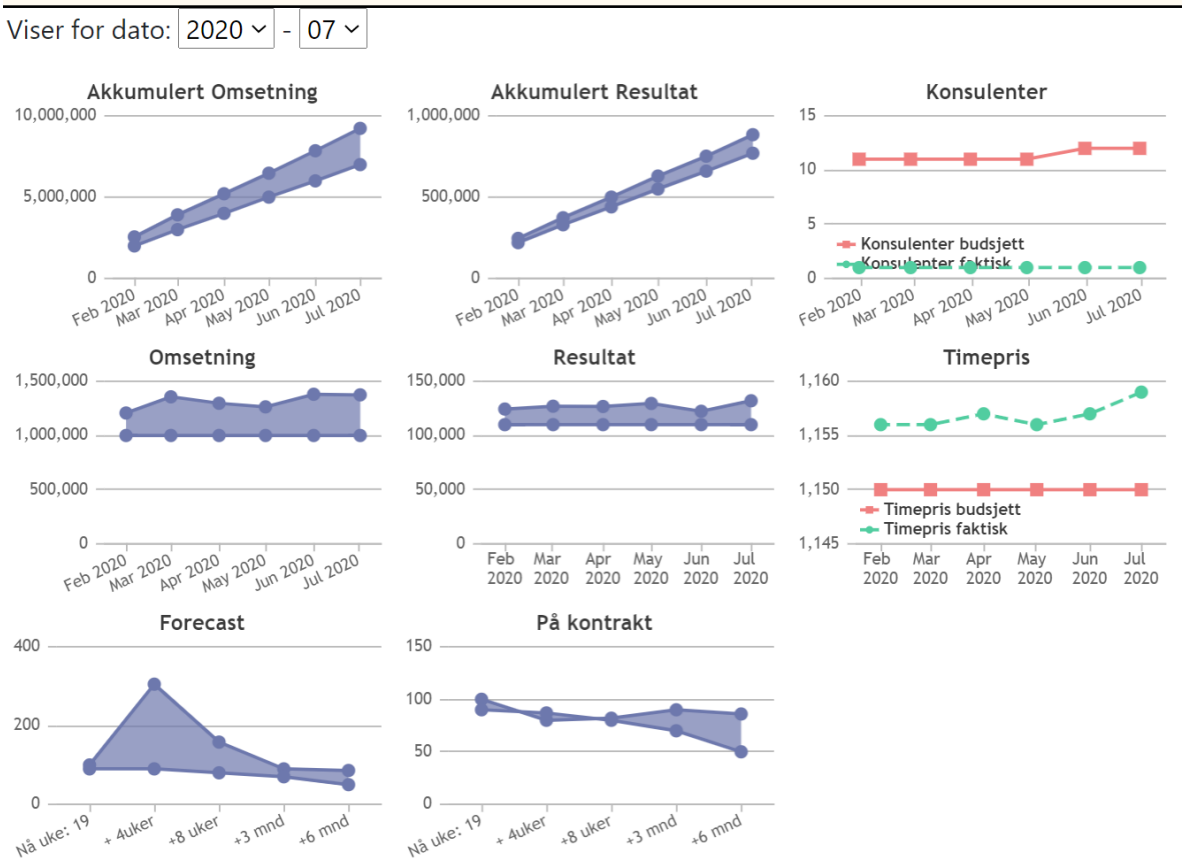


fig 33: Nøkkeltall som vises for et valgt periode

Det var mye kodeduplisering i koden logikken til layout og grafene. Dette skyldtes at vi prioriterte å få demoer raskt klare til produkteieren. Når produkteieren hadde sagt seg fornøyd med både grafer og layouts prioriterte vi å rydde opp i koden. Mye av refactoringen gjaldt å gjøre logikken gjenbrukbar for grafene til omsetning, akkumulert omsetning, resultat og akkumulert resultat grafene. Dette sparte oss for flere hundre linjer av kode. Vi ble senere rådet av produkteier at utregningen av akkumulerte tall burde skje i backend. Vi endte opp med å implementere logikken i backend og slette all akkumulert logikk i frontend.

De to siste sprintene ble funksjonalitet prioritert og vi gikk over på å implementere API kall og datahåndtering for andre grafer. Vi fikk laget grafer for alle nøkkeltall produkteier ønsket å se. Logikken bak grafene inneholder fremdeles mange bugs som må fikses.

6 Refleksjon

I dette kapitlet reflekterer vi over hvordan prosjektet gikk. Vi fulgte agil metodikk gjennom hele systemutviklingsprosessen. I løpet av prosjektet opplevde vi gevinster og problemer med vår planlagte metodikk for gjennomførelse. I de neste kapitlene utforsker vi lærepengene langs veien, hva som gikk bra og til slutt tiltakene vi tok for å holde oss agile.

6.1 Lærepenger

Gjennom å la våre planer, antagelser og prioriteringer møte virkeligheten fikk vi flere lærepenger. De neste kapitlene beskriver hvilke lærepenger vi fikk og endringene vi gjorde med kunnskapen de ga oss. For problemene vi ikke klarte å løse, beskriver vi hva vi ville gjort annerledes i neste prosjekt.

6.1.1 Tidsestimering

Vi nedprioriterte tidsestimering i starten av prosjektet. Årsaken er at vi mente våre estimater ville vært veldig unøyaktig siden vi ikke kunne språkene vi brukte. Argumentet er at det ville vært sløsing av tid og energi tidlig i prosjektet. Denne nedprioriteringen hadde noen konsekvenser. Blant dem, så er en av dem at vi ikke tidlig etablerte vaner rundt å alltid estimere tid for arbeidsoppgaver. Dette gjorde at vi slet med overgangen til å alltid estimere tid. En annen konsekvens er at vi hadde sprints der arbeidsmengden vi valgte var betydelig større enn vi kunne rekke på en sprint. Det var etter slike sprints vi begynte å være nøyere på tidsestimering. Hvis vi skulle begynt på nytt, ville vi ha krevd tidsestimering fra alle medlemmer fra starten. Selv om estimatene ville vært unøyaktige, så er det en god vane å ha som gruppe for senere deler av utviklingsprosessen.

6.1.2 Sprint Planning

Ved oppstart definerte vi arbeidsoppgaver etter sprinter hadde startet. Dette ga oss få gevinster og mange problemer. Gevinsten var at arbeidsoppgavene ble laget med en klar forståelse av problemet som skulle løses. Det største problemet var at det oppsto enorme

arbeidsmengder i hver sprint. Et kjennetegn på slike problemer er at burndown-chart går oppover.

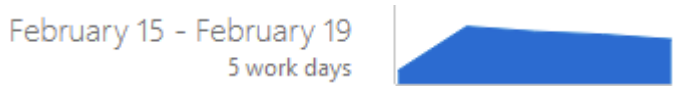


fig 34: Burndown chart i sprint 4

Som bildet over viser gikk vår burndown-chart feil retning i sprint 4. Ved sprint 6 bestemte vi oss for å begynne med Sprint Planning. Målet var å skape utførbare arbeidsmengder. bedre planlegging ved starten av sprints. I møtene definerte vi arbeidsoppgaver til beste evne og ga dem et tidsestimat. Dette valget om å innføre struktur, noe som potensielt reduserer vår agilitet, viste seg å være verdt det. Under er et bilde av burndown-chart etter vi begynte med Sprint Planning i sprint 6.

fig 35: Burndown chart i sprint 6

6.1.3 Strategi for læring

Vi valgte å gjennomføre læringen av språk og verktøy individuelt uten særlig samarbeid. Dette ga oss utvilsomt bedre evner til å selvstendig anskaffe teknisk kunnskap. Likevel, mener vi dette var en feil. Vår strategi for læring var avhengig av viljestyrke og motivasjon, noe som utvilsomt varierer i enhver gruppe. Denne strategien førte til at kompetansenivået i gruppen ble betydelig ujevnt i gruppen. Vi tror en kombinasjon av selvstendig læring, deling av kunnskap og parprogrammering kunne ha løst dette problemet. Hvis vi kunne begynt prosjektet på nytt, ville vi tatt tiltak for å sikre at alle lærer seg språkene.

