BACHELORPROSJEKT OSLOMET

Våren 2020

Storyteller

En frontend-applikasjon utviklet for og i samarbeid med Skatteetaten

Fredrik Haraldseth
Aleksander Andersen Skjelbred

Institutt for Informasjonsteknologi

Postadresse: Postboks 4 St. Olavs plass, 0130 Oslo

Besøksadresse: Holbergs plass, Oslo

Telefon: 22 45 32 00

PROSJEKT NR.
20 - 41

TILGJENGELIGHET
Åpen

BACHELORPROSJEKT

HOVEDPROSJEKTETS TITTEL	DATO
Storyteller	25/5 2020
	ANTALL SIDER / BILAG
	71
PROSJEKTDELTAKERE	INTERN VEILEDER
Fredrik Haraldseth, s315748 Aleksander Andersen Skjelbred, s326273	Roza Abolghasemi

OPPDRAGSGIVER	KONTAKTPERSON
Skatteetaten	Torhild Rørslett

SAMMENDRAG

En frontend-applikasjon for automatisering av opprettelse og sanering av testmiljøer. Applikasjonen er utviklet etter Skatteetatens krav til design og funksjonalitet. Designet er utformet for å tilby en robust og brukervennlig løsning. Kildekoden er skrevet for å tilby god skalerbarhet som legger til rette for videreutvikling og vedlikehold.

Applikasjonen er utviklet i React med bruk av Typescript og hooks

3 STIKKORD
React hooks
Kanban
Universell utforming

Innholdsfortegnelse

Introduksjon

Forord

Dette er sluttrapporten for vårt bachelorprosjekt ved OsloMet, våren 2020. Vi har vært gjennom en svært lærerik prosess, og fått god erfaring på veien videre mot arbeidslivet. Resultatet av vårt prosjekt er en frontend-applikasjon som vil benyttes på det interne systemet hos Skatteetaten.

Vi ønsker å takke alle som har hjulpet oss med gjennomføringen av prosjektet, og i den anledning vil vi trekke frem de som har betydd aller mest for oss under prosjektperioden.

Vi vil rette en stor takk til vår veileder, Roza Abolghasemi. Takk for at du har vært løsningsorientert og veiledet oss på veien til en god rapport.

En ekstra stor takk rettes også til våre kontaktpersoner hos Skatteetaten. Takk for at dere har tilrettelagt for oss og tålmodig delt av deres erfaring med oss. Vi setter stor pris på at vi har fått denne muligheten.

Leder for testsenteret
Marianne Rynning
marianne.rynning@skatteetaten.no

Prosjektleder
Torbjørn Nesheim
torbjørn.nesheim@skatteetaten.no

Administrativt ansvarlig
Torhild Rørslett
torhild.rorslett@skatteetaten.no

Fagansvarlig
Anders Mikkelsen
anders.mikkelsen@skatteetaten.no

Innledning

Introduksjonen vil beskrive bakgrunnen for vårt prosjekt. Her vil vi presentere oss selv, vår oppdragsgiver og prosjektet vårt. Vi vil gjøre rede for hvorfor vi valgte denne oppgaven, og hvilke resultater som har kommet ut av prosjektarbeidet.

Om gruppen

Vi er to studenter med en felles toppidrettsbakgrunn. Gjennom denne bakgrunnen kjenner vi hverandre godt, og begge har en god forståelse av hva arbeidsmoral og samarbeid innebærer. Dermed ble det helt naturlig for oss å danne en gruppe for bacheloroppgaven som skal avslutte og oppsummere vår studiegang ved instituttet for informasjonsteknologi på OsloMet.

Gjennom studiets gang har vi begge fattet interesse for å utvikle programmer og applikasjoner med fokus på god brukervennlighet og optimalisert ressursbruk. Derfor ønsket vi for vårt bachelorprosjekt å utvikle en nettapplikasjon.

Om oppdragsgiver

Skatteetaten har 6500 ansatte som jobber mot samme mål, å sikre finansiering av velferdssamfunnet. Skatteetaten er underlagt Finansdepartementet, og har ansvaret for et oppdatert folkeregister og at skatter og avgifter blir fastsatt og innbetalt på riktig måte.

IT-divisjonen hos Skatteetaten består av omtrent 900 ansatte, og regnes som et av Norges største IT-miljøer. Både utvikling og drift av systemer skjer internt.

Prosjektet

Opprettelse og sanering av testmiljøer er hos Skatteetaten store og ressurskrevende operasjoner. Applikasjonen Storyteller er hovedkomponenten i et prosjekt for å automatisere disse prosessene.

Vår oppgave har vært å produsere en frontend-løsning for Storyteller. Løsningen skal tilby et enkelt og effektivt brukergrensesnitt for å interagere med backend ved hjelp av en rekke API-kall.

Utviklingsarbeidet har foregått i Skatteetatens lokaler, i tett samarbeid med et erfarent utviklingsteam. Vi har måttet forholde oss til alle Skatteetatens retningslinjer for utviklingsprosessen og hvordan det ferdige produktet skal se ut og fungere.

Da vi fikk denne oppgaven presentert var vi ikke i tvil om at dette var noe vi ønsket å gå videre med. Dette prosjektet har gitt oss verdifull og relevant arbeidserfaring, vi har lært nye og attraktive teknologier og vi har fått benyttet kunnskap fra studiet vårt.

Produkt

Vi har utviklet en fullt fungerende frontend-applikasjon, som kommuniserer med backend ved hjelp av en rekke API-kall tilbudt fra Storyteller. Applikasjonen har allerede blitt benyttet i interne demonstrasjoner for å vise hvordan Storyteller fungerer.

I løpet av vårt prosjekt har vi vært nødt til å prioritere de viktigste funksjonalitetene for å ha et fungerende produkt klart til slutten av prosjektperioden. Dette betyr også at enkelte tilleggsfunksjonaliteter gjenstår før applikasjon er klar til å tas i bruk for fullt. Disse funksjonalitetene vil redegjøres for i detalj videre i dokumentasjonen.

For oss som studenter er det viktigste resultatet allikevel ikke hvordan produktet vårt ser ut, men hva vi har lært på veien dit. I løpet av tre år på OsloMet har vi lært mye, og gjennom dette prosjektarbeidet har vi virkelig har fått benyttet våre nye ferdigheter til noe med praktisk nytte. Det har vært svært inspirerende og motiverende å skulle utvikle et produkt med reell nytteverdi.

Vi har lært mye om hvordan utviklingsprosessen foregår ute i arbeidslivet, og vi har benyttet oss av flere av de mest populære og aktuelle teknologiene.

Dokumentasjon

Denne rapporten består av prosessdokumentasjon og produktdokumentasjon. I tillegg finnes det også en oppsummerende avslutning, og vedlegg som supplerer dokumentasjonen.

Prosessdokumentasjonen beskriver arbeidet vi har vært gjennom, hvilke valg vi har tatt samt hvilke metoder og teknologier vi har benyttet. Dette dokumentet redegjør for hvorfor og hvordan vi har kommet frem til vårt endelige produkt.

Produktdokumentasjonen beskriver selve produktet vårt, hvordan brukergrensesnittet fungerer og hvordan kildekoden er bygget opp. Dette dokumentet forklarer hva vårt endelige produkt består av.

Skatteetaten stiller meget høye krav til sikkerhet, og ettersom Storyteller benytter sensitiv informasjon fra interne databaser er det dessverre ikke mulig å legge ved applikasjonen eller kildekoden i sin helhet. Selv ved en passordbeskyttet rapport er det en omstendelig prosess å få godkjent innsyn i interne applikasjoner hos Skatteetaten. For å presentere vår applikasjon på en god måte vil vi gjennom de videre dokumentene i denne rapporten gå vårt eget prosjekt i dybden. Vi vil dekke alle aspekter av prosjektet som er relevant for å forstå hva vi har utviklet og hvordan vi har arbeidet. Vi vil videre illustrere og forklare funksjonalitet i design og kode, men samtidig unngå å eksponere sensitiv informasjon.

Under arbeidet med dokumentasjonen har vi tilstrebet å bruke et enkelt og konsist språk med norske ord og uttrykk. Det forventes allikevel at leseren har en viss teknisk forståelse for å forstå begreper, utviklingsmetoder og kodearkitektur.

For å skape et godt, helhetlig inntrykk har vi har valgt å utforme denne rapporten med samme stil for design, fonter og farge som Skatteetaten selv benytter.

Prosessdokumentasjon

Forord

Prosessdokumentasjonen har til hensikt å informere leseren om hvilke prosesser gruppen har vært gjennom i prosjektarbeidet. Innholdet vil gjøre rede for hvilke valg som er tatt, og på hvilken bakgrunn disse valgene er gjort. Videre vil det være en komplett oversikt over de teknologiene som er benyttet, arbeidsmetoder som er fulgt og hvilke arbeidsforhold gruppen har arbeidet under.

Teksten inneholder tekniske faguttrykk som forklares grundigere i ordlisten i vedlegg A. Ord og begreper som er forklart i ordlisten vil være merket i kursiv.

Flere steder i teksten henvises det til arbeid med kravspesifikasjonen. Den endelige utgaven av denne finnes i vedlegg B.

Dette dokumentet er nyttig for deg som ønsker å forstå hvordan vi har jobbet. Enten for å forstå helheten i vårt produkt eller for å nyttiggjøre våre erfaringer i andre prosjekter for webutvikling med tilsvarende teknologier.

Om du heller ønsker å sette deg inn i bakgrunnen for dette prosjektet, inkludert hvem oppdragsgiver er og hvem gruppen vår består av anbefaler vi å lese introduksjonen.

For å sette deg inn i kildekoden og tekniske detaljer om dens funksjonalitet anbefaler vi å lese produktdokumentasjonen.

En oppsummering som inkluderer vårt læringsutbytte og informasjon om hva som skjer videre med applikasjonen finnes i avslutningen.

Innhold

Innledning

Prosjektarbeidet hos Skatteetaten har vært en dynamisk prosess, og vi har erfart at et prosjekt i en større bedrift har en tendens til å vokse mens det pågår. Dette dokumentet vil gå gjennom hvordan vi har jobbet mot våre mål, hvilke rammebetingelser vi har hatt og hvilke endringer som måtte gjøres underveis.

Vi har jobbet sammen med et utviklingsteam hos Skatteetaten. Arbeidet har bydd på mye læring, og videre i dette dokumentet vil vi redegjøre for hvordan det har påvirket vår prosess. Vi vil også gå gjennom hvilke valg som har blitt tatt slik at applikasjonen er tilrettelagt for videre drift og utvikling.

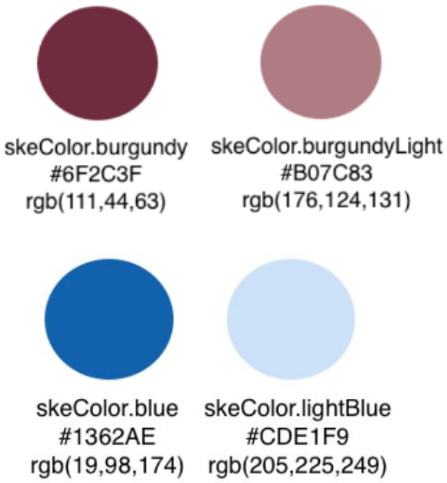
En app i Skatteetatens systemer

Skatteetaten har en rekke interne systemer og regler vi som utviklere har måttet forholde oss til, og tilpasse våre ideer etter. Dette kapitlet vil redegjøre for hvilke krav som har påvirket arbeidet vårt under prosjektperioden.

Design

Skatteetatens designsystem[1] består både av etatens visuelle profil, og et bibliotek med forhåndsdefinerte komponenter til bruk ved all *frontend* utvikling.

I designsystemet har Skatteetaten valgt Helvetica som font. Arial er satt som reserve, om Helvetica ikke er installert på maskinen. Fargepaletten inneholder forskjellige fargekoder for hovedelementer, bokser, rammer, handlinger, feilmeldinger og så videre. For full oversikt over fargepaletten henviser vi til kildelisten. Figur 1 viser hovedfargene (rødtoner) for utseendet på en side og interaksjonsfargene (blåtoner) for klikkbare elementer.



Figur 1: Utdrag fra fargepaletten

Alle applikasjoner i bruk hos Skatteetaten skal benytte seg av norsk språk. Språkbruken skal videre følge flere retningslinjer for å sikre at innholdet er konsekvent, tydelig og forståelig. Samtidig vil konvensjonene for kildekoden gi et annet språk med flere engelske uttrykk. Her har det vært helt nødvendig å holde fokus for å skille tydelig mellom språk i koden og språk i applikasjonen.

Designsystemet gjør selve prosessen med å skrive kode enklere da både utseende og funksjoner for komponenter er definert i et eget bibliotek. Samtidig kan komponentbiblioteket by på enkelte utfordringer når man skal implementere sine egne ideer ved hjelp av designsystemet. Vi opplevde at våre ideer og skisser ikke nødvendigvis var direkte kompatible med de komponentene vi hadde tilgjengelig.

[1] Skatteetatens designsystem

Mer om prosessen for å velge komponenter følger senere i dette dokumentet, og hvordan disse komponentene benyttes i vårt endelige produkt er beskrevet i produktdokumentasjonen.

Tilgjengelighet

Applikasjonen i dette prosjektet har en svært spesifikk målgruppe, da den kun vil eksistere på intranettet til Skatteetaten. Denne delen av systemet vil kun være tilgjengelig for, og benyttes av, en gruppe driftsansvarlige. De driftsansvarlige vil kun ha tilgang til applikasjonen på godkjente PCer. På bakgrunn av dette har vi vurdert at det ikke er relevant å tilpasse innholdet til mobilbruk. Det er også grunn til å anta at brukerne har høy teknisk forståelse og kompetanse.

Allikevel er det både viktig og lovpålagt å sørge for at applikasjonen følger krav til universell utforming. EUs webdirektiv (WAD)[2] pålegger intranett i offentlig sektor å følge retningslinjene i WCAG 2.1[3], nivå AA og A.

Alle komponentene i Skatteetatens designsystem er testet mot de aktuelle kravene for universell utforming. For å videre sikre korrekt utforming finnes det også retningslinjer for hvordan komponentene skal plasseres, og hvilke funksjoner hver enkelt komponent kan utføre. I tillegg har det vært en viktig del av vår prosess å også sørge for at all funksjonalitet er hensiktsmessig plassert, at det finnes tilstrekkelige og oversiktlige alternativer for navigasjon, og at den driftsansvarlige presenteres for relevante hjelpetekster og feilmeldinger. Mer utdypende om disse tiltakene kommer senere i dette dokumentet.

Skatteetatens metode

Som for designet har Skatteetaten også etablerte retningslinjer for hvilke metoder arbeidet skal følge. Her har vi blitt utfordret på å sette oss inn i flere av de mest aktuelle og ettertraktede metodene og teknologiene på markedet, samtidig som vi har fått svært verdifull trening i å jobbe som en del av et større utviklingsteam.

Under prosjektperioden har vi jobbet side om side med Aurora-teamet hos Skatteetaten. Her har vi møtt erfarne utviklere som villig har delt av sin kunnskap og erfaring med oss.

Aurora plattformen[4] er en skreddersydd Openshift[5] løsning tilpasset Skatteetatens bruk. I denne plattformen fungerer Aurora Konsoll som en hovedapplikasjon, utviklet for å kunne samle alle de mindre applikasjonene fra forskjellige team og miljøer under ett og samme system. Det er i dette hierarkiet av applikasjoner at Storyteller vil implementeres, som en del av innholdet i Aurora Konsoll. Det har derfor vært viktig å ha god og kontinuerlig kommunikasjon med Aurora-teamet for å sikre at vårt prosjekt

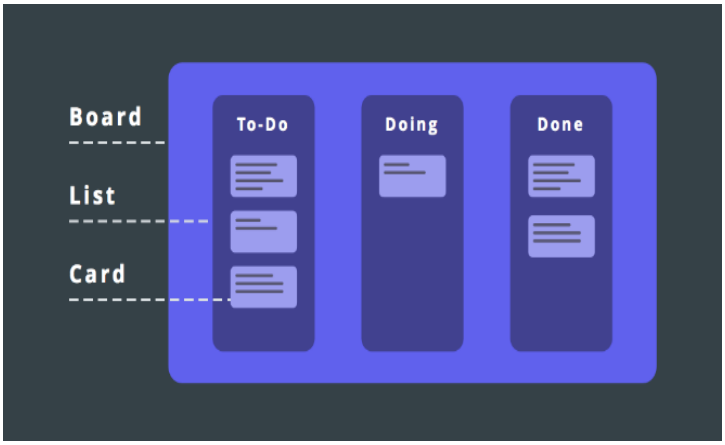
[2] Difi – EUs webdirektiv
[3] Difi – WCAG 2.1
[4] Aurora
[5] Openshift

på best mulig måte kan integreres med resten av programmene og applikasjonene under konsollet.

Metode

Hos Skatteetaten utføres arbeidet etter smidige prinsipper basert på Kanban-metodikk. Arbeidsprosessen i etaten er svært dynamisk, og det kan bli nødvendig med tidvis store endringer i kravspesifikasjonen for å tilpasse seg andre prosjekter og retningslinjer. I disse situasjonene tilbyr Kanban den mest fleksible og robuste løsningen. Sammenlignet med andre smidige metoder som Scrum har man i Kanban ingen strengt definerte sprints med tidspunkt for oppstart og avslutning. I Kanban er hele arbeidsflyten basert på oppgaver. Man utfører én oppgave av gangen, slik at hver oppgave blir som et miniprosjekt, og utvikleren kan ha fullt fokus på de spesifikke kravene for en gitt oppgave.

Arbeidet blir organisert ved hjelp av et Kanban-brett i Jira[6]. Figur 2 viser en skisse av et Kanban-brett. Her har man kontinuerlig oversikt over hvilke oppgaver som er gjennomført, under arbeid og gjenstående. En oppgave beskrives i et Kanban-kort, og oversikt over oppgavens status er definert ved hvilken Kanban-liste oppgaven er plassert i. Utviklingsteamet blir sammen enige om en rekke oppgaver og underoppgaver som skal føres opp. Deretter kan hvert medlem av teamet føre seg opp på oppgaver, og dermed vil alle med tilgang til Kanban-brettet ha oversikt over hvem som jobber med hva. Etter en oppgave er utført er standard prosedyre å legge til en, eller flere, av de andre teammedlemene på *QA*. Prosessen med *QA* benyttes for å godkjenne gjennomføringen av en oppgave, og minimerer risikoen for at ufullstendig kode når produksjonsstadiet. Etter denne sjekken er utført kan oppgaven merkes som gjennomført og flyttes til den siste listen. Dette sikrer både god kvalitetskontroll og full historikk over gjennomført arbeid.



Figur 2 – Illustrasjon av Kanban Board. Hentet fra https://zapier.com. (2020).

Mesteparten av planleggingen og kommunikasjon, både innad i vårt utviklingsteam og med Aurora-teamet, har vært gjort gjennom daglige stand-ups. Dette er et viktig prinsipp i smidig utvikling[7]. En stand-up gjennomføres stående, for å motivere til mest mulig effektiv avvikling av møtet. Innholdet i møtet består av en kjapp gjennomgang av hva hver utvikler har jobbet med siden sist, og hva som er planen videre. Her er det også rom for å ta opp eventuelle utfordringer man skulle ha støtt på, og koordinere samarbeidet innad i gruppen.

---

[6] Jira
[7] Stand-up

Styringsdokumenter

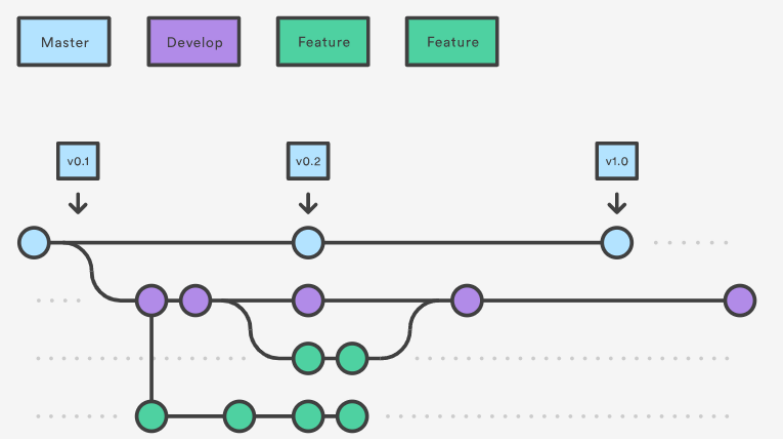
I vårt prosjekt har vi hatt en svært generell kravspesifikasjon, og stadig utvikling og forbedring av fremgangsmåter og metoder. Det har da vært kritisk å ha gode styringsdokumenter vi har kunnet forholde oss til. Til tross for endringer underveis har det overordnete målet hele tiden vært konstant og konkret.

Det ble på et tidlig tidspunkt utviklet en *MVP*-modell for å sette en fornuftig ramme rundt prosjektet vårt. Videre i utviklingen har det da vært mulig å legge til mer funksjonalitet i tillegg til denne målsetning vi satt fra start, om det skulle bli tid til det. Utgangspunktet med *MVP*-modellen er valgt for å sikre at vi kan prioritere de mest essensielle delene av systemet, og så eventuelt ha mulighet til å utvide der vi har tid og mulighet. Mer om disse valgene følger senere.

Under hele prosjektperioden har vi ført dagbok for å ha kontroll over arbeidshistorikken. Her har vi ført opp måloppnåelser, diskusjoner, problemer og løsningene på disse. Dagboken har vært verdifull i arbeidet med å lage skissen til dette dokumentet.

Underveis i arbeidet har også Kanban-brettet i Jira vært verdifullt, spesielt for planlegging av arbeid på en daglig basis. Her har det vært ført opp en *backlog* med oppgaver som må gjøres, og alle utviklerne har dermed hatt fullstendig oversikt over statusen på de forskjellige oppgavene.

For versjonshåndtering har vi jobbet etter Gitflow[8] metoden med Bitbucket[9], illustrert i figur 3. Med denne metoden blir koden delt opp i flere forgreninger og nivåer. På øverste nivå finnes produksjonsklar kode i master. Nivået under er utgangspunktet for utvikling, kalt develop. Herfra følger grenene hvor selve utvikling foregår, kalt feature. I en feature-gren kan utvikleren jobbe uforstyrret og trygt, uten risiko for å påvirke, eller i verste fall skade, produksjonskoden. Bitbucket kan også kommunisere med Jira, slik at når en oppgave i feature endrer status til «under arbeid», «*QA*» eller «utført» vil også oppgaven i Jira automatisk sorteres under riktig liste i vårt Kanban-brett.



Figur 3 - Illustrasjon av Gitflow[8]

[8] Gitflow
[9] Bitbucket

Arbeidsmiljø

Det daglige utviklingsarbeidet har foregått i Skatteetatens lokaler. Her har vi sittet i et kontorlandskap sammen med et utviklingsteam og fulgt tilnærmet normale arbeidsdager.

I tillegg til daglige stand-ups og muntlig kommunikasjon har vi hatt mulighet til å kommunisere over nett ved hjelp av Mattermost og Skype.

Sikkerhet er naturligvis prioritert svært høyt hos Skatteetaten, med store mengder sensitive data i systemene. Dette har også medført noen mindre utfordringer i tilfeller hvor det har vært nødvendig å jobbe fra utenfor Skatteetatens egne lokaler. Det stilles strenge krav til hvilket utstyr som kan brukes, hvilke kanaler for kommunikasjon som kan benyttes og sikker tilkobling til nett. På grunn av dette har vi i størst mulig grad prioritert å møte opp på kontoret, og ellers funnet fleksible løsninger de gangene det ikke har vært mulig. I en periode hvor vi begge var bortreist ble det prioritert å bruke mer tid på å oppdatere loggføring og historikk, fremfor å bruke mye ekstra tid på å finne en godkjent tilkobling til utviklingsmiljøet.

Den siste perioden av prosjektarbeidet har bydd på noen ekstra prøvelser da situasjonen rundt Covid-19 medførte at hjemmekontor ble eneste løsning for videre utvikling. I starten av denne perioden møtte vi noen frustrerende utfordringer med kapasitetsproblemer på de interne systemene for både utvikling og kommunikasjon. I første omgang kunne vi omgå dette med fleksible arbeidstider i form av kveldsjobbing og å bruke private telefoner for kommunikasjon. Heldigvis kom det, etter omtrent en ukes tid, en forbedret løsning for ekstern pålogging til utviklingsmiljøet. I perioder har det fremdeles vært problemer med interne kommunikasjonsløsninger, og vårt utviklingsteam har måttet tilpasse seg noe mer individuell jobbing enn hva som ville vært normal prosedyre. I disse situasjonene har versjonshistorikk og oppgavefordeling i Bitbucket og Jira vært ekstra verdifullt for å ha oversikt over fremdriften.

Teknologier

Å gjennomføre et utviklingsprosjekt hos Skatteetaten har medført at vi har møtt på, og brukt, mange av de mest aktuelle og populære teknologiene på markedet. En oversikt over hvilke teknologier vi har brukt følger på neste side.

- React[10] er et Javascript bibliotek, utviklet og driftet av Facebook. Det brukes for å utvikle brukergrensesnitt og baseres på flere mindre komponenter som til sammen kan utgjøre det komplette brukergrensesnittet. React legger til rette for å utvikle svært ressurseffektive applikasjoner, da en oppdatering kun vil laste inn de komponentene som er endret, fremfor å måtte laste inn hele grensesnittet i applikasjonen hver gang. I tillegg finnes det et stort utvalg av tredjepartspakker til React som kan tilby gode løsninger for de mer avanserte komponentene.
- React hooks[11] håndterer applikasjonsstatus. Hooks gir mulighet for å implementere status og fleksibel gjenbruk av logikk uten å skrive klasser for komponentene.
- Npm[12] er et pakkehåndteringsverktøy for Javascript. Npm brukes både til nedlastning og håndtering av tredjepartspakker og til å kjøre *scripts* som bedrer utviklingsopplevelsen. Disse *scriptene* kan inkludere kjøring av utviklingsapplikasjon, kjøring av tester og formatering av kode.
- Typescript[13] er en videreutvikling av Javascript, og tilfører funksjonalitet for å kontrollere korrekt brukt av *typer* og syntaks. Dette er spesielt gunstig for å sikre robusthet i store applikasjoner.
- Jest[14] er et testrammeverktøy for Javascript.
- Enzyme[15] er et testrammeverktøy for React.
- Node.js[16] brukes for å kjøre Javascript kode på klient og som API for backend.
- Swagger UI[17] er et verktøy for å visualisere og interagere med endepunkter fra API.
- Openshift[18] er en Kubernetes-basert programvare for docker containere[19]. Brukes for å distribuere applikasjoner på det interne systemet.
- Postman[20] er en API-klient, brukt for å interagere med endepunkter i API'et under utviklingen.
- Visual Studio Code[21] er en kildekodeeditor som er spesielt godt egnet for å utvikle grafiske brukergrensesnitt og har inkludert Git-kontroll.

---

[10] React
[11] React hooks
[12] npm
[13] Typescript
[14] Jest
[15] Enzyme
[16] Node.js
[17] Swagger UI
[18] Openshift
[19] Docker container
[20] Postman
[21] Visual Studio Code

Etter at Skatteetaten introduserte oss for React som det foretrukne Javascript biblioteket gjorde vi en kjapp sammenligning for å teste påstanden. Vi søkte opp antall stillingsannonser på Finn som inkluderte React, og sammenlignet antall treff mot to av de andre mest kjente Javascript bibliotekene, Angular og Vue.

Tabell 1 - Sammenligning av Javascript bibliotek

	React[22]	**Angular**[23]	**Vue**[24]
Antall annonser	81	40	34

Tabell 1 viser at React er det mest etterspurte Javascript biblioteket på Finn, per 22.05.2020.

Utviklingen

Selve koden vil omtales mer i produktdokumentasjonen, i dette kapittelet vil vi gjøre rede for noen av de viktigste valgene som ble tatt underveis i programmeringen.

Avgrense prosjektet

Storyteller er et stort og omfattende prosjekt som skal behandle mye data og mange hendelser. Det var derfor klart allerede fra starten av at vi ble nødt til å avgrense og definere rammene for vår applikasjon, slik at det lot seg gjennomføre på tiden som har blitt satt av til denne oppgaven.

Storyteller er et program for å automatisere opprettelse og sanering av testmiljøer internt hos Skatteetaten. Et testmiljø består av en konfigurasjonsfil, kalt miljøplan. Miljøplanen inneholder ett eller flere segmenter. Hvert segment vil igjen bestå av en eller flere varianter som kan ha opptil to konfigurasjonsfiler, kalt driftsplaner. Driftsplanene skal definere applikasjonene tilbudt i den gitte varianten.

Opprettelse eller endring av disse miljøene er svært ressurskrevende operasjoner å skulle gjennomføre manuelt, med mange repetitive operasjoner på store datamengder. Storyteller skal la driftsansvarlig kunne opprette, endre, gjenbruke eller slette konfigurasjoner for både miljøer og segmenter. Deretter skal driftsansvarlig enkelt kunne opprette eller sanere miljøer på bestilling, ved hjelp av disse konfigurasjonsfilene.

Vår oppgave har vært å produsere en *frontend*-applikasjon for Storyteller. Vi ble presentert med et sett *endepunkter* fra *backend* som skulle implementeres i et enkelt og effektivt grafisk brukergrensesnitt. *Endepunktene* inkluderte operasjoner på både miljøer og segmenter. I tillegg skulle vi utrede en løsning for hvordan den ferdige *frontend*-applikasjonen skal fungere innunder Aurora Konsoll.

[22] Finn. *React*
[23] Finn. *Angular*
[24] Finn. *Vue*

Det finnes selvsagt mye annen funksjonalitet denne løsningen kan tilby. På sikt skal *frontend*-applikasjonen også tilby en tjeneste hvor et testsenter kan sende inn bestillingene til driftsansvarlig.

Målene som nevnes i dette avsnittet er hva vi vurderte til et ambisiøst, men realistisk prosjekt innenfor våre tidsrammer. Der vi har droppet noe funksjonalitet har vi gjort dette med mål om å prioritere kjernefunksjonaliteten, for å sikre et mest mulig gjennomført sluttprodukt for denne prosjektperioden. Nevnte målsetninger er altså hva vi tok utgangspunkt i for å utvikle en mer spesifikk kravspesifikasjon i samarbeid med vår veileder og fagansvarlige hos Skatteetaten.

Oppstarten

Før prosjektperioden startet for alvor måtte vi bruke en del tid på å sette oss inn i nye teknologier. Vi har begge god kjennskap til Javascript fra før, men React og Typescript har en del tilleggsfunksjonalitet det var viktig å sette seg inn i. Vi jobbet også med tilvenning av interne løsninger og GitFlow, for å kunne være mest mulig effektive i det videre arbeidet.

Da vi satt i gang med prosjektarbeidet tok vi først utgangspunkt i en intern referanseapplikasjon for React hos Skatteetaten. Denne er skrevet i Javascript, og har mye til felles med Create React App[25]. Forskjellen fra Create React App til vår referanseapplikasjon er at alle *scripts og* konfigurasjonsfiler for koden i referanseapplikasjonen er tilpasset de interne løsningene hos Skatteetaten. På denne måten fikk vi et godt utgangspunkt for å kunne fokusere på utviklingen av applikasjonens kode, fremfor å måtte bruke tid på tilpasning av applikasjonen til det interne utviklingsmiljøet.

Med både mål og utgangspunkt på plass kunne vi opprette en *backlog* i Jira med våre første oppgaver.

Typescript

Vår første oppgave var å konvertere referanseapplikasjonen fra Javascript til Typescript. Det var her nyttig å ha noe kjennskap til begreper i React og Typescript fra før, men det var først gjennom denne oppgaven at vi virkelig begynte å få oversikt over hvordan både React og Typescript fungerer i praksis.

Aurora Konsoll

Den første større måloppnåelsen vi tok sikte på var å utrede hvordan vår applikasjon skulle integreres i Aurora Konsoll. Denne løsningen er avhengig av at vår app er satt opp på samme måte som resten av konsollet, slik at alt kan integreres sømløst. Et slik oppsett ville åpenbart kreve til en del tilpasninger sammen med Aurora-teamet, og dermed vurderte vi at det var gunstig å sette i gang prosessen så tidlig som mulig.

Det tok ikke lang tid før vi måtte innse at oppsettet for Konsollet var såpass omfattende at det ikke ville være hensiktsmessig for oss å bruke så mye av vår tid på

[25] Create React App

å sette oss inn i, og tilpasse, vårt produkt etter samme prinsipper. Vi ble derfor nødt til å vurdere alternative løsninger, og endte da opp med et forslag basert på *micro frontends*.

Single-Spa

Ved bruk av *micro frontends* kan hver enkelt applikasjon fungere som sitt eget, selvstendige system innunder rotapplikasjonen (her Aurora Konsoll). Dermed vil det være mer fleksibelt å gjøre spesifikke tilpasninger for hver enkelt underapplikasjon. Det blir også vesentlig enklere å utvikle nye underapplikasjoner, med mindre krav til å følge eksisterende arkitektur. Denne overgangen fordrer imidlertid at eksisterende applikasjoner også konverteres til *micro frontends*, slik at alle integrasjonene gjøres med samme metode.

Vi valgte å foreslå Javascript-rammeverktøyet Single-Spa[26]. Dette er et veldig fleksibelt verktøy, som er godt tilrettelagt for å migrere eksisterende applikasjoner i React. Det finnes flere metoder i Single-Spa for å sette opp *micro frontend* arkitekturen, men vårt forslag baserte seg på et oppsett basert på npm-pakker. Det vil si at man konverterer hver underapplikasjon til en npm-pakke, og dermed kan hver kildekode holdes adskilt i hvert sitt *repository*. Rotapplikasjonen kan da installere hver pakke via npm og benytte underapplikasjonen som en importert *modul*. Denne arkitekturen vil også gjøre det mulig å benytte *tokens* fra rotapplikasjonen til autentisering i underapplikasjonen. Slik er det mulig å sikre at kun godkjente brukere vil ha tilgang til konfigurasjonsfilene for testmiljøer.

Vi presenterte Single-Spa og oppsettet med npm-pakker på et møte med hele Aurora-teamet samlet, der vi la frem bakgrunnen for hvorfor vi kom frem til dette forslaget og hvordan teknologien fungerer. Forslaget ble tatt godt imot, men ettersom Aurora Konsoll er et stort system vil konvertering av den eksisterende arkitekturen bli et tidkrevende prosjekt i seg selv. Denne oppgaven har derfor blitt satt på vent på ubestemt tid, inntil resten av Aurora Konsollet er klart for konvertering.

Vi bestemte oss deretter for å fokusere på utvikling av vår applikasjon i et isolert miljø, slik at vi kan overlevere et mest mulig komplett produkt etter gjennomført prosjektperiode.

Skisser

Med alt lagt til rette for å starte utviklingen var første steg å tegne skisser og teste ut *lo-fi prototyper* for designet av nettsiden. Balsamiq[27] ble brukt for å tegne skissene. Vi startet med å designe brukergrensesnittet for interaksjon med segmenter, ettersom dette er det laveste nivået av konfigurasjonsfiler og mest konkret å forholde seg til. Etter en brainstormingprosess satt vi igjen med 3 aktuelle skisser for utseende og funksjonalitet i applikasjonen. Se vedlegg C.1 - C.3 for utkast.

---

[26] Single-Spa
[27] Balsamiq

Skisser i Balsamiq kan også utføre enkel interaksjon med trykkbare knapper og valg. Vi gjennomførte en enkel test med vår veileder hos Skatteetaten og et medlem av Aurora utviklingsteamet. Her fikk de prøvd funksjonaliteten i hver skisse. Etter gjennomført test diskuterte vi hvilke funksjonaliteter i skissene våre som fungerte godt. Til slutt kunne vi da kombinere de beste ideene til den endelige skissen, vist i vedlegg C.4. Denne prosessen ble gjennomført uformelt og effektivt. Ettersom vi har hatt kontinuerlig dialog med Skatteetatens utviklere gjennom hele prosessen vurderte vi det til å ikke være hensiktsmessig å følge en full vitenskapelig metode for brukertesting.

Resultater fra gjennomgang av skisser

Det er viktig å ha en overordnet valgmulighet for å bestemme hvilket segment som skal visualiseres i innholdet på nettsiden. Denne funksjonaliteten er viktig for at driftsansvarlig skal ha oversikt over hvilke varianter som hører til det spesifikke segmentet. Det ville ellers vært svært vanskelig å visualisere sammenhengen mellom varianter og segmenter om alt skulle vises på samme side.

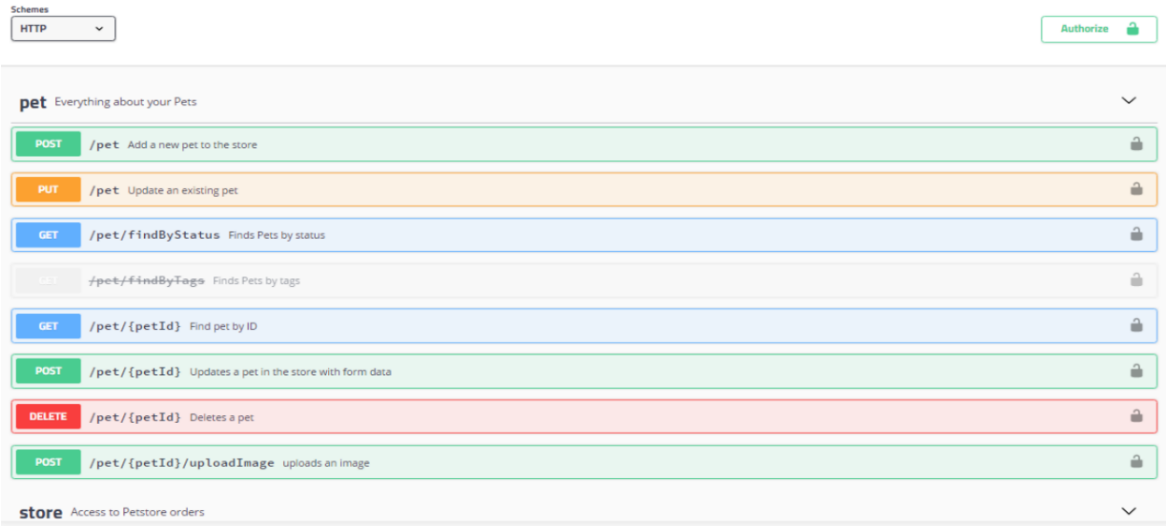
Vi tok først utgangspunkt i radioknapper, som vist i vedlegg C.1. Etter en grundigere gjennomgang av Skatteetatens designsystem måtte vi endre dette til en nedtrekksmeny, vist i vedlegg C.4. Videre er tabellvisning med komprimerte og forenklete data valgt fra vedlegg C.1, mens fargekoder for aktive varianter er hentet fra vedlegg C.3. Mer detaljert visning av innholdet for en variant blir vist i en dialogboks, hentet fra vedlegg C.2.

Vedlegg C.1 - C.3 viser også flere alternativer for filtrering og søk i selve innholdstabellen. Disse funksjonene ble kuttet ut, for å heller prioritere kjernefunksjonaliteten i applikasjonen. Vi vurderte at filtreringen vi gjør på hovedsiden for segmenter er nok til å skape en tilfredsstillende oversikt i første omgang. Muligheter for filtrering og søk er ført opp som ideer for videre forbedringer etter vårt prosjekt er overlevert.

API-kall

Ved bruk av den smidige arbeidsflyten i Kanban har vi kunnet jobbe parallelt på de grafiske komponentene og *API*-kall til *backend*. Med god planlegging av oppgavene i Kanban-brettet har det vært mulig å opprette grafiske komponenter fortløpende etter hvert som *API-kallene* har vært klare.

Vi opprettet først en *API* basert på Node.js og web rammeverket express.js[28]. *API*-kallene mot *backend* ble skrevet med Axios[29]. Dette er en enkel og oversiktlig *API* arkitektur, som ga oss muligheten til raskt å kunne returnere data i applikasjonen vår. Disse dataene ble så brukt i utviklingen av de grafiske komponentene.



Figur 4 - Swagger UI

Det ble fort tydelig at antallet *endepunkter* i *backend*, kombinert med mengden data og alle *typedefinisjoner* for disse, ville bli voldsomt og uoversiktlig å skrive som et eget *API* i Node.js. Sammen med vår veileder hos Skatteetaten fikk vi derfor konvertert *endepunktene* til et oppsett med Swagger / OpenAPI og Swagger UI. Swagger er et bibliotek for å automatisk kunne generere *API*-kall og en visuell dokumentasjon av disse. Figur 4 viser et generisk eksempel på hvordan visualiseringen av *endepunktene* kan se ut. Denne visualiseringen har vært nyttig for å ha en oversikt over hvilke kall applikasjonen må kunne utføre. De automatisk genererte API-kallene inneholder også *typedefinisjoner* i henhold til informasjon fra *backend*, og har oversatt disse til Typescript. Dermed har vi kunnet eksportere *API*-kallene direkte til funksjoner for en ryddig og robust kildekode. *Typedefinisjonene* har vi også kunnet eksportere som *interface* for å sette riktige *typer* i de andre typescript-filene.

For å kunne teste *API*-kallene på utviklingsstadiet har vi hatt tilgang på en *backend* bygget med eksempeldata som kjører i en docker container[30]. Dermed har vi kunnet gjøre forespørsler mot *backend* og fått returnert riktig formaterte data til utvikling av komponentene i applikasjonen.

Under utviklingen har også Postman vært hyppig brukt for å kontrollere hvordan de forskjellige forespørslene til *API'et* er formatert og hva de returnerer.

[28] Express.js
[29] Axios
[30] Docker container

React hooks

Enkeltkomponentene i React bygger på håndtering av *state* og *props*. Slik kan komponentene snakke sammen og utføre sine funksjoner. I den opprinnelige kravspesifikasjonen vår skulle vi bruke Redux[31] for å opprette en *store* som kan håndtere *state* i en stor applikasjon. I oppstartsfasen av utviklingen fikk vi et tips av vår veileder hos Skatteetaten om å prøve oss på React hooks. Hooks er designet for å løse mange av utfordringene som har eksistert i React, og dermed gjøre Redux langt på vei overflødig.

Grunnen til at Redux har vært nødvendig for å håndtere *state* i React er for å kunne gjenbruke logikk mellom relaterte klasser. Klasser har vært den standardiserte metoden for hvordan React organiserer koden, men er på dette området en lite fleksibel løsning. Hooks gir muligheten til å skrive funksjonelle komponenter med *state*, uten å ta i bruk klasser.

Hooks gir blant annet muligheten til å hente ut logikk og informasjon direkte fra komponentene, og gjenbruke logikken andre steder i applikasjonen. Om man ønsker å gjennomføre tilsvarende operasjon ved bruk av klasser må man benytte såkalt «prop-drilling». Denne fremgangsmåten innebærer å sende *state* som en *prop* gjennom hver eneste underkomponent i filstien til ønsket destinasjon. Vi valgte derfor å gå videre med hooks, og har konvertert all logikk i applikasjonen til å benytte funksjonelle komponenter med hooks.

Spesialtilpassete hooks

For å benytte de genererte *API*-kallene i komponentene kan disse kalles på i ordinære funksjoner. Funksjoner vil få jobben gjort, men gir ikke mulighet for å gjenbruke *API*-kallene på tvers av komponenter på en enkel måte. Her kommer hooks igjen inn i bildet. En spesialtilpasset hook kan returnere hvilke data du vil, på samme måte som en funksjon. I tillegg til dette kan hook'en få tilgang på *state*, *kontekst* og *referansepunkter* i komponenten den benyttes i. Disse verdiene kan da hook'en benytte i sin egen, interne logikk. En hook er altså alltid fleksibel og gjenbrukbar på tvers av all kode.