6.1.4 Ansvarsområder / Oppgavefordeling

Vi lot våre utviklere ha egne ansvarsområder. Dette hadde variert suksess. Utviklerne som mestret sine verktøy og språk opplevde høy effektivitet. Utviklerne som fremdeles var tidlig i læringsfasen av sine verktøy og språk slet med å oppnå like høy effektivitet. De sistnevnte utviklerne lærte også verktøyene og språkene i et lavere tempo, der mangel på hjelp og støtte var en bidragende årsak. Dersom vi skulle startet prosjektet på nytt, ville vi både økt kommunikasjonen mellom forskjellige ansvarsområder og lagt vekt på å støtte at utviklere lærer nok til jobbe selvstendig.

6.1.5 Prosjektstyring

Vi beholdt våre roller i gruppen gjennom hele prosjektet. Ved starten av prosjektet definerte vi krav og regler i en gruppekontrakt. Prosjektleder og scrum master mislyktes i å sikre at reglene og kravene ble fulgt av gruppen. Det er verdt å nevne at de fleste regler og krav ble fulgt av gruppen, men da ikke som en konsekvens av ledelsens tiltak. Kravet om å jobbe minst 2 timer hver ukedag ble brutt mest. Vi tror dette kravet var det mest krevende å oppfylle, men samtidig det viktigste.

I ettertid innser vi at ledelsen burde vært strengere på å sikre at regelbryting ikke gjentar seg. Dette innebærer å sette seg inn i årsakene bak at regler ble brutt og følge opp medlemmene det gjelder slik samme årsak ikke oppstår igjen.

6.1.6 Oppsummering

Ved oppstart måtte språk og verktøy læres, ferdigheter forbedres og systemet forstås. Dette gjorde at vi prioriterte, planla og antok med utgangspunkt i at omstendighetene ville forbli kaotiske. Dette utgangspunktet tok ikke hensyn til at vi etterhvert ville forstå våre språk, klargjøre kravene til systemet og danne de nødvendige ferdighetene til å utføre prosjektet. Etterhvert som vi forstod problemene med vårt utgangspunkt forsøkte vi å finne bedre løsninger, med varierende suksess.

6.2 Hva som gikk bra

Tross mange lærepenger og utfordringer underveis i prosjektet, så er det mye som også har gått bra. De neste avsnittene skal beskrive hvilke aspekter ved prosjektet vi er fornøyde med og hvorfor.

Det første vi vil trekke frem er prosjektstyring. Kombinasjonen av Sprint Planning, Daily Scrum og Sprint Review fungerte godt. I våre Sprint Planning-møter definerte vi utførbare stykker arbeid i retningen produkteier beskrev i forrige Sprint Review-møte. Vi skrev referater i hvert Sprint Review-møte, noe som viste seg å være til stor nytte i planleggingen av neste sprint. Alle medlemmer møtte opp til Daily Scrum hver ukedag gjennom hele prosjektet, der hvert møte fulgte samme struktur. Dette sikret jevnt tempo og regelmessig kommunikasjon innad i gruppen. Vi utførte hvert Sprint Review-møte sammen med produkteier i slutten av uken. Vi mener prosjektstyringen gikk bra fordi vi klarte å jobbe i et jevnt tempo mens vi vedlikeholdt trivsel.

Grunnet corona-pandemien ble hele prosjektet gjennomført uten fysiske møter, hverken med oss studenter innad, eller med produkteier og veileder. Dette er noe vi mener vi har taklet på en god måte. Ved å bruke kommunikasjonsplattformer som Discord, Slack og Google Meet, har vi klart å holde kommunikasjonen og samhandlingen på et høyt nivå.

Våre tiltak for kvalitetssikring er også noe vi vil trekke frem som fungerte godt. Ved å utføre tiltakene som er beskrevet i punkt 3.3.2 klarte vi å sikre produktkvaliteten. Selv om dette reduserte tempoet på utviklingen noe, klarte vi å sikre at sluttproduktet ble som ønsket. Tiltak for å sikre prosesskvalitet fungerte godt, men våre tiltak imot social loafing hadde begrenset suksess.

Vi er fornøyde med måten vi håndterte kontinuerlige endringer i krav. Nye krav kom ofte i våre Sprint Reviews med produkteieren. Basert på referat fra møtene ble de nye kravene omdannet til oppgaver i vår Sprint Planning. Størrelsen og kompleksiteten på oppgavene vi

opprettet ble holdt rimelig balansert gjennom sprintene. Den aktuelle endringen var som regel fullført til neste Sprint Review-møte.

Vi er også fornøyde med produktet. Vi oppfylte målet om å implementere all grunnfunksjonalitet i hver side. Utenom dette klarte vi å forenkle utførelsen av mange arbeidsoppgaver. Vi reduserte også mengden unødvendig informasjon som vises i brukergrensesnittet. All funksjonalitet ble implementert i samråd med produkteier for å sikre at det stemte overens med hans ønsker.

6.3 Hvordan vi holdt oss agile

Det finnes mange måter å definere agilitet. Vi går ut ifra at det betyr at man har et fokus på utførelse, tilpasser seg til endringer og baserer planlegging på tilbakemelding fra kunder/produkteier.

Våre omstendigheter var ideelle for agil utvikling. Dette er fordi kravene til design, brukergrensesnitt og funksjonalitet var under endring gjennom hele prosjektet. Under slike omstendigheter bør utviklingen kunne tilpasses til endrede krav. Det er derfor vi valgte å bruke prinsipper fra Scrum metodikken.

Vi sikret et fokus på utførelse ved å planlegge etter behov. Vårt mål var å skape mest mulig rom for utvikling mens vi beholdt nødvendig struktur. Vi tilpasset vår planlegging etterhvert som vi forstod våre behov for planlegging bedre. Ved oppstarten varte daglige møter 30 minutter. Vi reduserte denne tiden til 7 minutter ved å orientere all diskusjon rundt de tre obligatoriske spørsmålene. Vår planlegging for sprints fulgte ingen struktur, som førte til at innsats ble spredt utover mange brukerhistorier. Dette løste vi med å opprette, estimere og fordele oppgaver ved starten av hver sprint.

Vi brukte Sprint Planning for å sikre at arbeidsoppgaver reflekterte tilbakemeldingen fra produkteier. I møtene ble oppgaver opprettet etter referat fra forrige møte med produkteieren. Slik planlegging omdanner tilbakemelding til arbeidsoppgaver, som indirekte tvinger oss til å tilpasse oss.