Den spesialtilpassete hook'en skrives som en egen komponent, og kan eksporteres ut til alle andre komponenter i applikasjonen. Ved å ta i bruk *livssyklusmetoder* og ordinære hooks kan en spesialtilpasset hook gjøre alle nødvendige sjekker for å skape en robust og gjenbrukbar kode. Hook'en kan dermed implementeres hvor som helst i applikasjonen, med minimalt av tilpasninger for avhengigheter eller nye *livssyklusmetoder*.

Pakker

I enkelte tilfeller vil selv trivielle og normale funksjonaliteter kreve svært store og komplekse komponenter. I disse tilfellene finnes det ofte tredjepartspakker som kan lastes ned med npm. Disse pakkene er utviklet av andre utviklere, og både kvalitet og

[31] Redux

funksjonalitet kan variere mye. Det er derfor viktig å gjøre godt forarbeide og velge pakker med omhu. I utgangspunktet installerer man en pakke for å gjøre videre utvikling enklere, men om man gjør en forhastet beslutning risikerer man å måtte gjøre mye dobbeltarbeid for å tilpasse en pakke med manglende funksjonalitet.

Et av kravene vi skulle løse i vår applikasjon var å gi den driftsansvarlige mulighet til å redigere konfigurasjonsfilen for enten et segment eller et miljø. Denne filen kan være formattert som enten Json eller Yaml. Det er selvsagt mulig å redigere konfigurasjonsfilen i en ordinær tekstboks, men det vil være en veldig tungvint prosess uten relevante kodeuthevninger og feilmeldinger. Vi bestemte oss derfor for å finne en pakke med funksjonalitet for å opprette en fullverdig kodeeditor i nettløsningen.

Vårt første forsøk på å laste ned en npm-pakke for kodeeditoren ble preget av dårlig bakgrunnssjekk for pakken. Det gikk med mye tid på å prøve å tilpasse funksjonalitet som lignet på våre ønsker, men ikke møtte kravene fullt ut. Takket være versjonshåndteringen i Bitbucket hadde vi mulighet til å tilbakestille applikasjonen til siste fungerende versjon, da vi forsto at vi var på vei inn i en blindgate.

Etter å ha gjort et nytt, grundigere forarbeid kom vi frem til Ace Editor[32] og Codemirror[33], begge godt vedlikeholdte pakker med bred dekning for forskjellige språk og kodeuthevninger. Det endelige valget falt her på Ace Editor da den har en utvidelse som tillater at editoren kan importeres som et eget *JSX* element og implementeres direkte i React.

Hvordan kodeeditoren er implementert beskrives detaljert i produktdokumentasjonen.

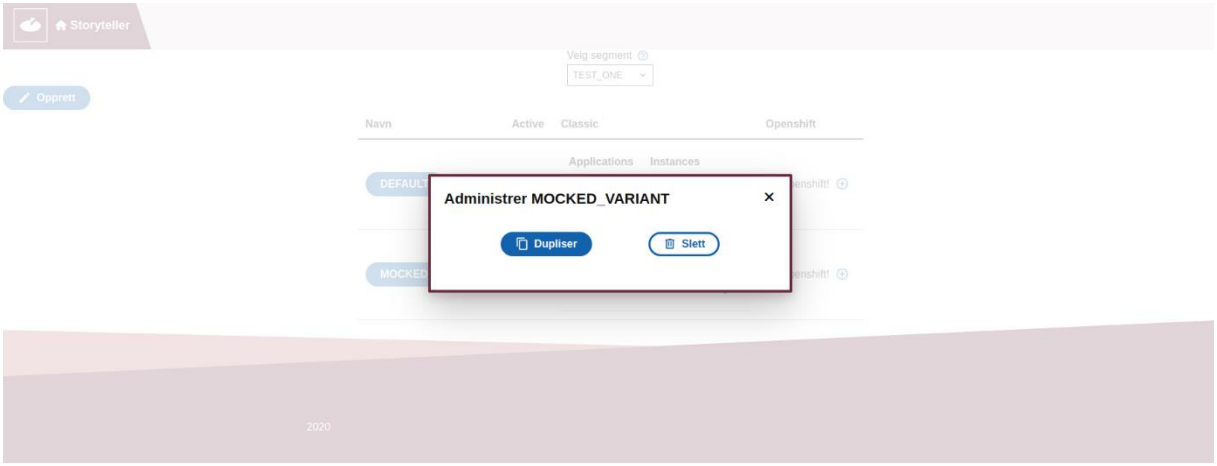
Design

Mye av designet for vår applikasjon var på forhånd bestemt gjennom retningslinjer i Skatteetatens designsystem. Det var i tilfeller hvor våre skisser ikke møtte disse føringene at vi måtte ta nye designvalg underveis i utviklingsprosessen. Ettersom retningslinjene i designsystemet er testet mot WCAG 2.1 fikk vi samtidig tilpasset vårt design til alle kravene på nivå AA. Et av de mest aktuelle punktene fra WCAG vår applikasjon må imøtekomme er 1.4.13. Her er det blant annet beskrevet hvilke krav som gjelder for følsomhet for både pekerfokus og tastaturfokus.

At vi kontrollerer vår bruk av Skatteetatens designsystem opp mot kravene i WCAG 2.1. bidrar til at vi skaper et godt og konsekvent design. På denne måten får vi utnyttet læringseffekten hos sluttbruker, og dermed øker tilgjengeligheten gjennom hele applikasjonen.

[32] Ace Editor
[33] Codemirror

Vedlegg C.4 med designskissene for segmentkonfigurasjoner viser en tabellstruktur med blanding av knapper og innholdsdata, samt fargekoding for visuell tilbakemelding på status for segmentene. Dette var en løsning vi fikk gode tilbakemeldinger på fra utviklingsteamet, men en grundig gjennomgang av designsystemet ga oss allikevel flere forbedringspunkter.



Figur 5 - Skjermdump fra applikasjon, primærknapp og sekundærknapp

Vi endte opp med å gi alle knapper et mer konsekvent utseende og kun fargebruk i henhold til interaksjonsfarger fra designsystemet. Differensiering mellom knappenes funksjon gjøres ved å skille mellom solide primærknapper og sekundærknapper med kun farget kantlinje, vist i figur 5. Det er også brukt mindre fremtredende handlingsknapper der det er hensiktsmessig med noe nedtonet synlighet.

Knapper er et veldig allsidig verktøy, og kan benyttes til nært alle interaksjoner på en side. Designet kan allikevel bli uoversiktlig om klart forskjellige funksjonaliteter har lignende knapper. Vi har derfor valgt å tydelig skille mellom linker for navigasjon og knapper som utfører databehandling.

Tabellen med informasjon om variantene ble endret til et mer kompakt og minimalistisk utseende hentet fra designsystemet.

Plasseringer og bruk av knapper for å opprette nye varianter eller endre eksisterende driftsplaner har i hovedsak blitt flyttet inn i dialogbokser. Ved å plassere interaksjonen i dialogbokser har vi kunnet minimere antall nødvendige steg en driftsansvarlig må gjennomføre for å interagere med *endepunktene* i systemet. Ved å plassere flest mulig av disse valgene inne i hver variant sin egen informasjonsdialog kan også systemet direkte sende riktig navn og id til *endepunktene*, og dermed gi en mer robust og effektiv kode.

Navigasjon i, og ut av, dialogboksene er utformet med tanke på best mulig tilgjengelighet for alle brukere med enten tastatur eller mus. Konkret hvordan dette designet fungerer er forklart i produktdokumentasjonen.

Prosessen med å forbedre design og tilgjengelighet er gjennomført som en white-box test av utviklerne. Det vil si at vi drar nytte av vår kjennskap til applikasjonens funksjonalitet for å oppsøke potensielle problemer og mangler mens vi går gjennom et typisk bruksmønster i henhold til kravspesifikasjonen. Denne prosedyren gir gode muligheter for å forbedre strukturen i applikasjonen på et tidlig tidspunkt i utviklingen. Dermed unngår vi å bruke mye ressurser på repetitive prosesser for å endre en mer kompleks kildekode på et senere tidspunkt.

White-box testingen viste også at funksjonalitetene for interaksjon som applikasjonen skal utføre kan deles i to distinkt forskjellige bruksmønstre. De mindre komplekse funksjonalitetene vil befinne seg på hovedsiden og er tilgjengelige ved hjelp av dialogbokser. Innholdet i dialogboksene består av ytterligere informasjon med hjelpetekster og knapper for å gjennomføre en ønsket operasjon. De mer komplekse funksjonalitetene består av å opprette eller redigere eksisterende driftsplaner. Disse operasjonene krever en helt annen oppbygning med ytterligere tilgang på både data og tilbakemeldinger. Dette er løst ved å la driftsansvarlig navigere til en ny nettside som laster inn driftsplanen i Ace Editor før videre operasjoner kan påbegynnes.

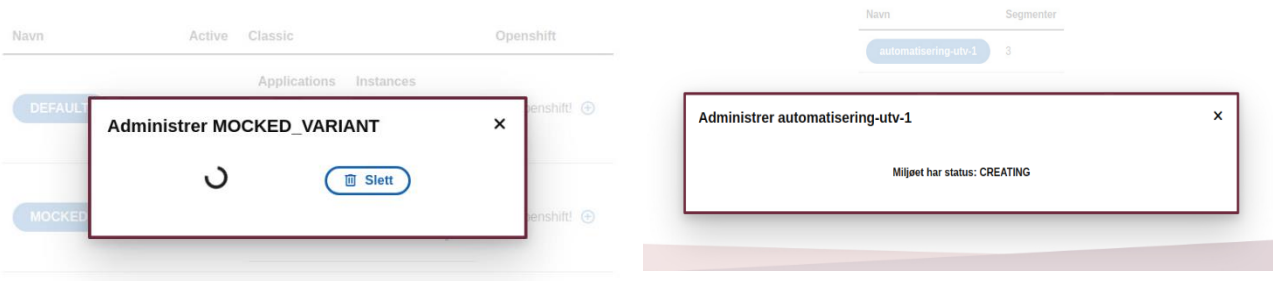
Figur 6 - Skjermdump av kodeeditor med feilmelding

På denne måten slipper vi å bruke unødvendige ressurser på å lese ny data i tabellen mens driftsansvarlig oppdaterer driftsplanene. Vi får også muligheten til å sende navn og id som parameter gjennom nettadressen for den nye siden som vil lastes inn. Valget om å utføre disse operasjonene på en ny nettside sørger også for en mer oversiktlig navigasjon med de tradisjonelle mulighetene for å gå tilbake i nettleseren, og å oppdatere denne ved behov. Den nye nettsiden kan da også ha funksjonalitet for å dynamisk gi tilbakemelding basert på eventuelle feilkoder i Json / Yaml filen som blir redigert. Eksempel på en feilmelding er vist i figur 6.

Vi har ikke prioritert å utføre ytterligere tester av designet, da alle Skatteetatens designkomponenter fra før er godkjent til gjeldene lovverk for universell utforming. Ettersom det er en svært spesifikk brukergruppe for denne applikasjonen har vår førsteprioritet for designet vært å følge de konvensjonene som fra før av eksisterer på interne systemer, og som de driftsansvarlige da vil være godt kjent med. Gjennom tett samarbeid med teamet som vil drifte applikasjonen videre har vi kunnet kvalitetssikre funksjonaliteten i designet for den aktuelle brukergruppen.

Da vi senere skulle utvikle designet for interaksjon med miljøer og miljøplanene tok vi i stor grad utgangspunkt i hva vi hadde gjort for segmenter. I disse tilfellene vil gjenbruk sikre et konsekvent design som brukeren kjenner seg igjen i ved hjelp av læringseffekten.

Det er også utviklet en felles forside med navigasjonsvalg for å laste inn data for enten miljøer eller segmenter.



Figur 7 - Skjermdump av applikasjon med visuell tilbakemelding og applikasjon med tekstlig tilbakemelding

Det er i alle komponenter laget dynamiske sjekker basert på komponentens status for å kunne gi tekstlig eller visuell tilbakemelding på systemets status. Figur 7 viser to forskjellige typer tilbakemeldinger i dialogboksene. Til venstre vises en visuell tilbakemelding for å illustrere at en operasjon arbeider / laster. Til høyre vises tekstlig tilbakemelding som i dette tilfellet har erstattet deaktiverte knapper med ugyldig funksjonalitet. Begge disse funksjonalitetene er opprettet ved hjelp av sideeffekter basert på *livssyklusmetoder*.

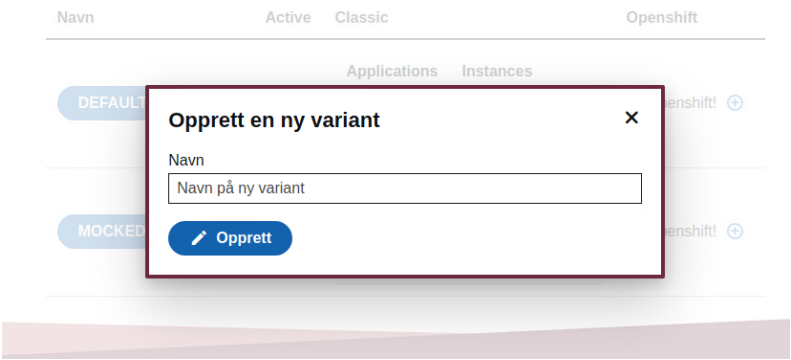
Disse tiltakene medfører at den driftsansvarlige har oversikt over hva applikasjonen gjør og hvordan den responderer. Samtidig bidrar tiltakene til å sikre en meget ressurseffektiv applikasjon som kun laster inn de nødvendige funksjonalitetene for hver oppdatering.

Etter hvert som valgene for designet ble ferdigstilt tok vi også hensyn til å gjenbruke like designelementer der det gir mening. På denne måten sikrer vi både et konsekvent design som er enkelt å benytte for driftsansvarlig, samt en enklere utviklingsprosess og en mer oversiktlig kode for vedlikehold og videreutvikling.

Designelementer

Dialogboks

Dialogboksen plasseres i forgrunnen av annet innhold, og har et konsekvent design som vist i figur 8. Innholdet i boksen består av en overskrift, informasjon og interaksjon. Mer om hvordan denne benyttes finnes i produktdokumentasjonen.



Figur 8 - Dialogboks

Knapper

Knappene er valgt utformet med myke hjørner som vist i figur 9. Slik skapes det en viss grafisk kontrast mot et ellers stramt design bestående av tabeller, objekter og verdier for disse. Det er gjort et skille mellom primærknapper og sekundærknapper som beskrevet tidligere.

Figur 9 - Knapper

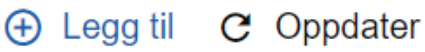
Linker

De ordinære knappene benyttes for å gjennomføre en funksjonalitet, mens klikkbare linker benyttes for å navigere mellom de forskjellige nettsidene som utgjør applikasjonen. Figur 10 viser to forskjellige typer linker.

Figur 10 - Linker

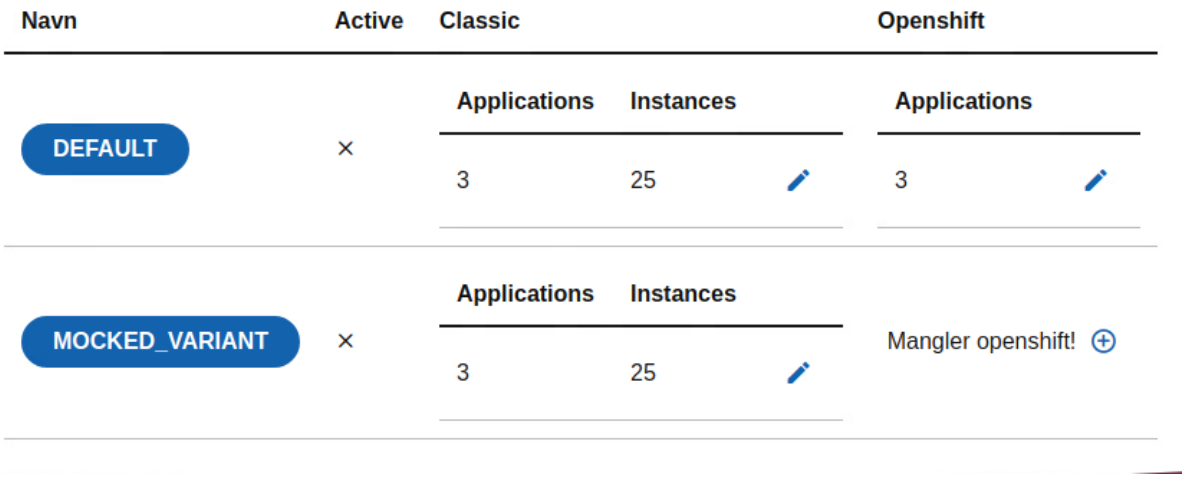
Handlingsknapper

For å ytterligere kunne differensiere mellom forskjellige funksjoner har vi tatt i bruk et eget design for handlingsknapper, vist i figur 11. Disse fungerer på mange måter likt som vanlige knapper, men er mindre dominerende på siden og benyttes for mindre viktige operasjoner, hjelpehandlinger eller i deler av applikasjonen hvor det er lite plass til designelementer. Et eksempel på det siste er i informasjonsfeltene i tabellene.



Figur 11 - Handlingsknapper

Tabell



Figur 12 - Tabell

Tabellen benyttes for å strukturere informasjon, og plassere visualiseringer og tilhørende funksjonaliteter innenfor et avgrenset område. Tabellen kan som i figur 12 inneholde både objekter, knapper, tekst og mindre tabeller.

Spinner

Figur 13 viser et dynamisk designelement som brukes for å gi tilbakemelding om at applikasjonen arbeider mens data fra *API*-kallene prosesseres.

Figur 13 - Spinner

Kodestruktur

Applikasjonen vil driftes og utvikles videre av Aurora-teamet etter at vi har overlevert vårt produkt. Det har derfor vært høyt prioritert å skrive oversiktlig og ryddig kode.

Hver komponent eller fil i React har kun en bestemt og overordnet oppgave, med navn som beskriver hvor i systemet de hører hjemme, og hvilken funksjon de har. Et slikt oppsett gjør at det er enkelt å navigere seg gjennom systemet under en eventuell feilsøking eller for videreutvikling.

Koden i hver enkelt komponent er organisert på en ryddig måte med funksjoner og hook'er sortert etter kronologisk rekkefølge. Slik har vi oppnådd en konsekvent og gjennomgående struktur gjennom hele applikasjonen. Under utviklingsprosessen er det også brukt en *linter* fra ESLint[34] som sørger for at all kode automatisk formateres på samme måte, uansett hvem som har skrevet koden.

Stilarkene for hver komponent er implementert som Styled Components[35] til slutt i koden. Dermed er også stilen skrevet inn i samme fil som resten av komponenten, og det sikrer god tilgjengelighet og lesbarhet når andre utviklere skal sette seg inn i koden. Dette vil illustreres i produktdokumentasjonen.

Filstruktur

Komponentstrukturen vi har valgt medfører også at det blir mange filer å holde styr på, og filstrukturen er kritisk for å sikre god vedlikeholdbarhet. Her har vi fulgt konvensjoner fra tidligere prosjekter hos Skatteetaten, slik at strukturen blir mest mulig gjenkjennelige for de som utvikler og drifter applikasjonen videre. Fra rotmappen til prosjektet skal det være logisk å følge en *filsti* frem til ønsket komponent. All kode vi har produsert er sortert etter om det er komponenter, visningsskjermer eller tjenester, og videre er det gjort et skille mellom komponenter for miljøer og segmenter. Det er også opprettet en egen mappe som inneholder tilleggstjenester som kan tilby gjenbrukbare elementer for å forenkle de større komponentene lenger ned i koden.

Alle spesialtilpassede hooks er navngitt i henhold til konvensjoner for hooks, der navnet skal starte med «use». Slik er det mulig å skille en hook fra en ordinær funksjon.

---

[34] ESLint
[35] Styled Components

Alle testfiler er navngitt på formatet komponentNavn.test.tsx. Jest er satt opp for å automatisk lete etter filer merket med .test, og samtidig sikrer dette at alle utviklere enkelt kan skille testfilene fra de faktiske komponentene.

Fil- og kodestrukturen er beskrevet i større detalj i produktdokumentasjonen.

Testing

Parallelt med utviklingsarbeidet har vi også skrevet enhetstester for komponentene i applikasjonen. For brukergrensesnittet er testene skrevet med Jest og Enzyme. Testene av tjenester med spesialtilpassete hook'er er skrevet med Jest og React-hooks-testing-library[36].

Jest tilbyr en enkel måte å *mocke*, eller etterligne, funksjonene i en komponent, mens Enzyme og React-hooks-testing-library er gode rammeverk for å teste at man faktisk får forventet data returnert fra komponenten som testes.

Gjennom all testing har det vært fokus på å kun teste den indre logikken, og unngå å teste rammeverk eller interaksjon mellom komponenter.

Videre utvikling

Kravspesifikasjonen for dette prosjektet har vært tilpasset tidsrammene vi har hatt tilgjengelig. Når dette prosjektet avsluttes har vi et fullt fungerende produkt, men det vil alltid finnes muligheter for videre forbedringer og utvikling.

Datasystemer har ofte en tendens til å vokse med tiden, og vår applikasjon med tabellvisning risikerer da å bli uoversiktlig om det blir for mange datafelter. I det innledende arbeidet bestemte vi oss for å kutte ut funksjonalitet for filtrering og søk, men dette kan bli aktuelt å implementere på et senere tidspunkt av utviklingen. Ved å ta i bruk filtrering på hovedsiden vil det være mulig å opprettholde god oversikt, selv med større datamengder enn hva vi har tatt høyde for.

Det har også vært et ønske fra Skatteetaten å kunne visualisere hvilke segmenter som er tatt i bruk i et miljø. *Endepunktet* for denne informasjonen er ikke ferdig utviklet, men det er tilrettelagt for å implementere dette så fort det finnes støtte for det i *API'et*.

Testing er så langt kun gjennomført på enhetsnivå, her gjenstår det en komplett ende-til-ende test med integrasjonstesting og systemtesting før det ferdige produktet er klart til å iverksettes på Openshift.

Til slutt gjenstår det å finne ut hvilken metode som skal benyttes for å integrere Storyteller i Aurora Konsoll. Da må det utredes om Storyteller skal tilpasses eksisterende arkitektur, eller om det blir valgt å konvertere til en *micro frontend*-løsning.

[36] React-hooks-testing-library

Produktdokumentasjon

Forord

Produktdokumentasjonen redegjør for hvordan applikasjonen fungerer og hvordan kildekoden er bygget opp. Innholdet informerer om hvilke funksjonaliteter brukergrensesnittet tilbyr, og det vil illustreres hvordan koden er utviklet for å sikre en mest mulig ressurseffektiv applikasjon.

Teksten inneholder tekniske faguttrykk som forklares grundigere i ordlisten i vedlegg A. Ord og begreper som er forklart i ordlisten vil være merket i kursiv.

Alle teknologier som nevnes i dette dokumentet er forklart og har henvisninger til kildelisten i prosessdokumentasjonen.

Dette dokumentet er nyttig for deg som ønsker å sette deg inn i kildekoden, funksjonalitet i brukergrensesnittet og tekniske detaljer for denne applikasjonen.

Om du heller ønsker å sette deg inn i bakgrunnen for dette prosjektet, inkludert hvem oppdragsgiver er og hvem gruppen består av anbefaler vi å lese introduksjonen.

For å sette deg inn i utviklingsprosessen og utredelser om hvilke valg vi har gjort anbefaler vi å lese prosessdokumentasjonen.

En oppsummering som inkluderer vårt læringsutbytte og informasjon om hva som skjer videre med applikasjonen finnes i avslutningen.

Innhold

Innledning

Frontend-applikasjonen er utviklet for å tilby et enkelt og effektivt brukergrensesnitt for å interagere med konfigurasjonene for Skatteetatens testmiljøer i Storyteller. Designet er utformet for å gi en oversiktlig visuell representasjon av miljøer og tilhørende segmenter.

Storyteller er et program for å automatisere opprettelse og sanering av testmiljøer internt hos Skatteetaten. Et testmiljø består av en konfigurasjonsfil, kalt miljøplan. Miljøplanen inneholder ett eller flere segmenter. Hvert segment vil igjen bestå av en eller flere varianter som kan ha opptil to konfigurasjonsfiler, kalt driftsplaner. Driftsplanene skal definere applikasjonene tilbudt i den gitte varianten.

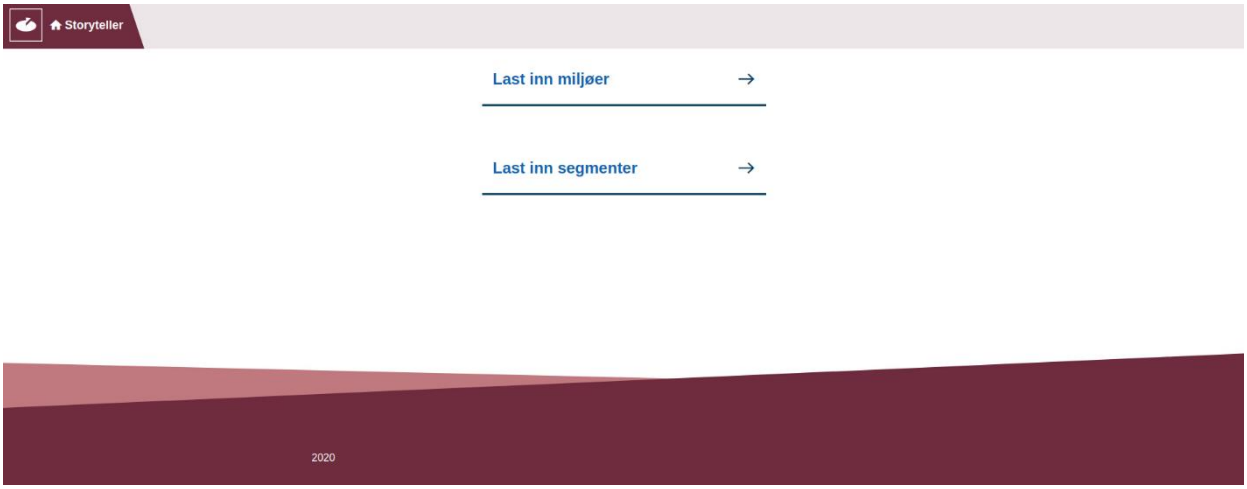
På bakgrunn av retningslinjene for sikkerhet vi følger vil vi ikke gå videre inn på hvordan testmiljøene og tilhørende konfigurasjonsfiler hos Skatteetaten er bygget opp, hvordan disse brukes eller hvordan man som sluttbruker kan bestille opprettelse eller sanering av testmiljøet. Dette er heller ikke viktig for å forstå vårt prosjekt og vårt produkt.

Applikasjonen vi har utviklet skal benyttes av en driftsansvarlig for å administrere miljøplaner og driftsplaner. Vi vil derfor forklare i detalj hvordan applikasjonen løser den driftsansvarliges oppgaver.

I prosessdokumentasjonen er design og utvikling forklart i den rekkefølgen vi brukte under utvikling, altså med utgangspunkt i segmentsiden. I denne produktdokumentasjonen vil vi følge den naturlige arbeidsflyten i applikasjonen, og derfor gå gjennom miljøsiden før vi kommer til segmentsiden.

Brukergrensesnitt

Startside



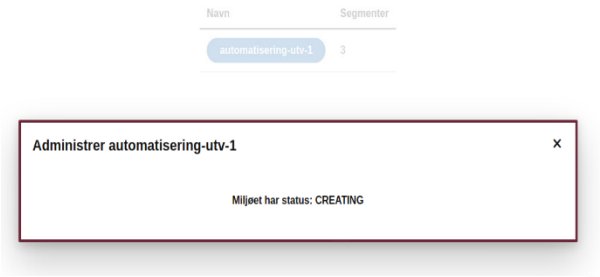
Figur 1 - Skjermdump av startside

Det er et gjennomgående fokus i applikasjonen på effektiv utnyttelse av nettverksressurser. Startsiden, vist i figur 1, lar den driftsansvarlige velge om data for miljøer eller segmenter skal lastes inn. Disse linkene laster inn hver sin nye nettside som så utfører en GET-forespørsel før videre operasjoner blir tilgjengelige.

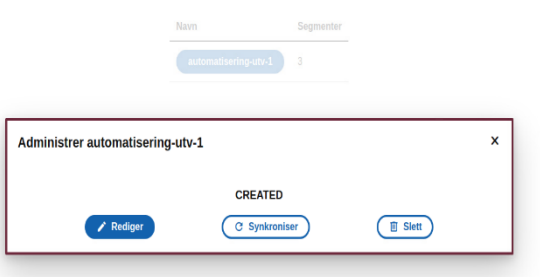
Topp- og bunnstripen for nettsiden går igjen gjennom hele applikasjonen. I toppstripen er det implementert navigasjon tilbake til startsiden i logo og tittelfeltet.

Miljøer

Hovedsiden for miljøer har to primærfunksjoner. Det skal visualiseres informasjon om eksisterende miljøplaner i en tabell, og det skal være en knapp for å opprette en ny miljøplan. Tabellen skal tilby interaksjon gjennom et klikkbart navnefelt, som åpner en dialogboks for administrasjon av miljøplanen.



Figur 2 - Dialog med tilbakemelding

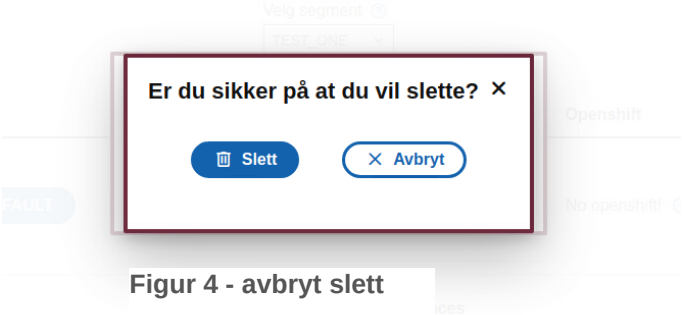


Figur 3 - Dialog med funksjonalitet

I dialogen for administrasjon av miljøplaner gir applikasjonen en tilbakemelding basert på status for valgt miljøplan. Om miljøet allerede er i gang med en prosess er ikke nye operasjoner tilgjengelig før denne er utført. Da vil knapper for operasjoner skjules og erstattes med aktuell status for miljøet, som vist i figur 2. Figur 3 viser dialogboksen for administrasjon av en ferdig opprettet miljøplan med tilgjengelig funksjonalitet for å redigere, synkronisere eller slette denne. I figur 3 vises også den aktuelle statusen for miljøet over knappene.

Dialogboksene er utformet for å ivareta prinsippene for tilgjengelighet og utforming som er utdypet i prosessdokumentasjonen. Når en dialogboks åpnes, vil automatisk bakgrunnen gråes ut og gjøres midlertidig utilgjengelig. Tastaturfokus settes til innholdet i dialogen, og sikrer at innholdet er tilgjengelig både for tastatur- og musebrukere. For å lukke dialogboksen er krysset i øverste hjørne tilgjengelig både ved hjelp av tastatur og mus. Dialogen lukkes også når det registreres et klikk på utsiden av boksen, eller når escape-tasten trykkes. Når ønsket operasjon i dialogen er utført vil boksen automatisk lukkes, og hovedsiden oppdateres med de siste endringene som er gjennomført.

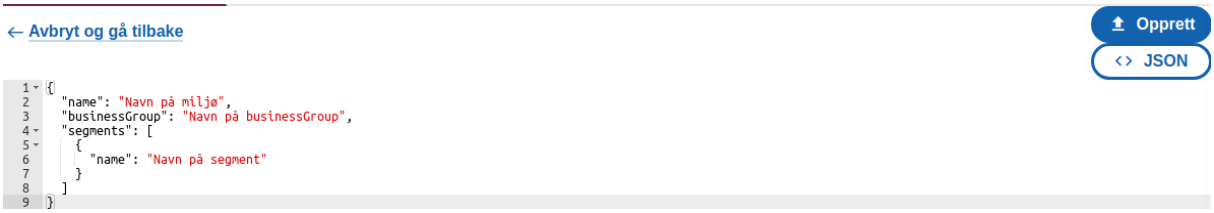
Ved ikke-reverserbare handlinger som “Slett” vil en dialog, vist i figur 4, be om ekstra bekreftelse fra brukeren før handlingen utføres.



Figur 4 - avbryt slett

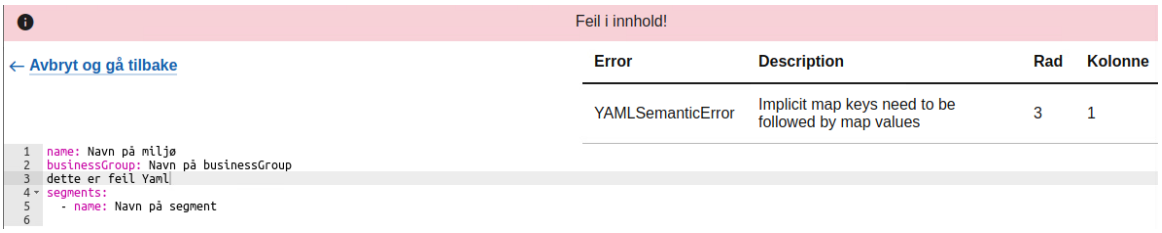
Opprett nytt miljø

Knappen for å opprette en ny miljøplan navigerer til en ny side hvor applikasjonens kodeeditor benyttes for å fylle ut den nye konfigurasjonsfilen. Det er her opprettet en mal som skal hjelpe den driftsansvarlige med å overholde riktig format på miljøplanen.



Figur 5 - Skjermdump av kodeeditor

Inne i kodeeditoren finnes det funksjonalitet for å opprette den nye miljøplanen med en POST-forespørsel mot databasen for miljøer. Det finnes også funksjonalitet for å konvertere teksten fra Json til Yaml, og motsatt, samt en link tilbake til hovedsiden. Se figur 5 for en oversikt.

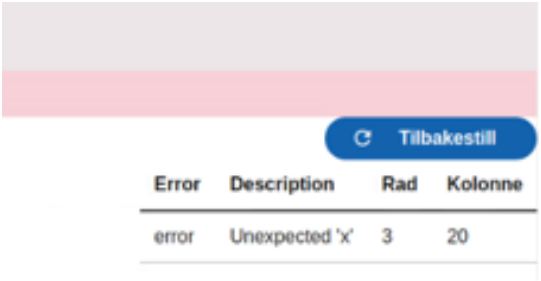


Figur 6 - Skjermdump av feilmelding

Det er også implementert syntakssjekk og feilmeldinger i kodeeditoren. Figur 6 viser hvordan en feil i Yaml håndteres. Funksjonalitet for å opprette miljøplanen eller konvertere teksten skjules og erstattes med en informativ feilmelding.

Endre miljøplan

For å endre en miljøplan blir man presentert med samme type kodeeditor som blir brukt ved opprettelse av nye miljøplaner i figur 4. Her vil eksisterende miljøplan lastes inn, slik at den driftsansvarlige kan redigere innholdet. Når den nye miljøplanen opprettes utføres det en PUT-forespørsel mot databasen for miljøer. Om feilmelding vises er det i dette tilfellet også funksjonalitet for å tilbakestille miljøplanen til de opprinnelige data, vist i figur 7. Funksjonaliteten for å tilbakestille miljøplanen er utført ved å gjennomføre GET-forespørselen for innholdet på nytt.



Figur 7 - tilbakestill editor

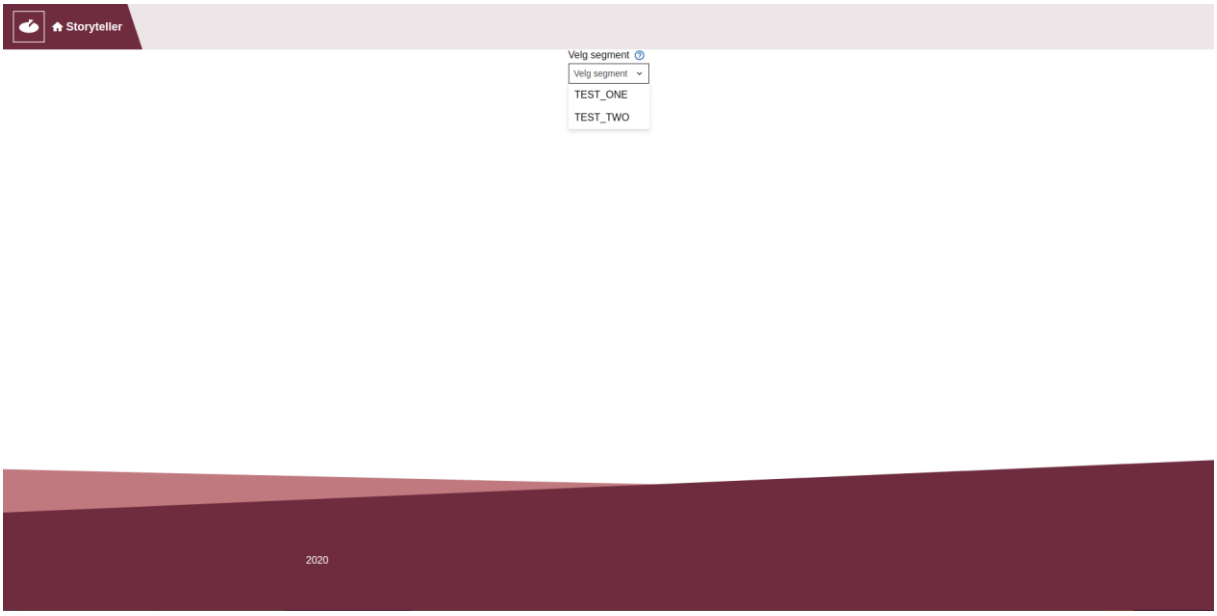
For både opprettelse og endring av miljøplan vil man etter gjennomført og vellykket opprettelse av denne bli sendt tilbake til hovedsiden som laster inn den nye informasjonen i tabellen.

Segmenter

Tabellen på hovedsiden viser en verdi for hvor mange segmenter som er tilknyttet et gitt miljø. Her er det lagt opp til å kunne implementere funksjonalitet for å knytte segmenter til et miljø, men dette *endepunktet* i *API'*et er ikke ferdig utviklet.

Segmenter

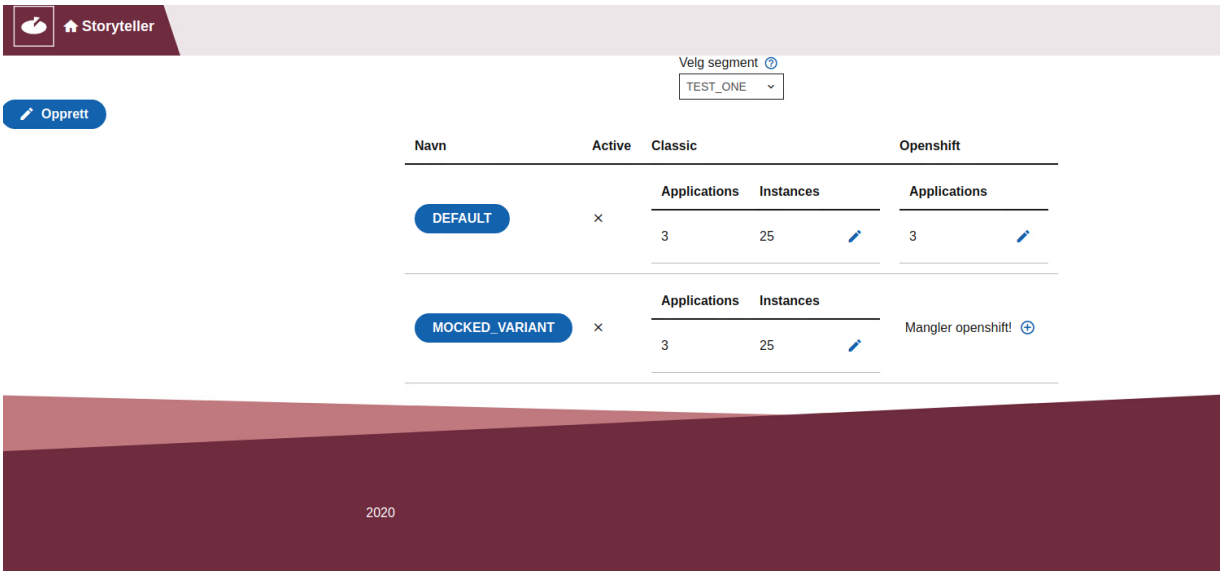
Hovedside



Figur 8 - Skjermdump av hovedside for segmenter

I brukergrensesnittet for segmenter er det ønskelig å kun visualisere variantene som utgjør et bestemt segment. Derfor vil applikasjonen i første omgang bare laste inn en liste over tilgjengelige segmenter, som vist i figur 8, før videre innhold vil lastes inn.

Når et segment er valgt vil hovedsiden gjennomføre en GET-forespørsel som oppdaterer nettsiden med en interaktiv tabell og informasjon om hver eksisterende variant.



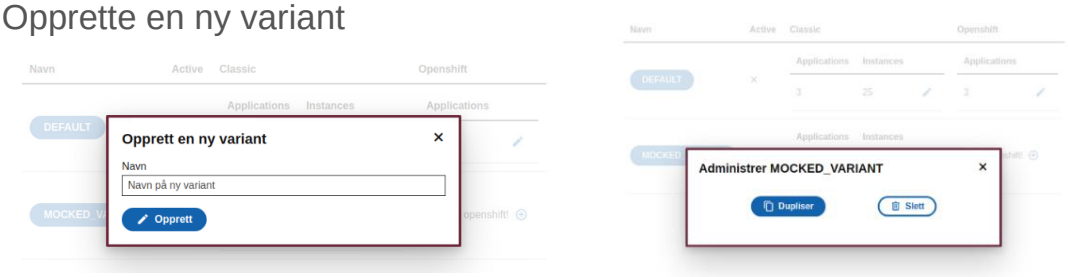
Figur 9 - Skjermdump av hovedside med tabellvisning

Ettersom designet for nettsiden til miljøer er basert på samme mal som nettsiden for segmenter har de to nettsidene flere likhetstrekk. Figur 9 viser segmentnettsiden med funksjonalitet for å opprette en ny variant, og tabellen som visualiserer eksisterende varianter.

Funksjonalitet for varianttabellen skiller seg ut fra hva som er vist for miljøer. En variant kan ha driftsplaner beregnet for enten «Classic» server, «Openshift» server, begge deler eller ingen av delene.

Avhengig av hvorvidt driftsplanen eksisterer vil tabellen vise gjeldende informasjon om denne, eller en tekstlig tilbakemelding og funksjonalitet for å opprette ny driftsplan.

Opprette en ny variant



Figur 10 - Skjermdump av dialoger for varianter

Funksjonaliteten for å opprette en ny variant skiller seg også fra hvordan applikasjonen fungerer på miljøsiden. I motsetning til et miljø kan en variant eksistere uten konfigurasjonsfil. Derfor består operasjonen for å opprette nye varianter kun av en dialogboks, vist til venstre i figur 10, som tar imot navn. Opprett-knappen utfører en POST-forespørsel for å opprette den nye varianten.

Når en variant er opprettet vil navnefeltet i tabellen være et klikkbart element som åpner en ny dialogboks, se til høyre i figur 10 for eksempel på dialogen. Denne dialogboksen tilbyr funksjonalitet for å duplisere varianten med en ny POST-forespørsel, eller slette hele varianten med en DELETE-forespørsel. Å duplisere varianten kan være nyttig i de tilfellene man ønsker å gjenbruke en allerede definert driftsplan.