7 Uttalelse fra produkteier

Studentene Daniel Lindalen, Gustav Eikaas, Mantas Karulaitis og Tomas Ryen har i perioden fra Januar til Mai 2021 gjennomført et utviklingsprosjekt hos Webstep. Studentene tok kontakt med oss i november 2020. Etter intervju med studentene ble det bestemt at de skulle få lov til å gjennomføre bachelorprosjektet deres hos Webstep.

Utviklingsprosjektet gikk ut på å lage en web applikasjon som skal erstatte den eksisterende regneark-løsningen hvor selgerne i Webstep utfører arbeidsoppgavene sine. Vi har ønsket at studentene skulle være kreative og finne nye løsninger for å gjøre det eksisterende systemet mer smidig. Studentene valgte å ta i bruk flere teknologier som var nye for dem, men relevant for oss og IT-bransjen i Kristiansands-området generelt.

Studentene har i denne sammenheng utvist en særlig god evne til å tilegne seg ny kunnskap og har jobbet systematisk og bra mot målet.

Webstep har også flere ganger i løpet av prosjektet endret prioriteringer og innholdet i produktet, noe som studentene har taklet på en god måte. Vi tror de har kjent litt på hvordan et prosjekt kan fortone seg når man begynner i et profesjonelt firma.

Prosjektet er kjørt i henhold til en tilpasset SCRUM metodikk. Studentene har også evnet å tilpasse SCRUM til deres prosjekt, både med tanke på størrelse og de forskjellige teknikker de har brukt. Vårt inntrykk er at de også har fulgt dette modifiserte opplegget bra.

På grunn av Covid-19 har ikke studentene hatt mulighet til å jobbe i Websteps lokaler, men vi har likevel klart å opprettholde god kontakt via Slack og ved faste digitale møter på slutten av hver uke.

Webstep er veldig fornøyd med gruppens innsats og måten de har fremstått på i forhold til oss som produkteier. Vi vil ønske studentene lykke til videre og takke for et veldig bra samarbeid.

Med vennlig hilsen

Underskrift

DocuSigned by:

F84788B7A7964E5...

Dato

07-May-2021

Fig 36 Uttalelse fra produkteier

8 Selvevaluering

I dette kapitlet forklarer hvert medlem deres rolle og bidrag i bachelorprosjektet.

8.1 Daniel

Jeg var Scrum Master samt hovedansvarlig for prospekt-siden og belegg-siden. For å utvikle disse sidene måtte jeg lære meg React og TypeScript. For å skape samarbeid med backend måtte jeg også lære grunnleggende GraphQL og verktøyet Apollo Client.

Jeg utviklet kalendersystemet og endringene nødvendig for overgangen til inline-edit mønsteret. Jeg implementerte Google Login og navigasjonsbaren. Jeg var også ansvarlig for filstrukturen i frontend, rydding av kode og definisjon av god “kode skikk” i React.

Som scrum master hadde jeg begrenset suksess. Jeg holdt Daily Scrum møtene fokuserte gjennom tiltak for å fjerne uproduktiv tid. Jeg tok ansvar for kommunikasjon med veileder. Jeg la ut ressurser om temaer FE utviklerne slet med, som steg-for-steg instrukser og linker til videoer. Jeg oppfordret riktig bruk av DevOps slik at arbeidet skulle holde seg strukturert. Jeg innførte Sprint Planning etter utviklerne opplevde at arbeidet ble ustrukturert og “scope” var ute av kontroll.

Jeg mislykkes i å få alle til å føre sine timer, estimere arbeidsoppgaver, flytte arbeidsoppgaver mellom new-doing-done og jobbe jevnt 2 timer i ukedagene. Jeg kunne også lagt opp til mer kunnskapsflyt mellom utviklerne for å jevne ut lærings tempoet. Daily Scrums alene var ikke nok til å forstå problemstillingene andre utviklere slet med. Jeg burde tatt tiltak for å øke samarbeid mellom utviklerne. Jeg begrunner dette med at det ved flere anledninger viste seg å være “enkelt” å løse noen problemstillinger etter noen få minutters veiledning.

8.2 Gustav

Jeg var prosjektleder samt hadde jeg ansvar for utvikling av hele backend. For å utvikle backend måtte jeg lære meg .Net Core og noe T-SQL. For å utvikle API'et måtte jeg lære REST og GraphQL. Jeg prøvde å holde åpen kommunikasjon med frontend rundt bruk av API og imøtekomme deres behov.

Som prosjektleder hadde jeg god kontakt med produkteier og resten av gruppen. Prosjektet gikk relativt bra men det kunne gått bedre. Prosjektleder og Scrum master hadde noe overlappende ansvar, noe som ikke var problematisk. Som prosjektleder gikk mest av min tid til planlegging og holde oversikt over fremgang, problemer og deadlines.

Jeg er veldig fornøyd med valget mitt om å gå over til GraphQL, selv om det betydde at all koden relatert til REST måtte fjernes. GraphQL gjorde det lettere for både frontend og backend. HotChocolate var et interessant rammeverk å jobbe med da det ikke var helt ferdig utviklet, dette bød på en del utfordringer og rare bugs. I løpet av denne bacheloroppgaven lærte jeg nok .Net Core til å få jobb som utvikler.

8.3 Mantas

Jeg begynte i backend, men gikk etterhvert over til frontend hvor jeg jobbet mest. Jeg utviklet og designet innstillinger og en stor del av nøkkeltall. Jeg begynte å lære meg .Net Core og T-SQL siden jeg skulle starte i backend, men på grunn av mangel av tasks i backend, gikk jeg over til frontend. Derfor begynte jeg å lære meg React og Typescript litt senere enn resten av frontend. Dette førte til at to ekstra sprints gikk til læring før jeg hadde kunnskap til å bidra i frontend utviklingen. Mens jeg lærte meg techstacken, jobbet jeg med design i Figma for web

applikasjonen, som vi aldri tok i bruk, på grunn rask fremgang i prospekter og kravene til produkteieren.

Jeg tok ikke timeføringen eller lukking av fullførte arbeidsoppgavene like seriøst som selve arbeidet. Jeg tenkte at det var det endelige resultatet som teller, ikke antall lukket arbeidsoppgaver i DevOps. Jeg mener at hele DevOps i seg selv var vrient å bruke og vanskelig å navigere av og til, men jeg forstår viktigheten av DevOps. Både timeføring og fullførte arbeidsoppgaver er viktig, spesielt i arbeidslivet, ettersom arbeidsgivere betaler for timene brukt på forskjellige arbeidsoppgaver. Ofte glemte jeg å føre timene brukt og å lukke fullførte arbeidsoppgaver. Det var heller ikke bestemt i gruppa at antall lukket arbeidsoppgaver skulle vises frem, derfor så jeg ikke viktigheten i det.

Jeg er mest fornøyd av utviklingen utført i nøkkeltall siden, hvor jeg skrev både logikken for akkumulerte verdier og grunnleggende logikk som ble brukt for alle nøkkeltall, samt css som endret etter skjermstørrelsen. Når akkumulasjons-logikken ble flyttet til backend, kunne jeg gjenbruke enda mer logikk for visning og håndtering av graf data.

Jeg er minst fornøyd av innstillinger-siden hvor jeg overkompliserte et problem og endte opp med å bruke mye tid på å løse noe som kunne blitt utført enkelt. Utviklingen av innstillinger startet rett etter jeg var ferdig med mine lærings sprints, dette kan ha vært ett av grunnene til overkoplisering av problemet. Det var mye jeg ikke visste eller kunne som jeg hadde lært på slutten av prosjektet.

8.4 Tomas

Min rolle i bachelorprosjektet har vært frontend-utvikler. Frontend-utvikling var noe jeg ikke hadde erfaring med fra tidligere. Jeg måtte derfor lære meg Typescript og React, friske opp kunnskapen i CSS, samt sette meg inn i GraphQL og Apollo Client for å kunne samarbeide med backend.

I begynnelsen av bachelorprosjektet fikk jeg hovedansvaret for å utvikle belegg-siden. Utviklingen av denne siden ble imidlertid satt på vent da produkteier ville se prospekt-siden ferdig først. Min oppgave ble da å bistå i utviklingen av prospect-siden. På denne siden implementerte jeg blant annet toast notifikasjoner, bekreftelses-modal for sletting, visuell feedback for handlinger, samt at jeg bidro med utviklingen av logikken for datohåndtering.

I sprint 8 da prospect-siden ble erklært ferdig-utviklet av produkteier, fikk jeg ansvar sammen med Mantas å utvikle siden for nøkkeltall. Ellers har jeg hatt ansvar for å skrive referat under Sprint Review-møtene og veiledningsmøtene.

Bachelorprosjektet har vært veldig lærerikt og jeg er fornøyd med å ha fått kjennskap til nye utviklings-teknologier. Det har vært nyttig å få kjenne på hvordan et prosjekt kan fortone seg når man jobber sammen med et profesjonelt firma.

9 Referanser

ApolloDocs. (2021). Queries. Hentet fra https://www.apollographql.com/docs/react/data/queries/

BugZilla. (2021). Incorrect handling of onmouseover event while left mouse button is down. Actual for divs with "overflow:hidden" style. Hentet fra https://bugzilla.mozilla.org/show_bug.cgi?id=620513

PatternFly. (2021). Inline Edit. Hentet fra https://www.patternfly.org/v3/pattern-library/forms-and-controls/inline-edit/

ReactJS. (2021). Render Props - React. Hentet fra https://reactjs.org/docs/render-props.html

ReactPatterns. (2021). Container component. Hentet fra https://reactpatterns.com/#container-component

Savona, J. (2021). Best Practices for GraphQL Clients. Hentet fra https://www.graphql.com/articles/best-practices-for-graphql-clients

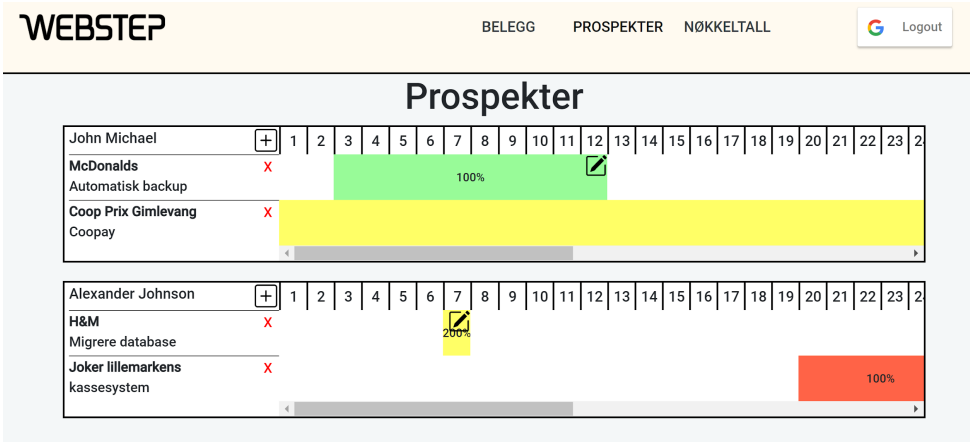
10 Appendix

10.1 Design iterasjoner

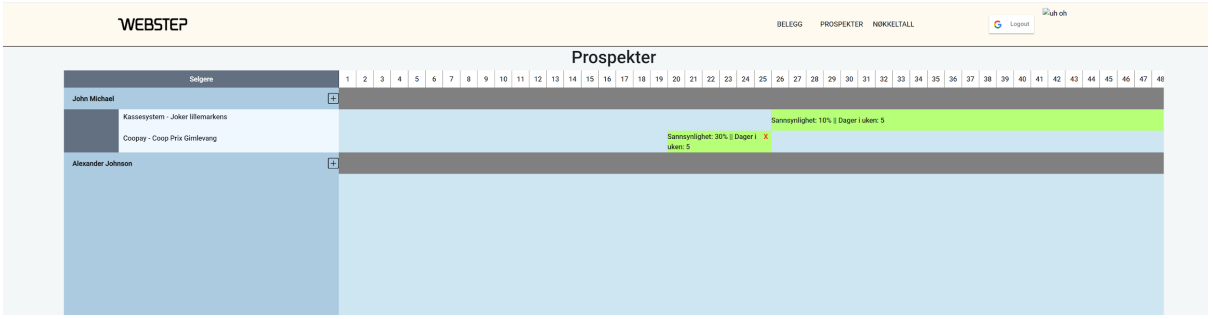
Dette kapitlet inneholder bilder av iterasjonene designet til sidene gikk gjennom.