Redigere en driftsplan

I de tilfellene en variant har en driftsplan er denne visualisert med informasjon om innholdet i tabellen, se figur 9. Her finnes det også en klikkbar handlingsknapp for å redigere driftsplanen. Ved klikk på denne vil en dialogboks tilby valg for å redigere eller slette driftsplanen. Valget for å redigere laster inn en ny nettside med tilsvarende kodeeditor som kjent fra miljøsiden. I denne kodeeditoren vil knappen for å oppdatere driftsplanen utføre en PUT-forespørsel mot databasen for segmenter. Knappen merket «Slett» utfører DELETE-forespørselen for å slette den spesifikke driftsplanen.

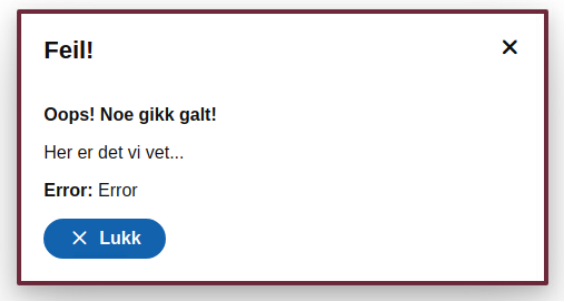
Opprette en ny driftsplan

I de tilfellene en driftsplan ikke eksisterer vil applikasjonen returnere en tekstlig tilbakemelding og funksjonalitet for å opprette en ny driftsplan. Handlingsknappen for å opprette ny driftsplan laster inn en ny nettside med kodeeditoren og tilhørende POST-forespørsel for å opprette ny driftsplan.

På samme måte som for miljøsiden vil en gjennomført og vellykket opprettelse av en driftsplan benytte en *rutingfunksjon* for å navigere tilbake til hovedsiden, og laste inn en oppdatert tabell.

Feilmeldinger

Om det skulle oppstå en uventet feil i applikasjonen vil dette utløse en funksjon for å opprette en feilmelding til driftsansvarlig. Feilmeldingen vist i figur 11 vil for eksempel vises om *frontend* mister kontakt med serveren i *backend.*



Figur 11 - Skjermdump av feilmelding

Kodegjennomgang

Filstruktur

All kildekoden for applikasjonen er samlet i src-mappen. På øverste nivå i src ligger index-filen som fungerer som inngangspunktet i applikasjonen og definerer basisoppsettet for design i alle *DOM*-elementer som skal vises i nettleseren. Videre kode er sortert i mapper etter hvilken funksjonalitet som utføres.

Data fra Bitbucket viser at vi til sammen har produsert omtrent 20.000 linjer med kode. Det vil derfor ikke være mulig å presentere alt vi har produsert i dette dokumentet, men avsnittene som følger vil gi en overordnet oversikt og presentere noen av løsningene vi har kommet frem til i vår applikasjonen.

Screens

Logikken for hvilke komponenter som vises i nettleseren finnes i screens-mappen. Den viktigste adapteren for *DOM*-elementene håndterer *ruting* i applikasjonen, og hver gang en ny nettside i applikasjonen skal lastes inn går det via denne filen. *Rutingen* er implementert med *switch* fra *react-router-dom,* som laster inn ønsket komponent i henhold til hvilken nettadresse som blir lest inn som *path.* Denne filen tar seg også av tilkoblingen mot *API'*et og oppretter en global kontekst for dette som kan benyttes senere i hooks. Denne filen inneholder sensitiv tilkoblingsinformasjon som ikke kan eksponeres i dokumentasjonen.

Ruting benyttes blant annet for å skille mellom kodeeditoren for miljøplaner og kodeeditoren for driftsplaner. Avhengig av hvilken nettadresse som sendes inn i *path* vil riktig komponent med tilhørende *API*-kall mot riktig del av *backend* lastes inn.

Screens-mappen inneholder også startsiden med statisk innhold, som vist i figur 1.

Components

Komponentene som inneholder *DOM*-elementene har sin egen mappe kalt components. Denne består av en mappe for filer tilknyttet miljøer, og en mappe for filer tilknyttet segmenter. I tillegg har vi en mappe med mer generelle elementer som skal være tilgjengelig for alle komponentene i applikasjonen. Sistnevnte mappe inneholder blant annet animasjonen som illustrerer at en komponent laster inn data og dialogboksen for å vise feilmeldinger.

Miljøer

Mappen for komponentene tilknyttet miljøer og miljøplaner inneholder alle *DOM*-elementene som vises i brukergrensesnittet for miljøer. Filen med navn EnvironmentTable.tsx inneholder komponenten for hovedsiden med tabellen som viser eksisterende miljøer. Knappen for å opprette nye miljøplaner er et element i denne filen. Det klikkbare navnefeltet er en komponent importert fra NameButton.tsx. NameButton.tsx inneholder data og funksjoner for dialogboksen som åpnes når navnefeltet mottar et klikk.

```
const [environmentButtons, setEnvironmentButtons] = useState(
  <EnvironmentContainer>
    <h3 style={{ flexBasis: '100%' }}>{input.environment.state}</h3>
    <RouteEditEnvironment id={input.environment.id ?? ''} />
    <SynchronizeEnvironment
      id={input.environment.id ?? ''}
      actionCompleted={actionCompleted}
    />
    <DeleteEnvironment
      id={input.environment.id ?? ''}
      actionCompleted={actionCompleted}
    />
  </EnvironmentContainer>
);

useEffect(() => {
  if (
    input.environment.state !== 'APPROVED' ||
    'CREATED' ||
    'UPDATED' ||
    'DELETED'
  ) {
    setEnvironmentButtons(
      <EnvironmentContainer>
        <h4>Miljøet har status: {input.environment.state}</h4>
      </EnvironmentContainer>
    );
  }
}, [input.environment, setEnvironmentButtons]);
```

Figur 12 - Skjermdump av NameButton.tsx

Figur 12 viser et kort utdrag av koden i NameButton.tsx. Koden i figuren illustrerer et godt eksempel på hvilken rolle hooks spiller i vår applikasjon. Alle funksjoner som

starter med «use» er hooks. UseState brukes for å deklarere en utgangsverdi for innholdet, mens useEffect utfører en sideeffekt basert på *livssyklus*. UseEffect tar inn en avhengighetsliste som sin andre parameter, vist på siste linje i figur 12. Innholdet i useEffect blir utført når dataene i avhengighetslisten oppdateres. Det vil si at om innholdet i variabelen «input.environment» eller funksjonen «setEnvironmentButtons» blir opprettet, endret eller fjernet, vil if-sjekken i vår useEffect utføres. If-sjekken inneholder en set-funksjon som brukes for å endre eller oppdatere innholdet i *state*-variabelen. Resultatet av dette vises i figur 2 og 3. Knappene i figur 3 kommer fra de importerte komponentene vist i deklarasjonen av useState i figur 12. Statusmeldingen i figur 2 er resultatet etter at useEffect har fått utføre sin set-funksjon.

RouteEditEnvironment.tsx implementeres på linje 4 i figur 12 og er komponenten som viser knappen merket «Rediger» i figur 3. Denne navigerer til en ny nettside for å redigere miljøplanen, ved hjelp av *rutingfunksjonen.*

```
export const EditEnvironmentLoader = () => {
  const { id } = useParams();

  return <UpdateEnvironment id={id!} />;
};
```

Figur 13 - Skjermdump av innhold i rutingkomponenten

Når vi *ruter* til en ny side kan vi bruke forhåndsbestemte felter i nettadressen til enkelt å hente parametere ned som variabler i koden vår. Figur 13 viser hvordan useParams løser dette svært elegant. Konstanten «id» benyttes da i «UpdateEnvironment» for å laste inn korrekt miljøplan til kodeditoren.

UpdateEnvironment.tsx inneholder kodeeditoren som brukes for å redigere miljøplaner, og alle tilhørende funksjoner. Dette er en av de største og mest komplekse komponentene i vår applikasjon.

```
const UpdateEnvironment = (input: Input) => {
  const configuration = useEnvironmentConfiguration(input.id);
  const aceEditor = useRef<AceEditor>(null);
  const [value, setValue] = useState<string | undefined>();
  const [body, setBody] = useState<EnvironmentConfiguration>({
    name: '',
    segments: [],
  });
  const [errors, setErrors] = useState<IAnnotation[]>([]);
  const [mode, setMode] = useState<Mode>(Mode.JSON);
  const [complete, setComplete] = useState(false);
  const result = useUpdateEnvironment({ id: input.id, body: body });
  const history = useHistory();
```

Figur 14 - Skjermdump av UpdateEnvironment.tsx

Figur 14 viser hvordan useState benyttes for å deklarere alle variabler som benyttes i komponenten. Disse variablene er definert med *typer* (markert i grønt) slik at videre kode i Typescript kjenner formatet og *typen* for variabelen. Merk at noen av disse *typene* er hentet fra Swagger sine automatisk genererte *interface*, og gir derfor ikke nødvendigvis umiddelbar mening. Hvordan disse *interface*'ene er opprettet er beskrevet i prosessdokumentasjonen. Variablene definert ved hjelp av et *interface* inneholder som regel et objekt. Konsekvent bruk av *typer* sikrer robust kode, og er spesielt viktig i mer komplekse komponenter.

Figur 14 viser også implementasjon av spesialtilpassete hooks på andre og nest siste linje. Begge disse tar i bruk «id» fra input, som ble forklart i figur 13.

```
const returnHome = useCallback(() => {
  history.push('/operations/environments');
}, [history]);

useEffect(() => {
  if (!complete && body.name !== '') {
    if (!result.loading) {
      setComplete(true);
    }
  } else if (complete) {
    setBody({ name: '', segments: [] });
    setComplete(false);
    returnHome();
  }
}, [complete, body.name, result.loading, returnHome]);
```

Figur 15 - Skjermdump av UpdateEnvironment.tsx. #2

Komplekse komponenter vil ofte stille spesielt høye krav til presis bruk av sideeffekter og *livssyklusmetoder* for å håndtere alle operasjonene som skal utføres. Den boolske variabelen «complete» fra figur 14 benyttes i denne komponenten for å bestemme om ønskede *API*-kall er utført, og dermed kan komponenten gjennomføre en prosess for å nullstille alle *state* og avslutte operasjonen. Figur 15 viser en useEffect som utfører denne prosessen.

Siste kodelinje før avhengighetslisten til useEffect'en i figur 15 inneholder *callback*-hook'en «returnHome». Når denne kalles sender den en ny verdi for nettadressen til *path* i *routingkomponenten*, og applikasjonen returner til hovedsiden og oppdaterer tabellen med den nye miljøplanen.

```
const validateYAML = () => {
  try {
    yaml.parse(aceEditor.current?.editor.getValue() || '', {
      // @ts-ignore
      prettyErrors: true,
    });

    setErrors([]);
  } catch (error) {
    const yamlError = error as YAMLError;

    setErrors([
      {
        type: yamlError.name,
        text: yamlError.message.substring(
          0,
          yamlError.message.indexOf(' at line')
        ),
        // @ts-ignore
        row: yamlError.linePos.start.line - 1,
        // @ts-ignore
        column: yamlError.linePos.start.col - 1,
      },
    ]);
  }
};
```

Figur 16 - Skjermdump av UpdateEnvironment.tsx. #3

Ace Editor som benyttes i denne kodeeditoren har innebygget funksjonalitet for å produsere feilmeldinger for Json. Innebygde feilmeldinger manglet for Yaml, og vi var derfor nødt til å skrive en egen funksjon for å få dette på plass. Figur 16 viser hvordan blant annet referansepunktet «aceEditor» fra figur 14 er brukt på den tredje linjen for å hente ut innholdet i kodeeditoren. Det brukes en «try...catch»-blokk i figur 16. Hvis catch oppdager en feil, vil den utføre set-funksjonen «setErrors». Denne funksjonen definerer en feilmelding på samme format som Ace Editor allerede bruker for Json. Dermed kan feilmeldingen senere kalles manuelt i samme *callback*-funksjon som skriver ut feilmeldinger for Json fra før.

```
return (
  <EditorContainer>
    {buttons()}
    <TextContainer>
      <AceEditor
        ref={aceEditor}
        onLoad={(editor) => editor.focus()}
        onChange={(value) => {
          if (mode === Mode.YAML) {
            validateYAML();
          }

          setValue(value);
        }}
        onValidate={(annotations) => setErrors(annotations)}
        mode={mode.toLowerCase()}
        theme="xcode"
        name="PlatformConfiguration"
        fontSize={14}
        showPrintMargin={false}
        showGutter={true}
        highlightActiveLine={true}
        value={value}
        maxLines={Infinity}
        width="100%"
        setOptions={{
          showLineNumbers: true,
          tabSize: 2,
        }}
      />
    </TextContainer>
  </EditorContainer>
);
```

Figur 17 - Skjermdump av UpdateEnvironment.tsx. #4

Til slutt kommer koden, vist i figur 17, hvor selve kodeeditoren for komponenten blir returnert. Det er her alle funksjonene og variablene vi nå har definert kan settes sammen for å fullføre den komplette funksjonaliteten. Her er referansepunktet koblet til kodeeditoren, og vi har to *callback*-funksjoner (onChange og onValidate) som oppretter eventuelle feilmeldinger når det er behov for det. For å sikre universell utforming settes markørens fokus til editoren med en gang den er lastet inn. Feltet med navn «mode» kaller en funksjon som dynamisk vil endre editorens språkmodus fra Json til Yaml, og tilbake igjen, ved bruk av knappen vist i figur 5.

På linje 3 i figur 17 vil funksjonen «buttons» laste inn knappene for blant annet å oppdatere miljøplanen. Oppdateringsknappen inneholder en funksjon som igjen vil kalle på sideeffekten i figur 15 for å endre «complete» til «true», og dermed *ruter* applikasjonen automatisk tilbake til hovedsiden.

Segmenter

Mappen for komponentene knyttet til segmenter er strukturert på samme måte som for miljøer. Hovedsiden består av en komponent som viser tabellen fra figur 9. Alle dialogbokser for interaksjon med driftsplanene er importert som egne komponenter.

```
const actionCompleted = () => {
  setPerformPostAction(true);
};

useEffect(() => {
  if (performPostAction) {
    setPerformPostAction(false);
    input.setReloadSegment(true);
  }
}, [input, performPostAction]);
```

Figur 18 - Skjermdump av kode for oppdatering av innhold

Brukergrensesnittet for segmenter inkluderer flere enklere operasjoner enn hva tilfellet er for miljøer. Disse operasjonene kan utføres i dialogbokser som de vist i figur 10. Det er praktisk å slippe ruting til nye sider for disse operasjonene, men dette gjør også at innholdet på nettsiden må oppdateres på en annen måte enn ved innlasting av nye nettsider. Figur 18 er et lite utdrag av kode som går igjen i alle komponentene som finnes i varianttabellen. Her utfører useEffect en funksjon som er arvet fra forelderkomponenten gjennom en *prop* kalt «input». Forelder i dette tilfellet er varianttabellen. Denne sideeffekten gir mulighet til å oppdatere tabellen etter en operasjon i dialogboksen er gjennomført. Dette er en ressurseffektiv løsning, ettersom det ikke er nødvendig å laste inn hele nettsiden på nytt i disse tilfellene. Disse funksjonene for oppdatering trenger heller ikke å sendes lenger ned i filstrukturen, og derfor er «prop-drilling» en akseptabel fremgangsmåte i dette tilfellet.

Ved hjelp av denne funksjonen som oppdaterer komponentens *state* er det også mulig å bestemme at en dialogboks automatisk skal bli lukket etter dens operasjon er gjennomført, basert på verdien i *state*. Denne fremgangsmåten for automatisert navigasjon sikrer maksimal tilgjengelighet etter prinsippene beskrevet i prosessdokumentasjonen.

```
return (
  <ClassicConfigurationContainer>
    {input.classic ? (
      <Table id="ClassicConfig" data={data} columns={columns} />
    ) : (
      <NoClassicContainer>
        <p>Mangler classic!</p>
        <RouteCreateConfiguration
          id={input.id}
          variant={input.variant}
          platform={'CLASSIC'}
        />
      </NoClassicContainer>
    )}
  </ClassicConfigurationContainer>
);
```

Figur 19 - Skjermdump av innhold i tabell

På grunn av oppdateringsfunksjonen beskrevet i forrige avsnitt blir det også vesentlig enklere å håndtere *livssyklus* i disse komponentene. Komponenten blir uansett oppdatert ved hjelp av funksjonen i *props*, og når en komponent blir oppdatert vil også React automatisk laste inn denne komponenten på nytt i nettleseren. Dermed slipper vi å benytte useEffect for å dynamisk endre innholdet i tabellen. I figur 19 benyttes en *ternary operator* som sjekker om en variant har gyldig driftsplan. Avhengig av sjekken vil enten innholdet for variantens driftsplan, eller en komponent for å opprette ny driftsplan returneres.

Funksjonalitet for å opprette eller redigere driftsplaner er utviklet med samme arkitektur som beskrevet for miljøplanene i forrige kapittel.

Styled components

Alle komponentene bruker stilark fra styled components.

```
const MenuContainer = styled.span`
  display: flex;
  flex-wrap: wrap;
  overflow-x: hidden;
  text-align: center;
`;

const ButtonContainer = styled.span`
  display: flex;
  width: 50%;
  flex-wrap: wrap;
  justify-content: flex-end;
`;
```

Figur 20 - Skjermdump av styled components

Figur 20 viser typisk implementasjon av styled components for en av komponentene. Det er benyttet *flexbox* for å plassere elementene.

Services

Funksjonaliteten for å interagere med *endepunktene* til *API'et* ligger i service-mappen. Her ligger alle de spesialtilpassete hook'ene som blir benyttet av komponentene i mappen «components».

Alle de spesialtilpassete hook'ene er bygget opp etter samme format. Først kommer en rekke forskjellige useEffect's som til sammen utgjør en robust løsning for å sikre at alle nødvendige avhengigheter er oppfylt før *API*-kallet kan gjennomføres. Etter at alle kriterier og avhengigheter er oppfylt vil selve endepunktet, generert ved hjelp av Swagger/OpenAPI, bli kalt i en *promise*-funksjon.

Når en hook skal benyttes i en komponent må alltid hook'en implementeres på det øverste nivået i komponenten. Det vil si at en hook ikke kan plasseres inne i en annen funksjon, *callback* eller sjekk. Disse reglene fører til at arkitekturen som illustreres videre i teksten, med avhengighetssjekker og sideeffekter inne i selve hook'en, er nødvendig.

```
const isMounted = useRef<boolean | null>(null);

useEffect(() => {
  isMounted.current = true;

  return () => {
    isMounted.current = false;
  };
}, []);

useEffect(() => {
  if (error) {
    throw error;
  }
}, [error]);

useEffect(() => {
  if (reload) {
    setComplete(false);
  }
}, [reload]);
```

Figur 21 - Skjermdump av spesialtilpasset hook

Figur 21 viser tre forskjellige useEffect's inne i en spesialtilpasset hook. Ved mer tradisjonell bruk av *livssykluser* og sideffekter ville alle disse funksjonene vært skrevet sammen i en stor kodeblokk, og det ville vært behov for flere kodelinjer med sjekker bare for å bestemme hvilke tilfeller som skal føre til hvilket resultat. Hooks gir mulighet for å benytte avhengighetslisten til å skrive svært konkrete funksjoner for hvert enkelt tilfelle av *livssyklusene*, som igjen gir en mer robust og effektiv applikasjon.

Den første useEffect'en tar i bruk et referansepunkt fra useRef. Legg merke til at avhengightslisten her er tom. Denne fremgangsmåten returnerer en boolsk «true» når selve komponenten blir lastet inn, og når komponenten lukkes vil den boolske verdien endres til «false». Denne boolske verdien kan brukes videre i koden som et

slags sikkerhetsnett for å hindre at det vil utføres en uendelig loop av mislykkede operasjoner i bakgrunnen så fort server og klient er koblet sammen.

De spesialtilpassete hook'ene skal ha funksjonalitet for å returnere feilmeldinger ved behov, og funksjonalitet for å gjennomføre *API*-kallet på nytt om siden skal oppdateres. Om disse to funksjonalitetene var skrevet i samme kodeblokk ville begge funksjonene påvirket en felles *state*. Det ville igjen ført til unødvendig ressursbruk med overflødige oppdateringer. Derfor er de to funksjonalitetene adskilt i hver sin useEffect, som vist i figur 21, med hver sin avhengighetsliste som bestemmer når funksjonen skal utføres.

```
api.segments
  .indexSegments()
  .then((result) => {
    switch (result?.status) {
      case 200:
        if (isMounted.current) {
          setOutput(result.data);
        }
        break;
      case 400:
        throw Error('Bad request!');
      case 401:
        throw Error('Unauthenticated!');
      default:
        throw Error('Unknown error!');
    }
  })
  .catch((error) => {
    if (isMounted.current) {
      setError(error);
    }
  })
  .finally(() => {
    if (isMounted.current) {
      setComplete(true);
    }
  });
```

Figur 22 - Skjermdump av spesialtilpasset hook. #2

Etter alle avhengighetene er oppfylt vil den spesialtilpassete hook'en iverksette *API*-kallet.

Api.segments på første linje i figur 22 er en konstant opprettet med useContext, som henter konteksten sin fra *rutingkomponenten* nevnt i starten av dette kapitlet. Denne arkitekturen muliggjør en global *API*-konstant, fremfor å måtte opprette nye forbindelser med *backend* for hvert eneste *API*-kall.

Filen som brukes i dette eksempelet returnerer en liste eller index over eksisterende segmenter, og «indexSegments» på andre linje er funksjonen som utfører dette *API*-kallet. Resultatet fra denne hook'en vises i nedtrekkslisten i figur 8.

Promise-funksjonen vil avhengig av resultatet enten benytte «setOutput» til å returnere dataene, eller returnere en feilmelding, for så å endre status til gjennomført

operasjon. Til slutt, når operasjonen har *state* satt til «complete», kan komponenten som implementerer denne spesialtilpassete hook'en motta dataene som ble returnert. Legg merke til at her blir «isMounted» fra figur 21 benyttet for å forhindre den nevnte uendelige loopen.

Tester

Det er skrevet enhetstester for både brukergrensesnittet og *API*-kallene. Disse er plassert i samme mappe som sine tilhørende filer med logikk, men er navngitt tydelig som tester. Ved å plassere testene i samme mappe som kildekoden vil testkjøring bli mer effektiv med færre oppslag i kodestrukturen.