Prospekt-siden

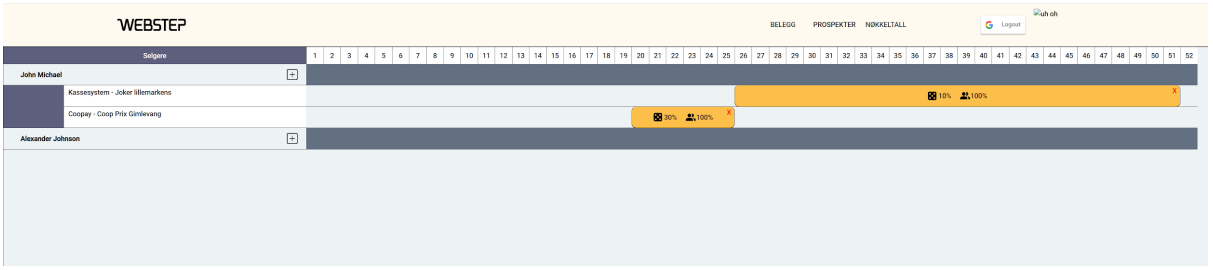
Første iterasjon



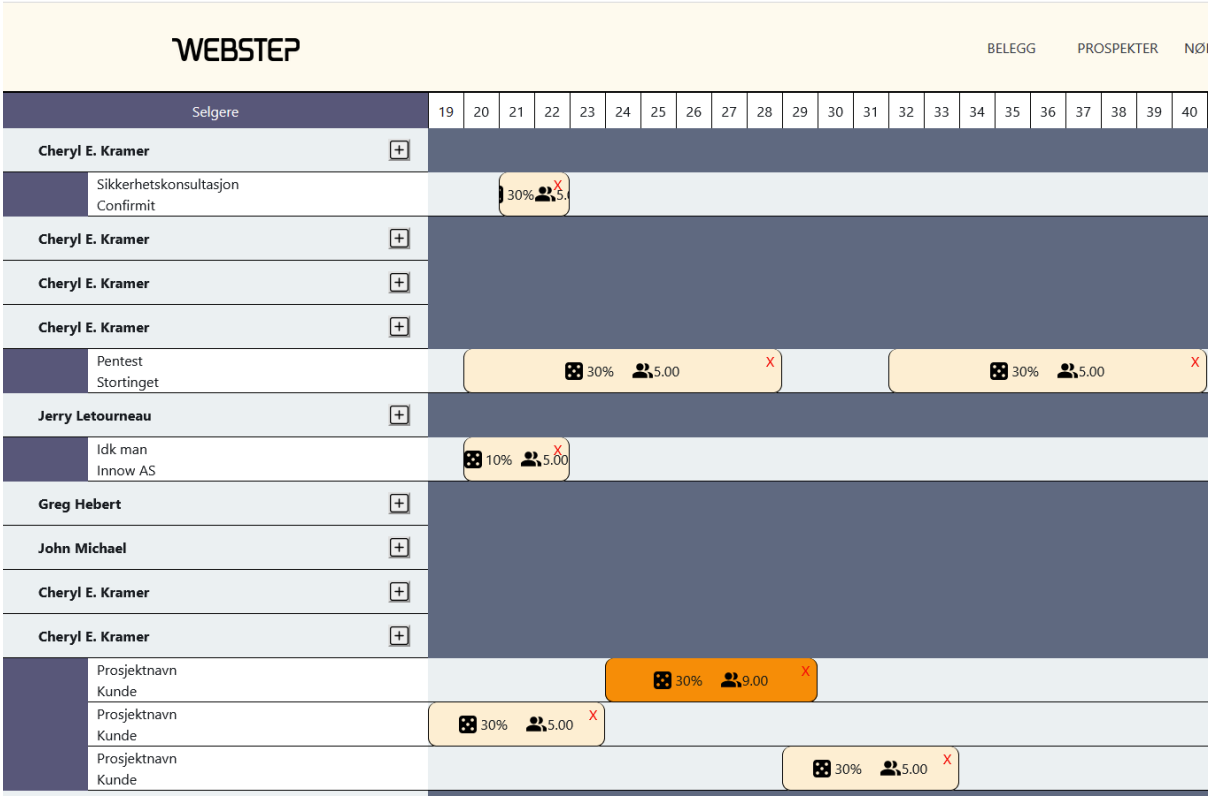
Andre iterasjon



Tredje iterasjon



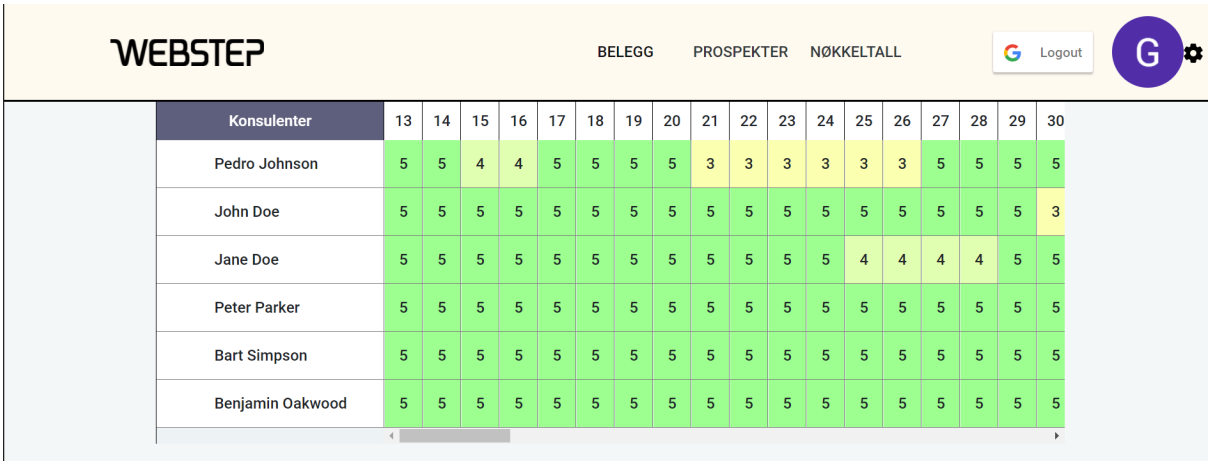
Fjerde iterasjon



Belegg-siden

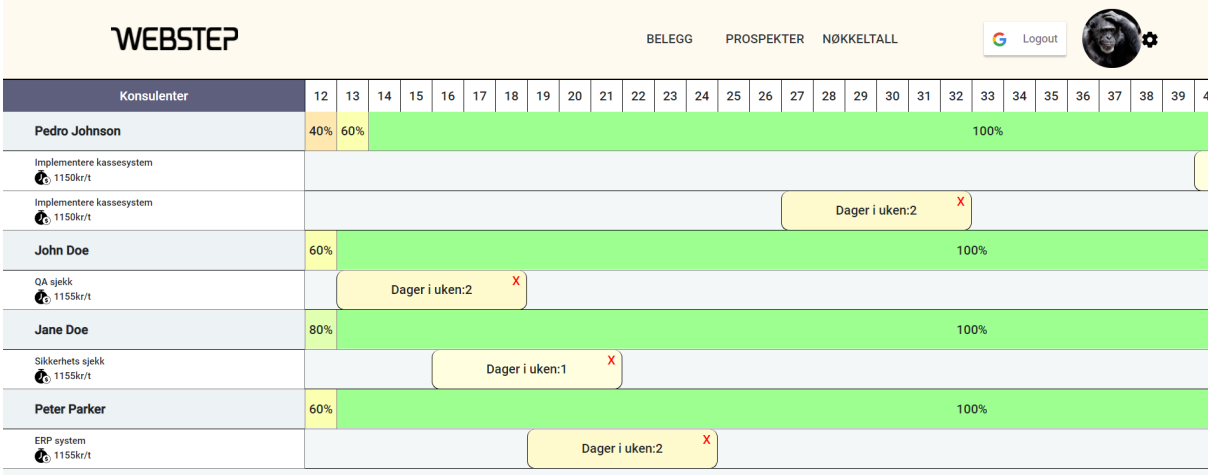
Første iterasjon

Første versjon av belegg-siden. Viser antall ledige dager per uke for hver konsulent.

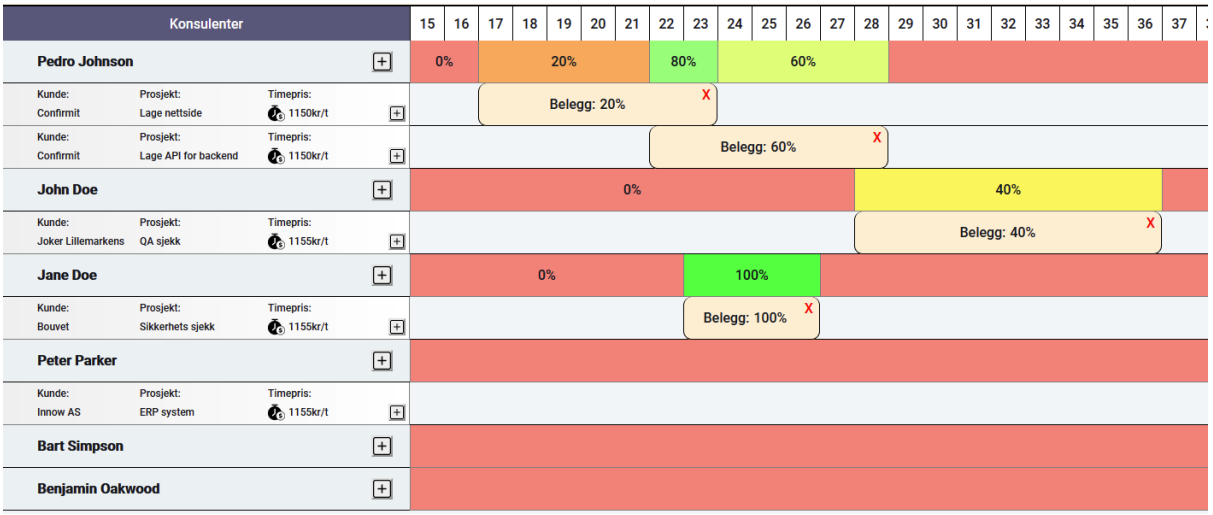


Konsulenter	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Pedro Johnson	5	5	4	4	5	5	5	5	3	3	3	3	3	3	5	5	5	5
John Doe	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	3
Jane Doe	5	5	5	5	5	5	5	5	5	5	5	5	4	4	4	4	5	5
Peter Parker	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
Bart Simpson	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
Benjamin Oakwood	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5