Test av brukergrensesnitt

Det er brukt Enzyme med Jest for å utføre testene av brukergrensesnittet. Input og eventuelle funksjoner er *mocket* med Jest, før komponenten kan testes i Enzyme og det gjennomføres en funksjon som tester faktisk resultat mot forventet resultat. For komponentene som dynamisk endrer innhold basert på data er det skrevet separate testtilfeller for hvert mulige utfall. For komponenter med interaksjon er det også skrevet funksjoner for å simulere tastetrykk og annen relevant interaksjon som er tilgjengelig.

```
const completeMock = jest.fn();

const input = {
  id: 'MOCK',
  actionCompleted: completeMock,
};

describe('DeleteEnvironment', () => {
  it('Should allow user to open popup and delete an environment', () => {
    const component = mount(<DeleteEnvironment {...input} />);

    expect(component.find('Dialog').prop('hidden')).toEqual(true);
    component.find('#openPopup').at(0).simulate('click');
    expect(component.find('Dialog').prop('hidden')).toEqual(false);
    component.find('#cancel').at(0).simulate('click');
    expect(component.find('Dialog').prop('hidden')).toEqual(true);
    component.find('#openPopup').at(0).simulate('click');
    component.find('#delete').at(0).simulate('click');
    expect(completeMock).toHaveBeenCalled();

    component.unmount();
  });
});
```

Figur 23 - Skjermdump av enhetstest for brukergrensesnitt

Figur 23 viser et eksempel på en enhetstest som gjennomfører en serie interaksjoner for så å avslutte med å teste om komponenten vil ha utført forventet funksjonalitet ved å kalle på *mock*-funksjonen.

Test av API-kall

Når en hook skal testes gjelder det samme som når de benyttes i applikasjonen, en hook må kalles i en komponent. For å unngå å opprette en *mocket* komponent for hver test benyttes testrammeverket fra React-hooks-testing-library. Ved hjelp av testrammeverket blir det mulig å utføre testing av hook'en direkte i testfilen.

Det er vesentlig større og mer komplekse data involvert i *API*-kallene enn hva tilfellet er for brukergrensesnittet. Derfor har vi opprettet en egen fil som vil *mocke* ut en responsverdi for hver av de spesialtilpassete hook'ene vi skal teste.

```
describe('useCreateSegmentConfiguration', () => {
  it('Creates a configuration for a platform of a variant of a segment', async () => {
    const contextWrapper = buildCreateSegmentConfigurationResponse(data);

    const { result, waitForNextUpdate } = renderHook(
      () => useCreateSegmentConfiguration({ path: params, body: data }),
      {
        wrapper: contextWrapper.wrapper.context,
      }
    );

    expect(result.current.loading).toBeTruthy();

    await waitForNextUpdate();

    expect(result.current.loading).toBeFalsy();
    expect(result.current.result).toStrictEqual(data);
  });
});
```

Figur 24 - skjermdump av enhetstest for API-kall

For å teste hook'en benyttes funksjonen «renderHook» fra React-hooks-testing-library. Denne funksjonen lar testen få tilgang til all logikk i den spesialtilpassete hook'en, uten at vi trenger å kalle denne i en komponent.

Responsen vi ønsker å sammenligne testen vår med vil ofte være en *mocket* konfigurasjonsfil. Det opprettes derfor en kontekst i testen som renderHook kan benytte for sammenligningen, slik at den *mockete* konfigurasjonsfilen ikke må skrives på nytt i sin helhet.

Api-kallene utføres asynkront. Det vil si at *promise*-funksjonen illustrert i figur 22 først returnerer et resultat etter at *API*-kallet er ferdig utført. Den lille forsinkelsen som da oppstår må også testen ta høyde for, og til dette benyttes funksjonen «waitForNextUpdate» fra React-hooks-testing-library.

Hvordan disse stegene blir satt sammen til en ferdig test viser figur 24 et eksempel på. Testen i figuren sjekker både om den asynkrone funksjonen gjennomføres som forventet, og at de returnerte data stemmer overens med forventet respons.

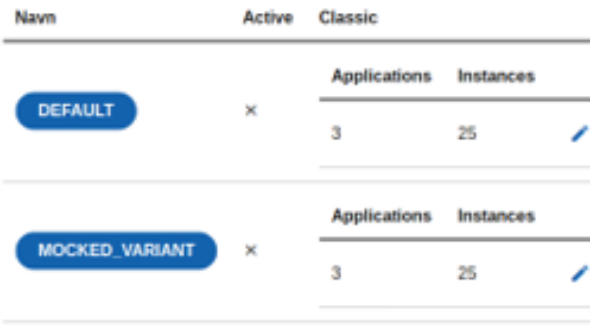
I tillegg testes også feilmeldingene for alle hook'ene. Ved å sende inn en HTTP-status, for eksempel 404, sjekker testen at forventet feilmelding returneres.

Videre utvikling

Prosessdokumentasjonen nevner flere punkter som gjenstår i utviklingen av applikasjonen. Det har derfor vært viktig for oss å produsere en skalerbar og oversiktlig kode, i tillegg til å utvikle de funksjonalitetene vi selv har hatt som mål.

Hovedsidene for både miljøer og segmenter benytter en tabell som henter informasjon fra komponenter lenger ned i hierarkiet. Den enkle løsningen vil være å legge til filtrerings- og søkefunksjonalitet direkte i tabellen for å behandle dataene denne viser. En mer optimalisert løsning vil være å utvikle nye *endepunkter* med filtrering implementert direkte i *API*-kallet. Slik kan GET-forespørslene mot *backend* utvides til å ta imot filtreringsparametere, og dermed unngår man å returnere unødvendige data. Denne løsningen vil kreve noe mer utviklingsarbeid, men sluttresultatet er en mer effektiv applikasjon.

For å visualisere hvilke segmenter og miljøer som er koblet sammen har vi allerede opprettet et felt i varianttabellen som skal visualisere hvorvidt varianten er aktiv eller ikke, vist i figur 25. Denne visualiseringen mangler *endepunkt* fra *API*'et før den kan tas i bruk.



Figur 25 - Skjermdump av varianttabell med felt "Active"

For miljøtabellen er det allerede visualisert hvor mange segmenter den aktuelle miljøplanen inneholder. Når de nye *endepunktene* er klare kan tabellen enkelt utvides med et felt for å visualisere hvorvidt en variant er koblet mot et spesifikt miljø gjennom nevnte segmenter.

Avslutning

Oppsummering

Vi har vært gjennom en svært lærerik prosess, og utviklet et velfungerende produkt. Vårt endelige produkt svarer godt til kravspesifikasjonen, og det er tilrettelagt for skalering og videre utvikling av applikasjonen når all funksjonalitet blir gjort tilgjengelig fra *endepunktene*.

Vi er fornøyde med våre vurderinger rundt prioriteringer i prosjektet, som har gjort det mulig for oss å ferdigstille en fungerende applikasjon. Om vi hadde forsøkt å gape over det fullstendige omfanget av Skatteetatens prosjekt er det ikke sikkert vi hadde kommet i mål med et vellykket prosjekt.

Prosjektet vårt har måttet rette seg etter noe strengere rammer og føringer for teknologi- og designvalg enn hva normalen er for studentprosjekter. I de tilfellene vi har mistet noe valgfrihet for utviklingen har vi fått tilbakebetalt i form av innsikt i avanserte teknologier som det ikke er sikkert vi hadde oppsøkt på egenhånd.

Gjennom denne prosessen har vi fått verdifull førstehåndserfaring fra hvordan et moderne utviklingsteam jobber. Vi har deltatt på møter, demonstrasjoner og diskusjoner. Flere av teknologiene vi har benyttet og satt oss inn i er svært relevante i arbeidsmarkedet.

Bruken av React med Typescript og hooks er noe vi har hatt stor glede av. Sammenlignet med mer tradisjonell utvikling i Javascript er logikken og arkitekturen annerledes bygget opp i disse mer moderne teknologiene. For oss har disse forskjellene medført at terskelen for å benytte teknologiene var noe høy i starten av prosjektperioden. Når vi nå omsider har fått en bedre forståelse av Typescript og hooks har vi imidlertid virkelig satt pris på de enorme mulighetene for fleksibel og robust kode disse teknologiene tilbyr.

Som vi nevnte i starten av denne rapporten har selve arbeidet med prosjektet vært veldig inspirerende. Det har vært motiverende å se hvordan teori vi har lært på skolen kan omsettes til applikasjoner med stor nytteverdi i praktisk arbeid.

Dette semesteret har vært spesielt på mange måter. Den første delen av utviklingsprosessen vår har vært utført i Skatteetatens lokaler. Den siste delen av utviklingsprosessen har grunnet situasjonen rundt Covid-19 blitt utført på hjemmekontor. Etter å ha tilegnet oss mye arbeidserfaring på relativt kort tid har vi derfor måttet omstille oss til å være løsningsorienterte og finne fleksible løsninger for samarbeid i en hverdag hvor man ikke lenger kan kommunisere ansikt til ansikt uten forbehold.

Refleksjoner

Da vi gikk inn i dette prosjektet med Skatteetaten hadde vi med oss visse forventninger. Forventningene vi satt med var at vi kom til å møte et profesjonelt arbeidsmiljø, at vi skulle få erfaring med prosessen bak systemutvikling, og å se teorien vi har lært gjennom studiet bli praktisert.

Som nevnt i prosessdokumentasjonen er arbeidet i Skatteetaten utført etter smidige prinsipper basert på Kanban-metodikk. Flere av fremgangsmåtene vi har benyttet har bidratt til å forenkle og effektivisere arbeidsprosessen.

Gjennom hyppig bruk av Kanban-brettet har vi holdt god oversikt over oppgaver som skal gjøres, oppgaver som er under arbeid og utførte oppgaver. Gjennom denne prosessen har vi også brukt *QA* etter hver oppgave har blitt gjennomført, og slik har vi kunnet kvalitetssikre utført arbeid. *QA* kan gi noe av samme nytteverdi som parprogrammering, men i tillegg beholdes fleksibiliteten ved at alle utviklere kan jobbe individuelt. *QA*-prosessen blir alltid utført av en annen utvikler enn den som har utført oppgaven, og man får dermed kvalitetssikret arbeidet av flere utviklere. Disse tilbakemeldingene har bidratt til at vi fortløpende har fått konstruktiv kritikk på arbeidet, noe som igjen har medført økt kodekvalitet og maksimalt læringsutbytte.

Daglige stand-ups hører også med i Skatteetatens metode. Disse møtene har gitt oss mulighet til å ha full oversikt innad i utviklingsteamet. Vi har blitt oppdatert på hva de andre utviklerne jobber med, og vi har måttet fortelle hvilke oppgaver vi selv jobber på. Dette har gjort det lettere for oss å planlegge dagen med teamet. De gangene vi har støtt på problemer og blitt stående fast med disse, har vi kunnet ta opp dette på stand-up'en, og fått hjelp til å løse problemet etter møtet.

Bruk av Kanban med stand-ups har for oss vært en svært effektiv og god arbeidsprosess. Det har gjennom hele prosjektperioden vært enkelt å koordinere oppgavene, og sammen sette av tid til å finne løsninger på problemer som har dukket opp underveis.

En potensiell ulempe ved en slik Kanban-metodikk er at det kan oppstå et visst prestasjons- og effektivitetspress under utviklingen. Det blir veldig synlig og målbart hvem som skriver mest kode og løser flest oppgaver. Et slikt prestasjonsjag kan igjen medføre at det blir fristende å kaste seg ut i nye oppgaver uten å sette av tid til grundig forarbeid.

I etterkant av prosjektarbeidet har vi sett at det kunne vært gunstig å inkludere en prosess med brukertesting etter vitenskapelig metode for å kvalitetssikre arbeidet og designet vårt. Utviklingen har vært gjennomført som en iterativ prosess i nært samarbeid med teamet som skal utvikle applikasjonen videre og drifte denne. Vi har derfor følt vi har hatt god kontroll på ønsker, bruksområder og tilgjengelighet i applikasjonen, og prioriteten har vært på å utvikle den komplette funksjonaliteten fra kravspesifikasjonen.

Forbedringspotensial

Utviklingen av dette prosjektet tok utgangspunkt i en svært generell beskrivelse av oppgaven. Derfra har vi justert oppgavens omfang og kravspesifikasjonen etter den tiden vi har hatt til rådighet. For oss har disse vurderingene vært strengt nødvendige, men det har selvsagt også medført at enkelte utviklingsoppgaver gjenstår før applikasjonen er klar for å settes i drift på det interne systemet.

I starten av prosjektperioden gikk det med mye tid til prøving og feiling mens vi gjorde oss kjent med nye teknologier. Om vi skulle startet et lignende prosjekt på nytt nå, ville vi selvsagt hatt mye bedre forutsetninger for å jobbe mer strukturert og effektivt fra første stund. Dermed hadde vi også hatt mulighet til å utføre flere prosesser for undersøkelser og testing, eller implementere flere funksjoner i løpet av prosjektperioden.

Som nevnt i refleksjonene kunne vi med fordel ha inkludert brukertesting. En slik testprosess ville ytterligere sikret og dokumentert kvaliteten på applikasjonen. At det var utvikling og oppnåelse av funksjonalitet fra kravspesifikasjonen som ble prioritert kan tyde på at vi i neste prosjektarbeid må vurdere å være enda mer selektive i avgrensingene for prosjektets rammer. Til tross for at vi har en god følelse rundt prioriteringen av funksjonalitet i dette prosjektet, er det alltid viktig å vurdere hvilke prosesser eller funksjonaliteter som er viktigst i hvert enkelt prosjekt. Om vi hadde siktet på litt færre måloppnåelser kunne vi også fått tid til å gjøre hver oppgave enda grundigere og gjennomføre både brukerundersøkelser og brukertester.

Konklusjon

I dette prosjektet har vi dedikert oss fullt ut til å skaffe oss erfaring fra samarbeid i utviklingsteam, og skaffe oss erfaring med nye og attraktive teknologier.

Vi sitter igjen med mye ny kunnskap som komplementerer pensum fra de teoretiske fagene vi har vært gjennom på en veldig god måte.

Vi har begge fått forsterket inntrykket av at webutvikling er noe vi ønsker å gå videre med etter gjennomførte studier.

Applikasjonen vi har utviklet i dette prosjektet inneholder den aller viktigste funksjonaliteten for å fungere som en god *frontend*-løsning for Storyteller. Applikasjonen er allerede fullt fungerende, og er enkelt skalerbar for videre utvikling og forbedringer. Etter hvert som *backend'en* i Storyteller også videreutvikles vil flere *endepunkter* gjøres tilgjengelig og det er godt tilrettelagt for at disse kan implementeres i *frontend*-applikasjonen.

Gjennom et tett samarbeid og daglige møter med teamet som skal overta applikasjonen er vi også sikre på at løsningen holder en høy kvalitet, til tross for at ytterligere testing ikke har blitt gjennomført. I løpet av en relativt kort prosjektperiode er det vanskelig å unngå tøffe prioriteringer, og disse prioriteringene vil alltid føre til et visst kompromiss mellom funksjonalitet og andre prosesser. Når alt kommer til alt

føler vi at vi har fått et godt læringsutbytte og levert et godt resultat ved å prioritere å utvikle mest mulig av funksjonaliteten som var ønsket i denne applikasjonen.

Tilbakemeldingene fra utviklerne hos Skatteetaten har vært svært positive, og vår *frontend*-applikasjon er allerede i bruk for å demonstrere hvordan Storyteller fungerer. Applikasjonen vi har utviklet kommer også til å bli brukt som grunnlag for den endelige *frontend*-løsningen når Storyteller blir satt i drift. En attest fra Skatteetaten finnes i vedlegg D.

Prosjektet med Skatteetaten har i aller høyeste grad svart til forventningene vi hadde. Vi har lært mye, og fått erfare hvordan teoretisk kunnskap om systemutvikling henger sammen med det praktiske arbeidet. Vi har fått se hvordan det er å jobbe under Kanban-metodikken, og blitt inkludert i arbeidshverdagen til et moderne og erfarent utviklingsteam. Det har vært svært lærerikt å delta og bidra på felles møter og diskusjoner. Denne erfaringen er i kombinasjon med kunnskap om ny teknologi noe vi er sikre på at vi vil dra stor nytte av i arbeidslivet.

Vedlegg

Innhold

Vedlegg A: Ordliste

API
Programmeringsgrensesnitt som tillater frontend å kommunisere med backend.

Backend
Den delen av programvaren som ligger nærmest databasen. Det er her logikken for databehandling ligger.

Backlog
Liste over oppgaver som benyttes ved smidig utvikling.

DOM
Document Object Model er en datamodell og et API for HTML- eller XML-dokumenter.

Endepunkt
En nettadresse benyttet av API for å få tilgang til spesfikke operasjoner i backend.

Filsti
En detaljert beskrivelse av hvor i filhierarkiet en bestemt fil kan finnes.

Flexbox
Metode i stilark for plassering av et DOM-element. Benyttet for å lage et fleksibelt og responsivt design.

Frontend
Den delen av programvaren som ligger nærmest brukeren. Det er her det brukeren ser og kan gjøre ligger.

Interface
En definisjon på formatet og innholdet for et objekt i Typescript.

JSX
Javascript XML, brukt for å beskrive brukergrensesnittet og skrive HTML i React.

Kontekst (React)
Alternativ metode for å sende data gjennom en React applikasjon, unngår å manuelt sende data gjennom hele filstien.

Linter
Verktøy som sjekker kildekode for formattering og kjente feil.

Livssyklus
Faser av implementasjon for en komponent i React. Komponenten kan være “mounting”, “updating” eller “unmounting”.

Lo-fi prototype
For eksempel skisser for å teste design og interaktsjon før utviklingen starter

Micro frontends
Flere selvstendige og uavhengige webappliksjoner som eksisterer side om side som et stort, samlet system.

Mock
Etterligning, brukt i testing for å forenkle kjøring av en komponent.

Modul (React)
En pakke, tilgjengelig via npm, som tilbyr elementer eller funksjonalitet for direkte implementasjon i applikasjonen.

MVP
Minste brukbare produkt. Et godt utgangspunkt for prioriteringer i starten av utviklingen.

Path (React-router-dom)
Nettadresse som brukes som en variabel for navigasjon i React-router-dom.

Promise
En funksjon som utfører en asynkron operasjon, og behandler resultatverdien som et synkront objekt. Returnerer enten et resultat eller en feilmelding avhengig av om innsendt promise blir besvart.

Props (React)
Verdi som sendes inn i en komponent når den rendres.

QA
Spørsmål og svar etter en utført oppgave i Jira Kanban board. Brukes for å kvalitetssikre arbeidet.

React-router-dom
npm pakke for React som tilbyr funksjonalitet for ruting mellom forskjellige

nettadresser med tilhørende komponenter.

Referanse (React)
Defineres som et punkt i DOM. En enkel måte å få tilgang til innhold i et element eller en node.

Repository
Oppbevaring og lagring av kode. I programvare for versjonshåndtering inneholder repository både alle versjoner av koden, og metadata om denne.

Ruting
Prosessen med å styre trafikken i et nettverk til riktig destinasjon.

Script
Automatisering av enkle oppgaver som vil repeteres mange ganger. For eksempel prosedyre for å starte applikasjonen i utviklingsmiljøet.

State (React)
Statusen for en komponent og tilhørende statusvariabler. Oppdateres og vedikeholdes internt i komponenten.

Store (Redux)
En fil som håndterer state for en stor applikasjon.

Switch (react-router-dom)
Komponent i React-router-dom som brukes for å rendre komponenter basert på path fra nettadressen.

Ternary operator
En operasjon som tar imot tre argumenter. Det første argumentet sjekkes til “true” eller “false”, og argument to eller tre returneres som verdi avhengig av resultatet.

Token
En elektronisk nøkkel for å gi tilgang på en trygg og sikker måte.

Type
Format på innholdet for en variabel eller konstant i Typescript. Definerer hva slags data som kan tilskrives en variabel.

Vedlegg B: Kravspesifikasjon

Ved oppstart av prosjektet var kravspesifikasjonen svært generell, og med en tydelig hensikt om at kravene ville utvikles og forbedres gjennom prosjektperioden. Det overordnete målet har hele veien vært å lage en enkel og effektiv frontend-løsning for funksjonaliteten som tilbys i applikasjonen Storyteller.

Kravspesifikasjonen som presenteres i dette dokumentet er de kravene vi i fellesskap med vår veileder og fagansvarlige hos Skatteetaten har kommet frem til underveis i arbeidet. De fleste av kravene var klare fra starten av utviklingen, mens enkelte krav har ved hjelp av smidig utviklingsmetodikk kommet til etter hvert som applikasjonen tok form.

Funksjonelle krav

- Som drifter skal man raskt og enkelt kunne opprette, og sanere felles miljøer.
- Som drifter skal man raskt og enkelt kunne opprette og redigere segmenter.
- Konfigurasjonsfiler skal være tilgjengelige i både Json- og Yaml-format.
- En bruker skal alltid få relevant tilbakemelding på status og feil i applikasjonen.
- En bruker skal enkelt kunne orientere og navigere seg basert på erfaring fra Skatteetatens andre applikasjoner.

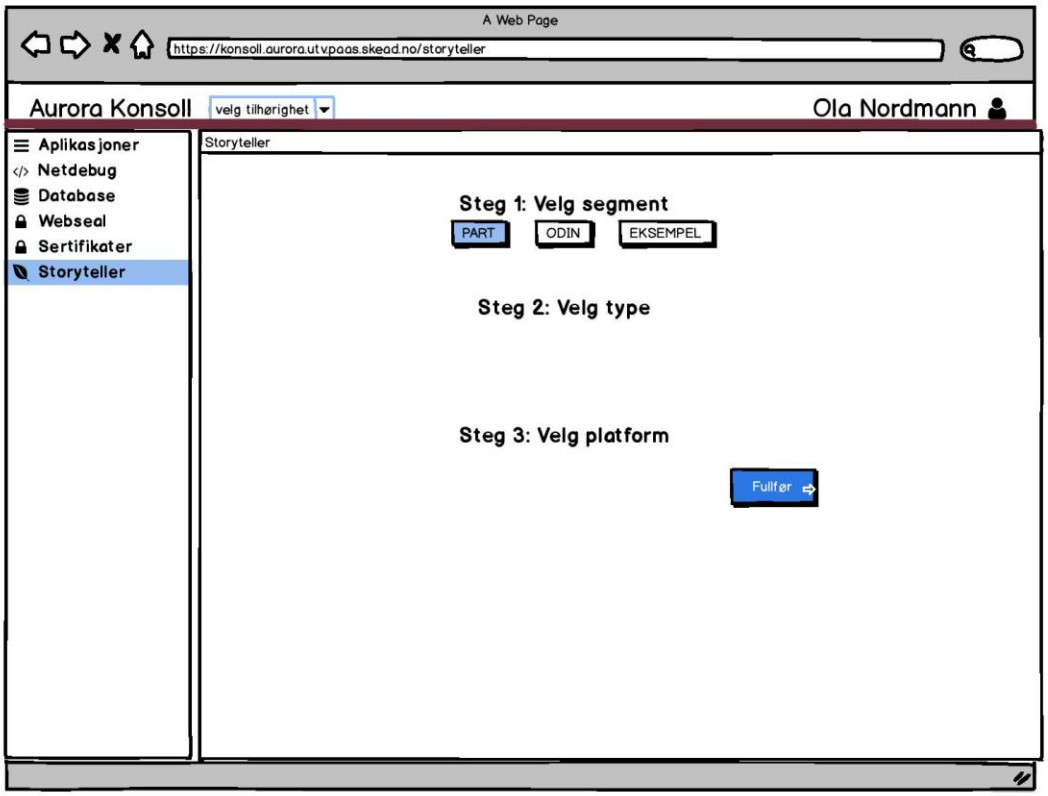
Ikke-funksjonelle krav

- God frontend-design
- Applikasjonen skal følge Skatteetatens designsystem
- Applikasjonen skal bruke minst mulig nettverksressurser, uten å gå på bekostning av funksjonalitet og produktivitet.
- Applikasjonen skal utvikles i React med Typescript
- Koden skal være godt strukturert med selvforklarende bruk av navn på komponenter og variabler.
- Koden skal formatteres ved hjelp av en linter
- Applikasjonen må støtte parallellisering for å minimere tidsbruk for operasjoner.
- Applikasjonen må ha høy grad av skalerbarhet for videre utvikling.

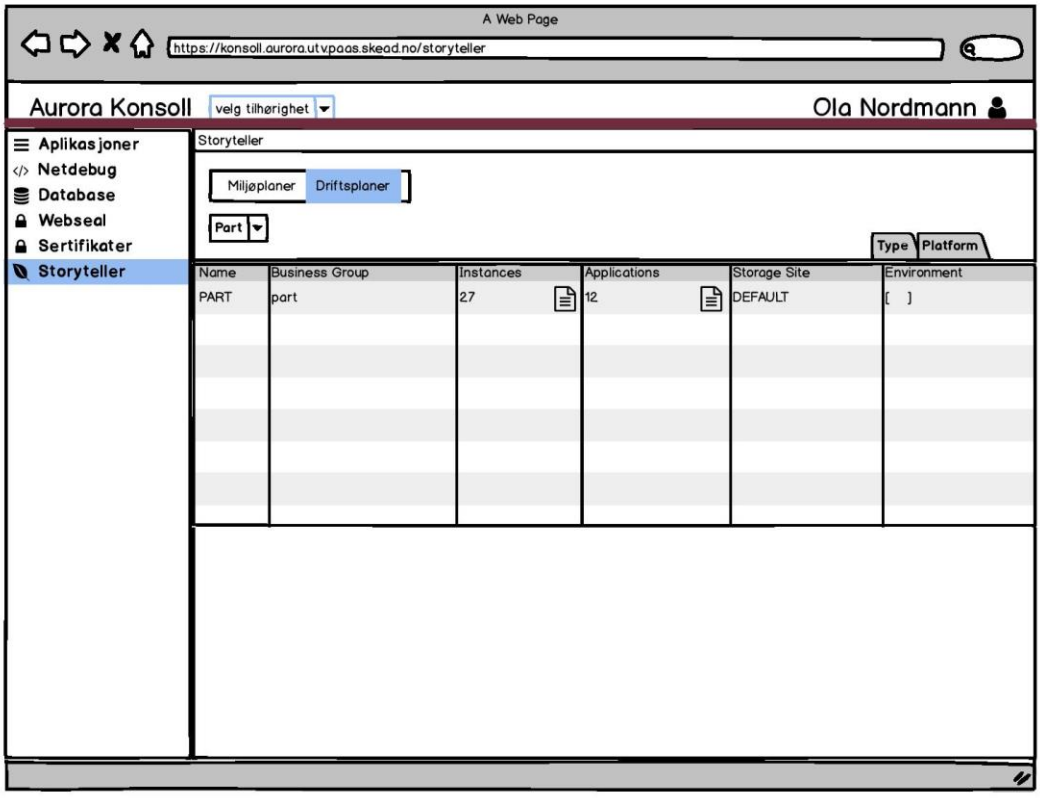
Vedlegg C: Skisser

C.1

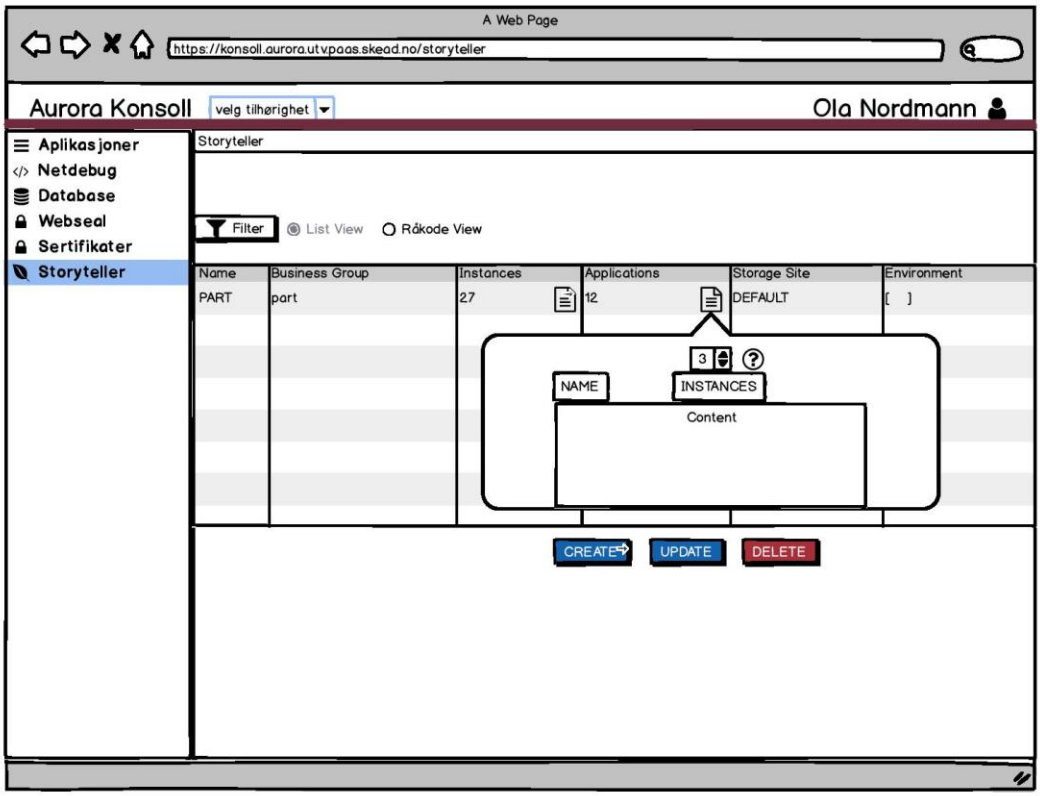
start



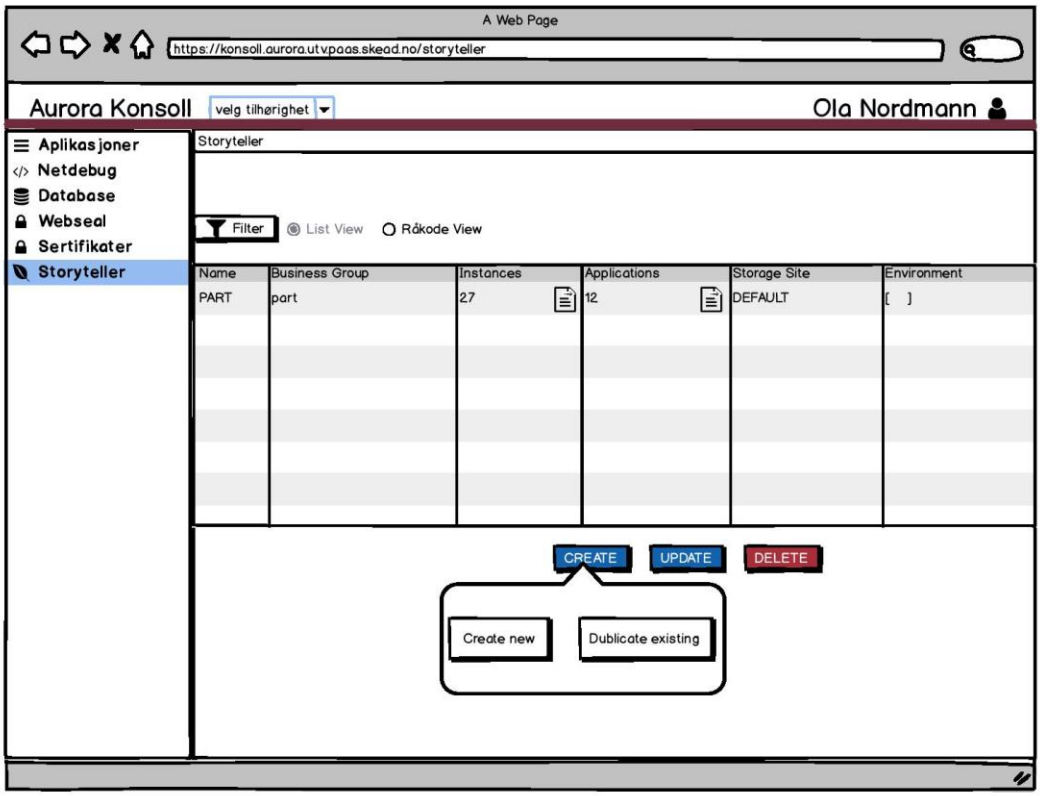
start (Alternate 773v)