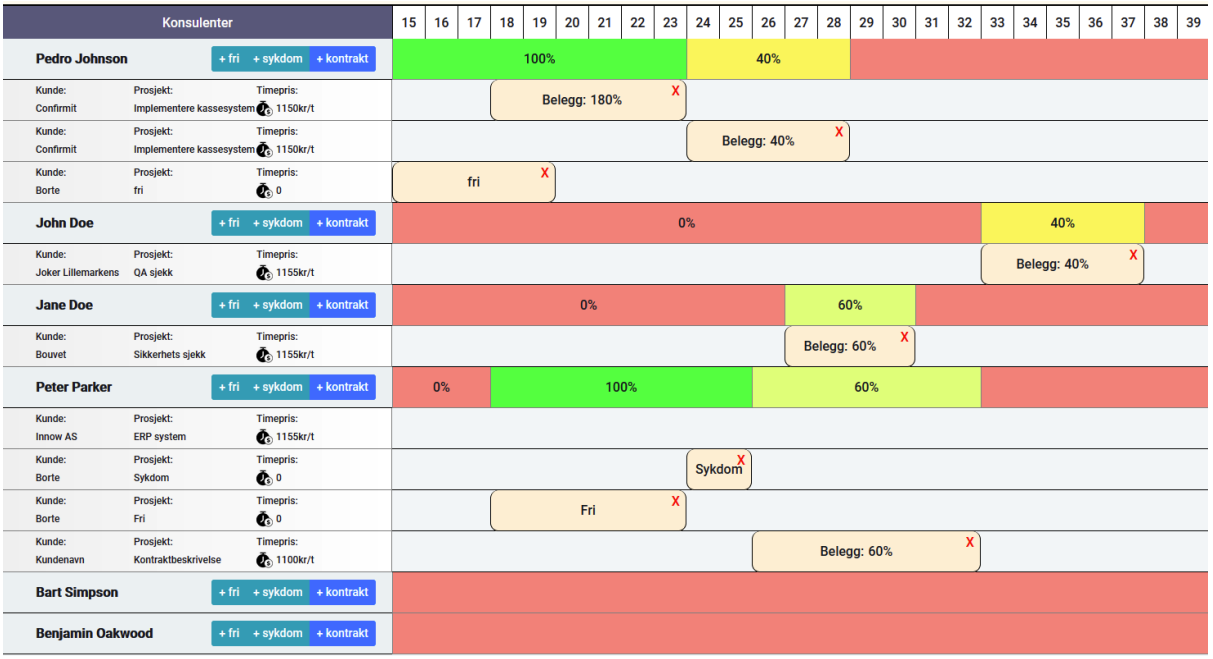
Andre iterasjon



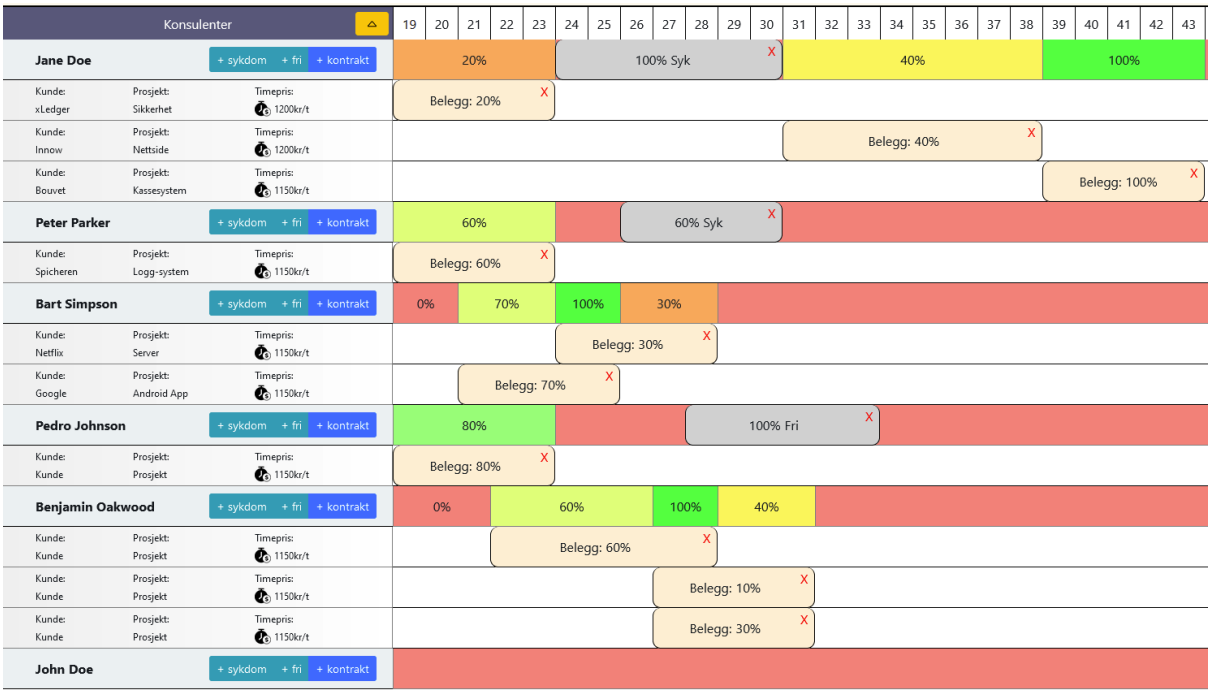
Tredje iterasjon



Fjerde iterasjon



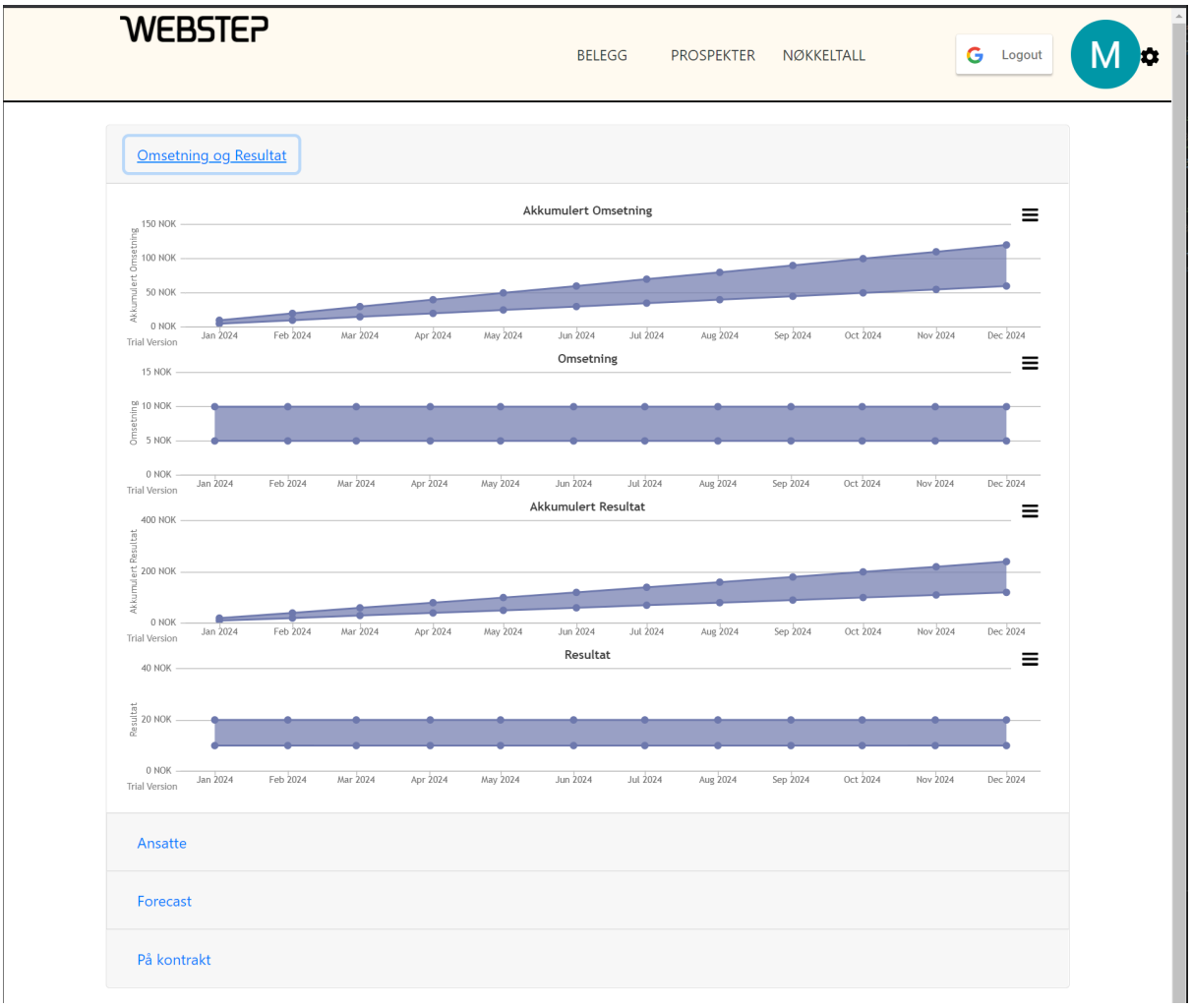
Femte iterasjon



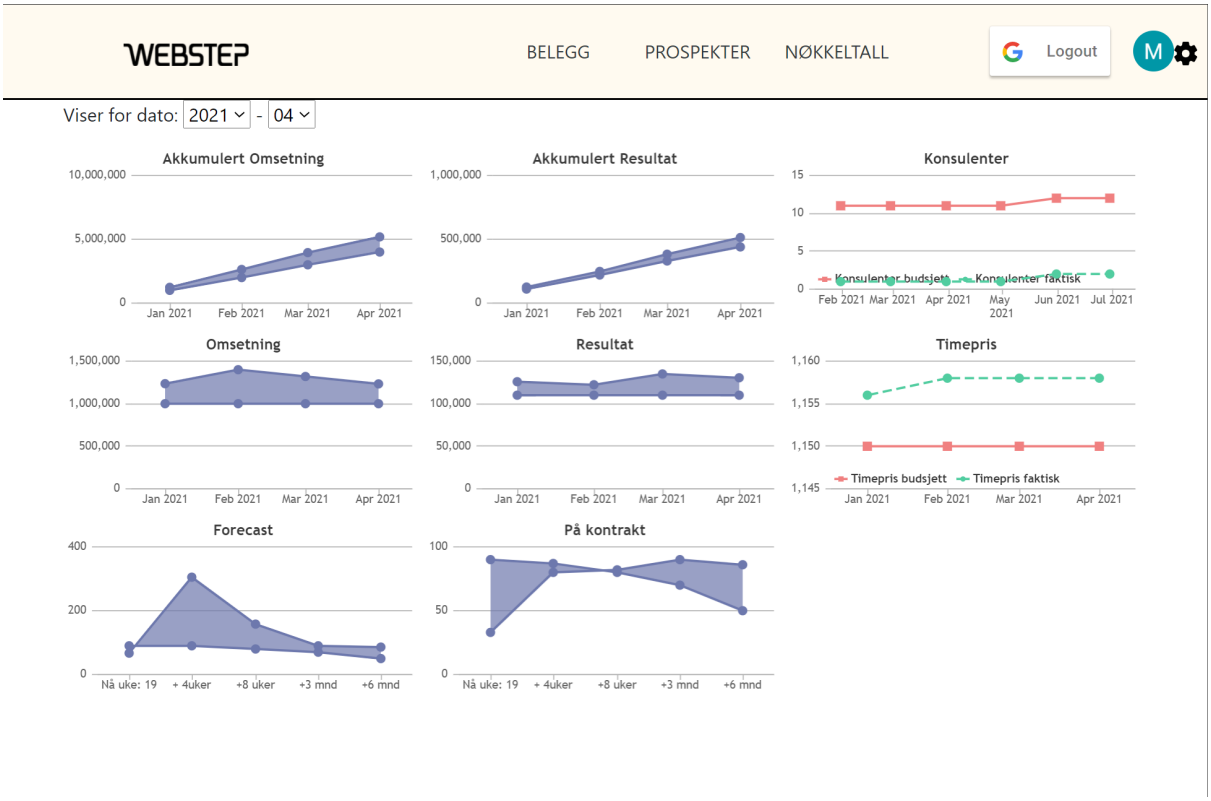
Nøkkeltall

Første iterasjon

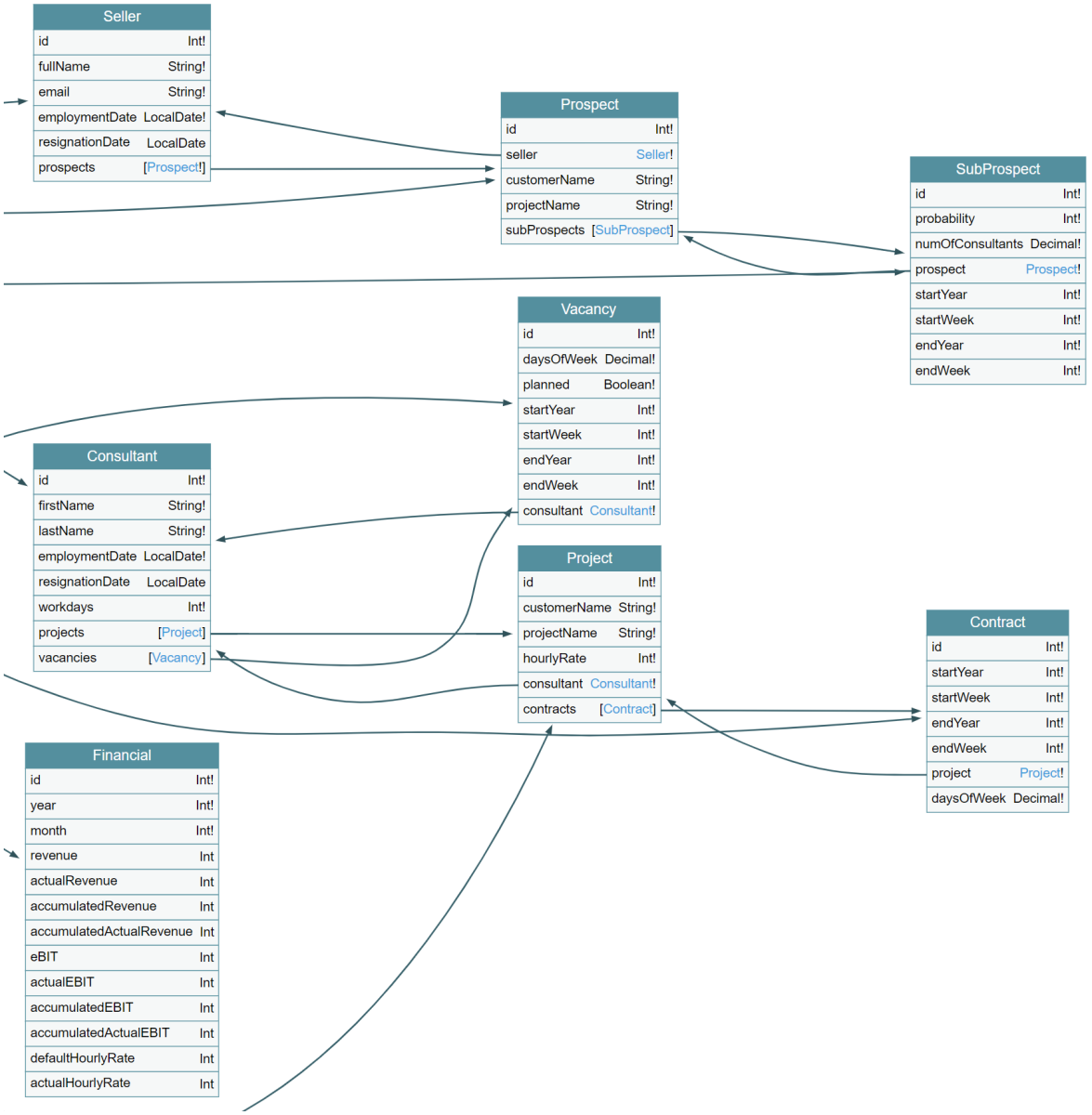
Andre iterasjon



Tredje iterasjon


WEBSTEP
BELEGG
PROSPEKTER
NØKKELTALL
Logout
M
Viser for dato: 2021 - 04
Akkumulert Omsetning
10,000,000
5,000,000
0
Jan 2021
Feb 2021
Mar 2021
Apr 2021
Akkumulert Resultat
1,000,000
500,000
0
Jan 2021
Feb 2021
Mar 2021
Apr 2021
Konsulenter
15
10
5
0
Feb 2021
Mar 2021
Apr 2021
May 2021
Jun 2021
Jul 2021
Konsulenter budsjett
Konsulenter faktisk
Omsetning
1,500,000
1,000,000
500,000
0
Jan 2021
Feb 2021
Mar 2021
Apr 2021
Resultat
150,000
100,000
50,000
0
Jan 2021
Feb 2021
Mar 2021
Apr 2021
Timepris
1,160
1,155
1,150
1,145
Jan 2021
Feb 2021
Mar 2021
Apr 2021
Timepris budsjett
Timepris faktisk
Forecast
400
200
0
Nå uke: 19
+ 4uker
+8 uker
+3 mnd
+6 mnd
På kontrakt
100
50
0
Nå uke: 19
+ 4uker
+8 uker
+3 mnd
+6 mnd

GraphQL Voyager



Bilde som viser objektene og deres felter.(Ignorer pilene)

10.2 Brukerhistorier

I de neste kapitlene vises ett eksempel av en brukerhistorie for hver side.
For å se alle brukerhistoriene (kun lesetilgang), bruk denne linken:
https://docs.google.com/document/d/1K-oeiI2cjn0_ddPLysIc2imT2qgyXVqesWMa4MwYCdw/edit?usp=sharing (kan ha blitt ugyldig hvis over 30 dager har passert)

1 - Belegg - Føre antall dager i uken
Som en selger, så vil jeg føre antall dager i uken en konsulent har på hver kontrakt,

slik at ledige ressurser blir nøyaktig.

Implementeringskrav

Data:
- Konsulenter med kontrakt
- Kontraktenes prosjektnavn
- Kontraktenes kundenavn

Funksjonalitet:
- POST/PUT API kall til backend
- Backend utfører input sanitization og verifiserer informasjon
- Backend oppretter belegg på konsulent i databasen

UI:
- Input for antall dager
- Velge konsulent
- Input for startdato (eller uke tall)
- Input for sluttdato (eller uke tall)

1 - Prospects - Se gantt-chart

Som en selger,
så vil jeg se et gantt chart som viser mine prospects,
slik at jeg kan få en oversikt over hvilke prospects jeg har.

Implementeringskrav

Data:
- Eksisterende prospects

Funksjonalitet:
- GET API kall til backend
- Backend henter og returnerer prospects.
- Frontend omdanner data til Gantt-chart elementer.
- Prospects kan filtreres etter hvilken selger som førte de

UI:
- Gantt-chart

1 - Nøkkeltall - Se omsetning (budsjett og faktisk)

Som en selger,
så vil jeg se omsetning for hver måned,
slik at jeg kan sammenligne budsjett omsetning med faktisk omsetning for avdelingens ansatte.

Implementeringskrav

Data:

- Omsetning hver måned
- Budsjett omsetning hver måned

Funksjonalitet:

- Omsetning hentes fra grunndata (eller lignende)

1 - Ledige ressurser - Se gantt chart
Som en selger, så vil jeg se gantt chart som viser % opptatthet til hver konsulent over tid, slik at jeg kan få en oversikt over kapasiteten til hver konsulent.
Implementeringskrav
Data: - Antall ledige dager i uken for hver konsulent (beregnet fra fri+sykdom+kontrakt) - Hver konsulent sin kapasitet i hver uke - Hvilken uke verdiene gjelder - Konsulentdetaljer: fornavn og etternavn Funksjonalitet: - Grad av opptatthet til en konsulent grupperes etter like og etterfølgende verdier (f.eks 50% opptatt fra uke 20 til 23) - Gruppert kapasitet omdannes til element i gantt-chart kalender - Hvis konsulenten er 100%(eller tilsvarende) opptatt, så vises de ikke under ledige ressurser UI: - Gantt-chart

1 -. Autorisering/Autentisering - Logge inn med SSO(Google)
Som en selger, så vil jeg kunne logge inn i systemet ved bruk av min Google konto, slik at jeg kan autentisere meg og få tilgang til systemet. (Kun selgere skal kunne logges inn).
Implementeringskrav
Funksjonalitet: - Google SSO integrert i vårt system UI: - Knapp for å logge inn - Knapp for å logge ut