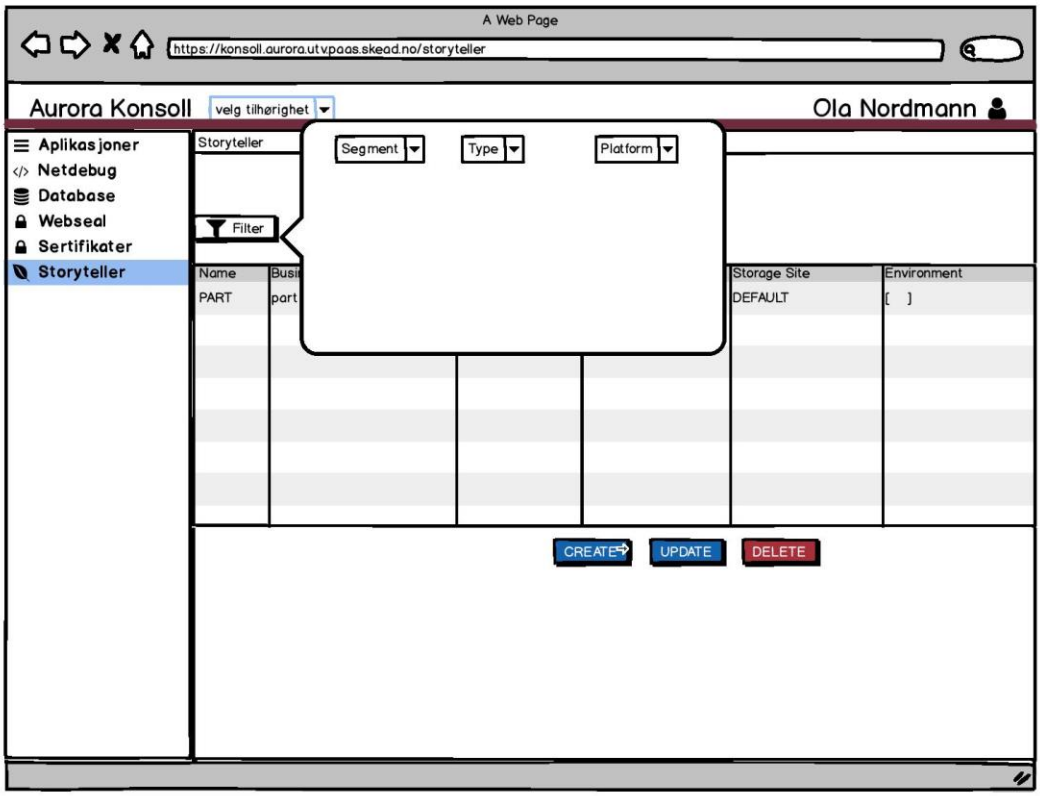
popup app



Create

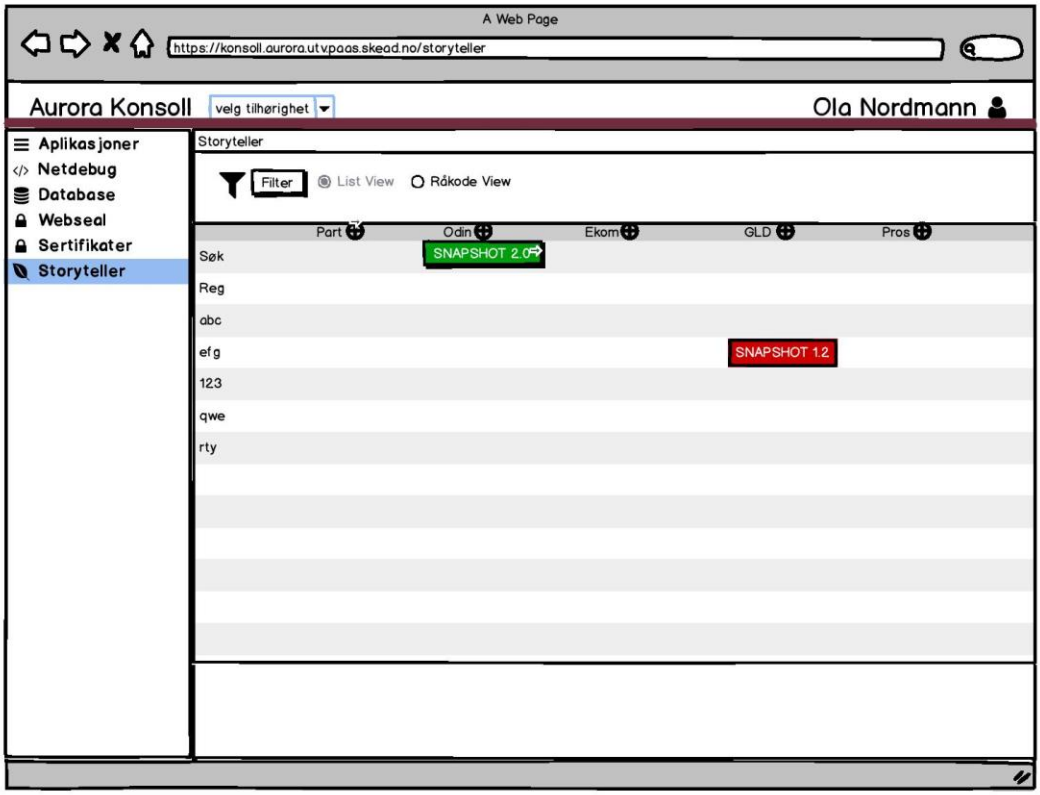


filter popup

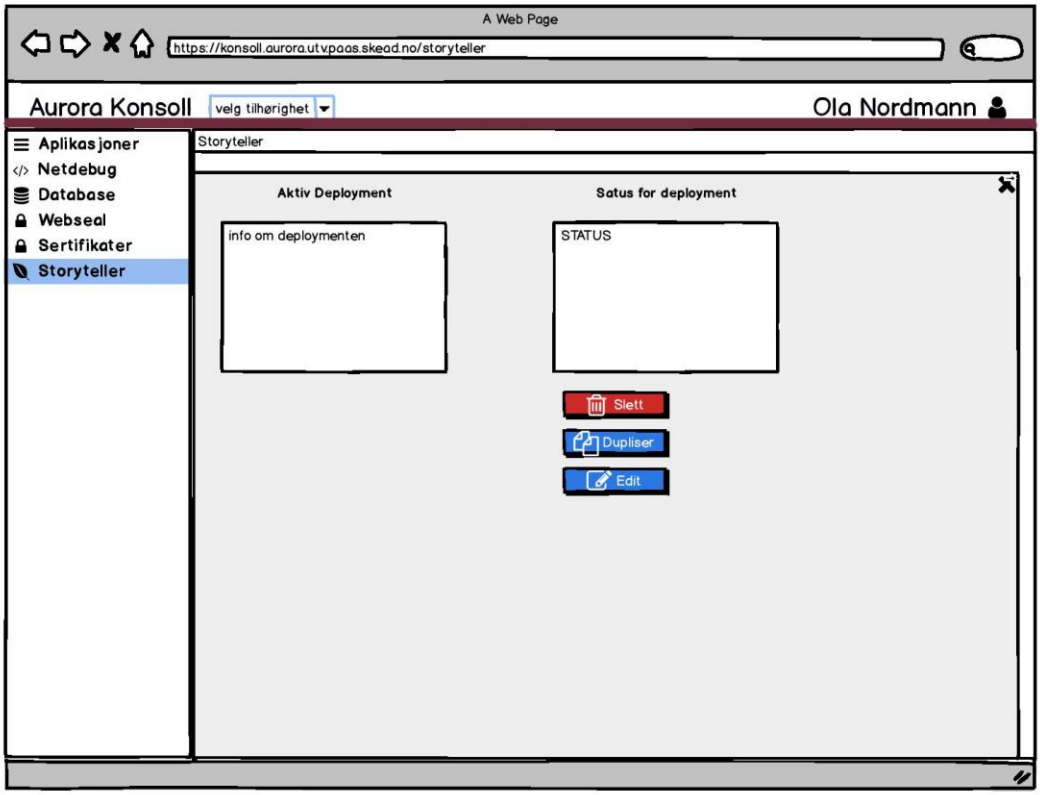


C.2

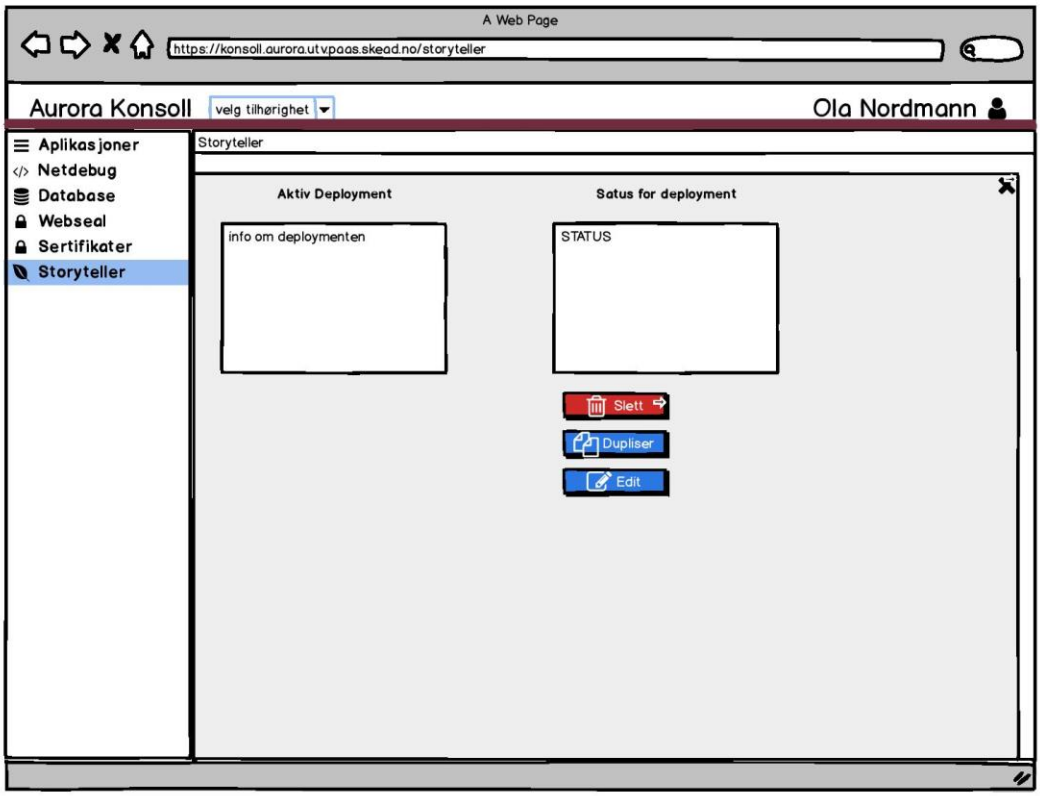
Start



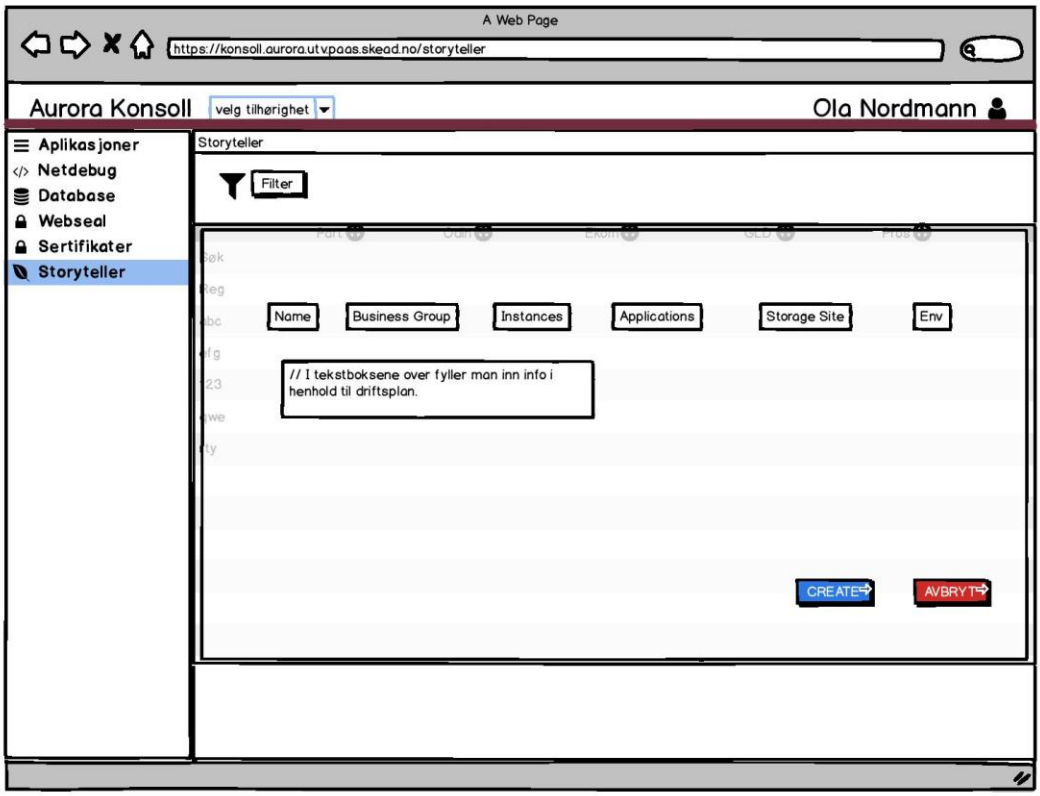
info page



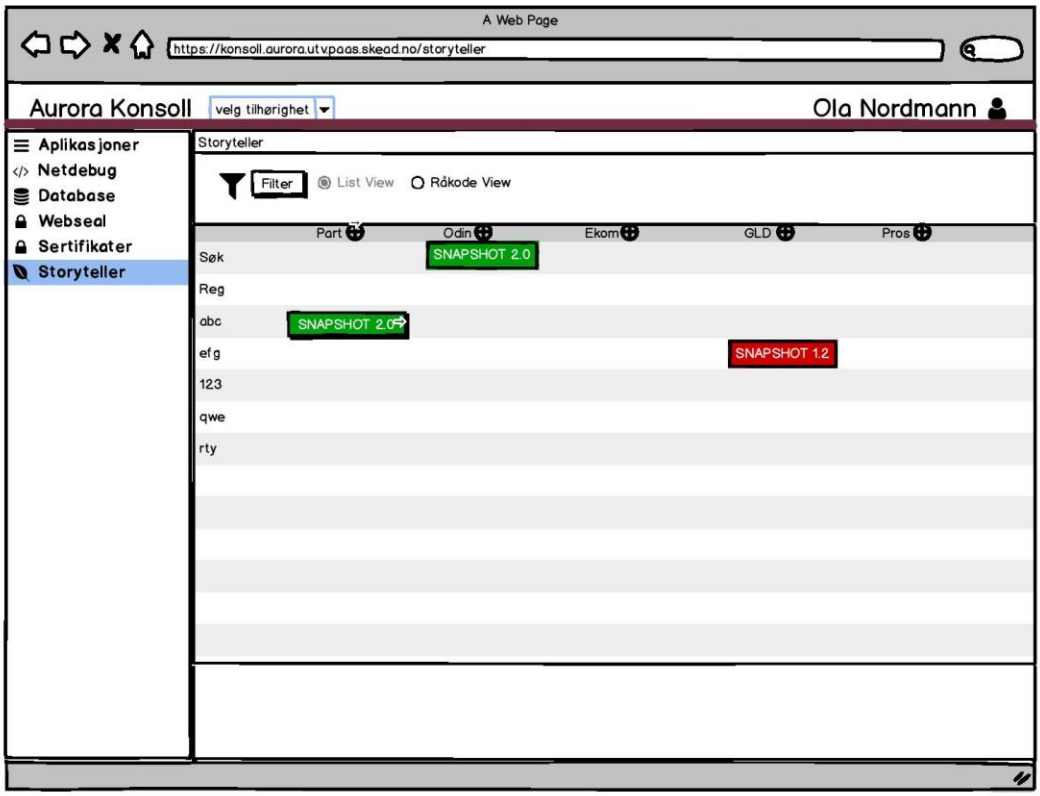
info page delete part



add part

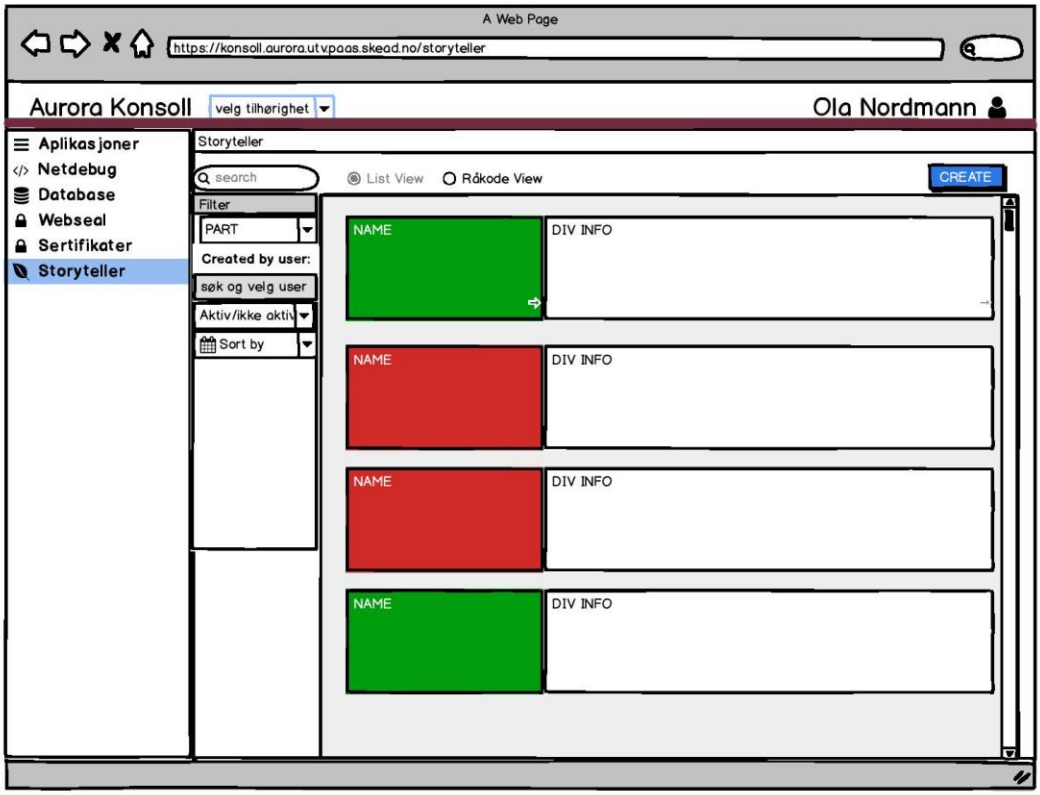


part added

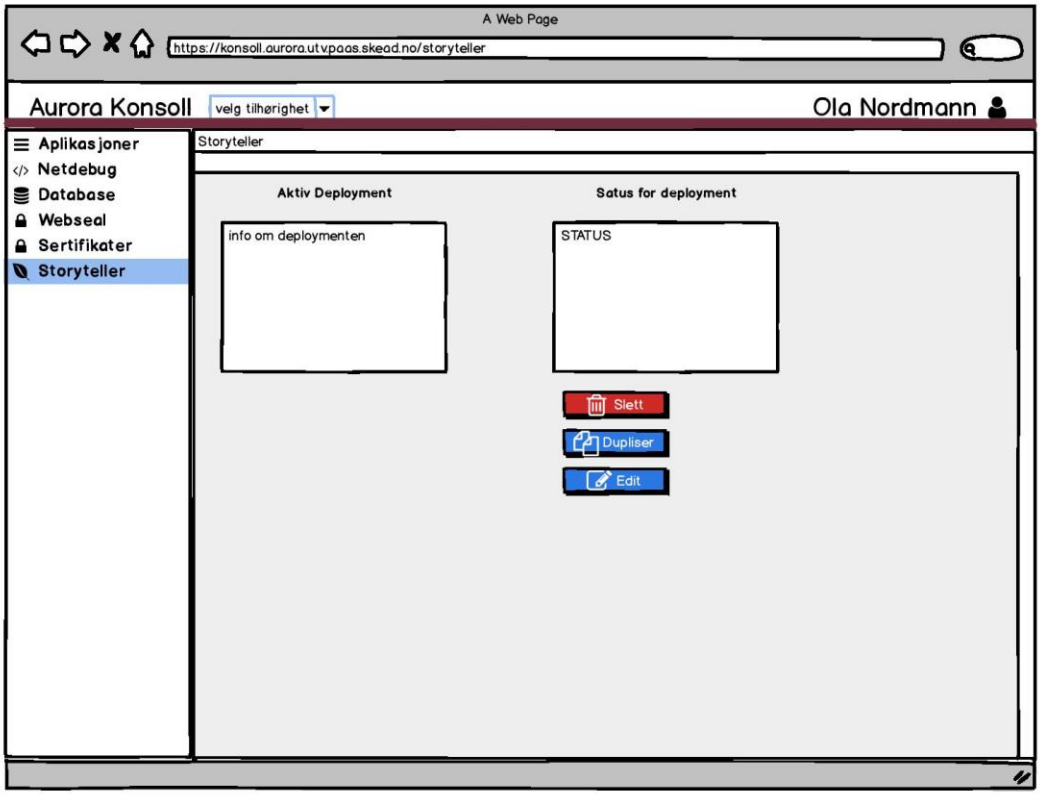

A Web Page
https://konsoll.aurora.utvpaas.skead.no/storyteller
Aurora Konsoll
velg tilhørighet
Ola Nordmann
Aplikasjoner
Netdebug
Database
Webseal
Sertifikater
Storyteller
Storyteller
Filter
List View
Råkode View
Part
Odin
Ekom
GLD
Pros
Søk
SNAPSHOT 2.0
Reg
abc
SNAPSHOT 2.0
efg
SNAPSHOT 1.2
123
qwe
rty

C.3

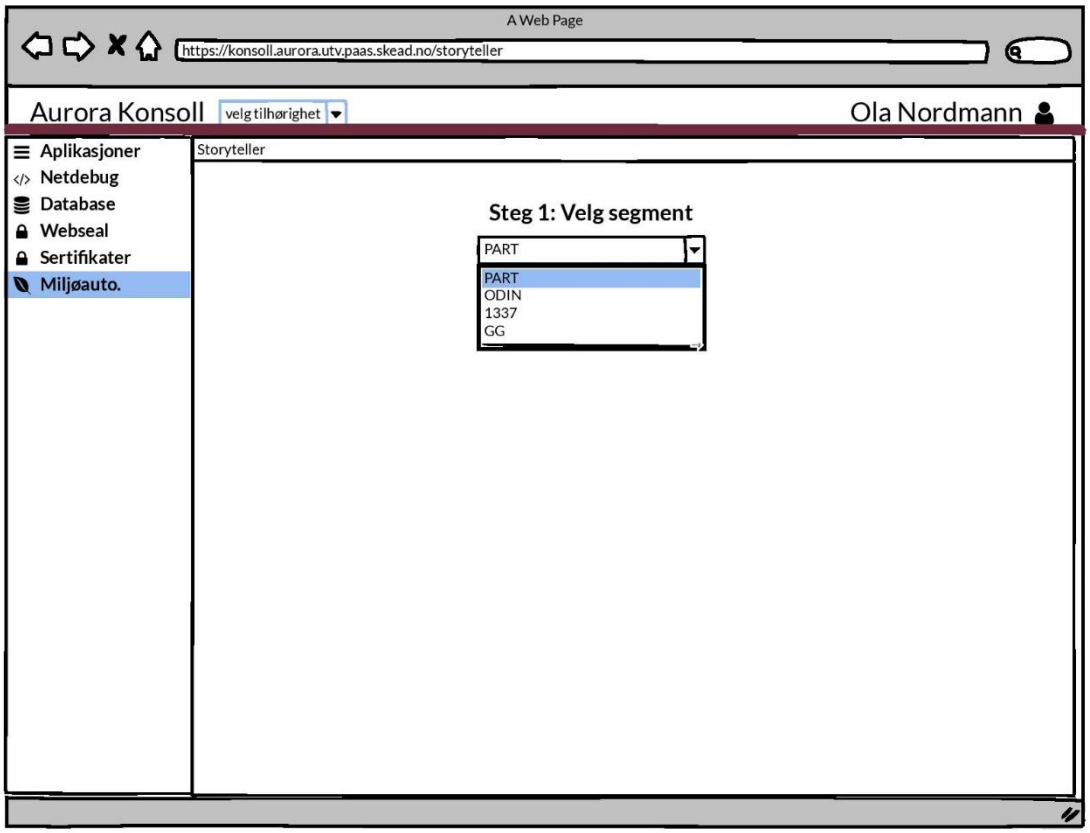
Start

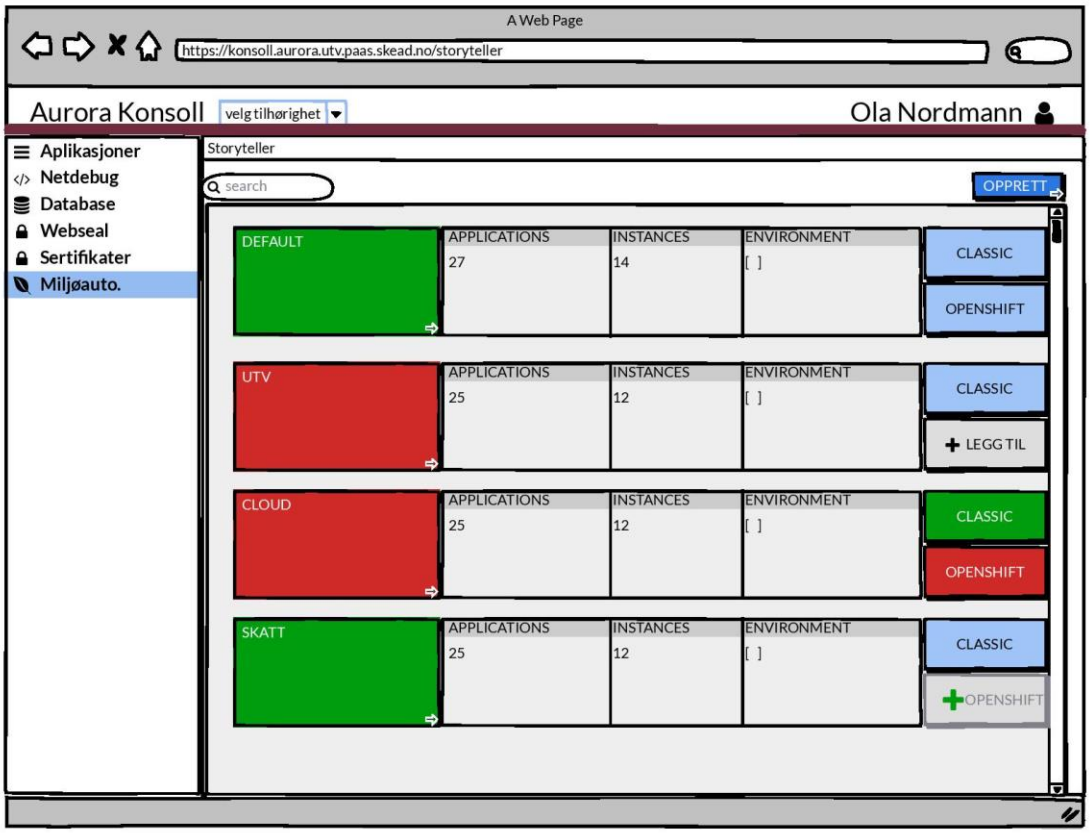


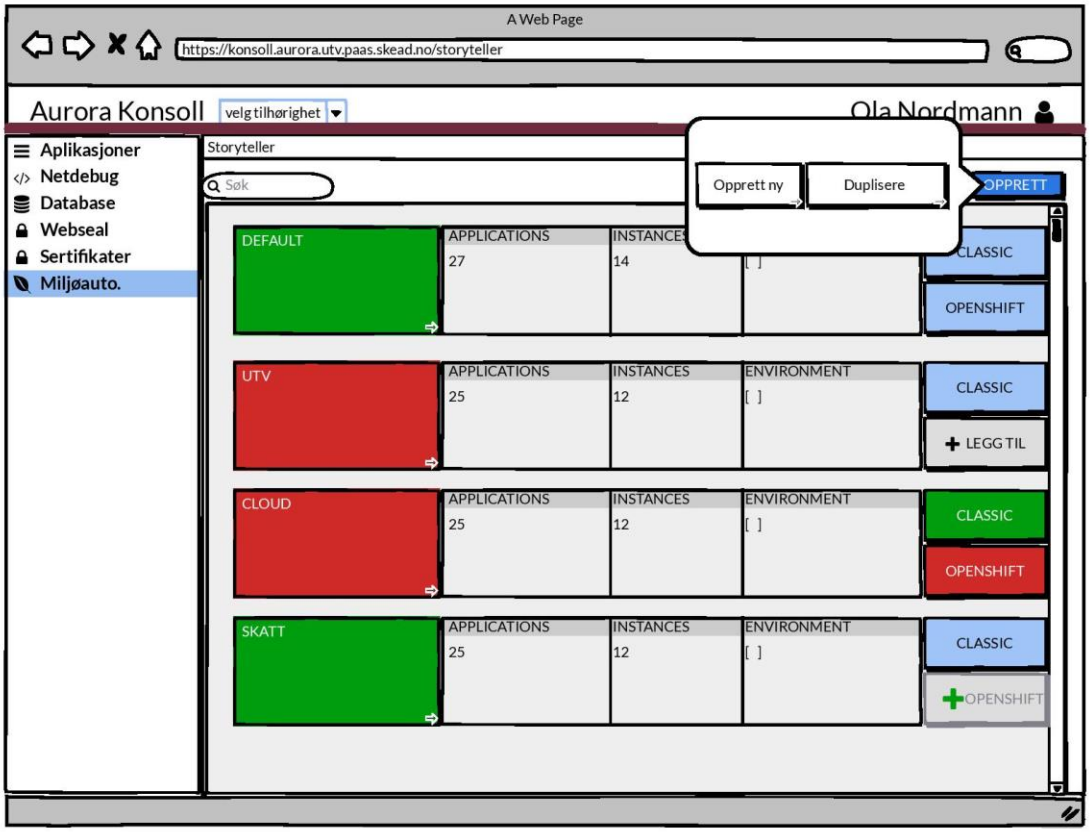
Start copy

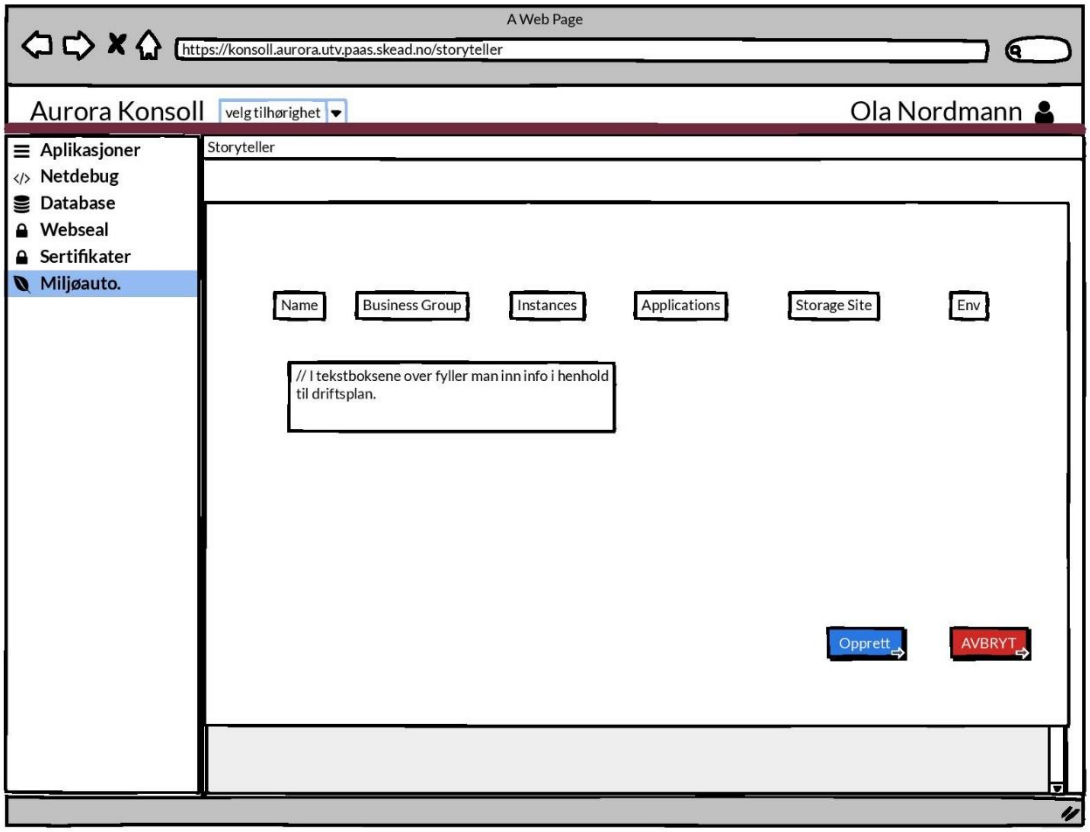


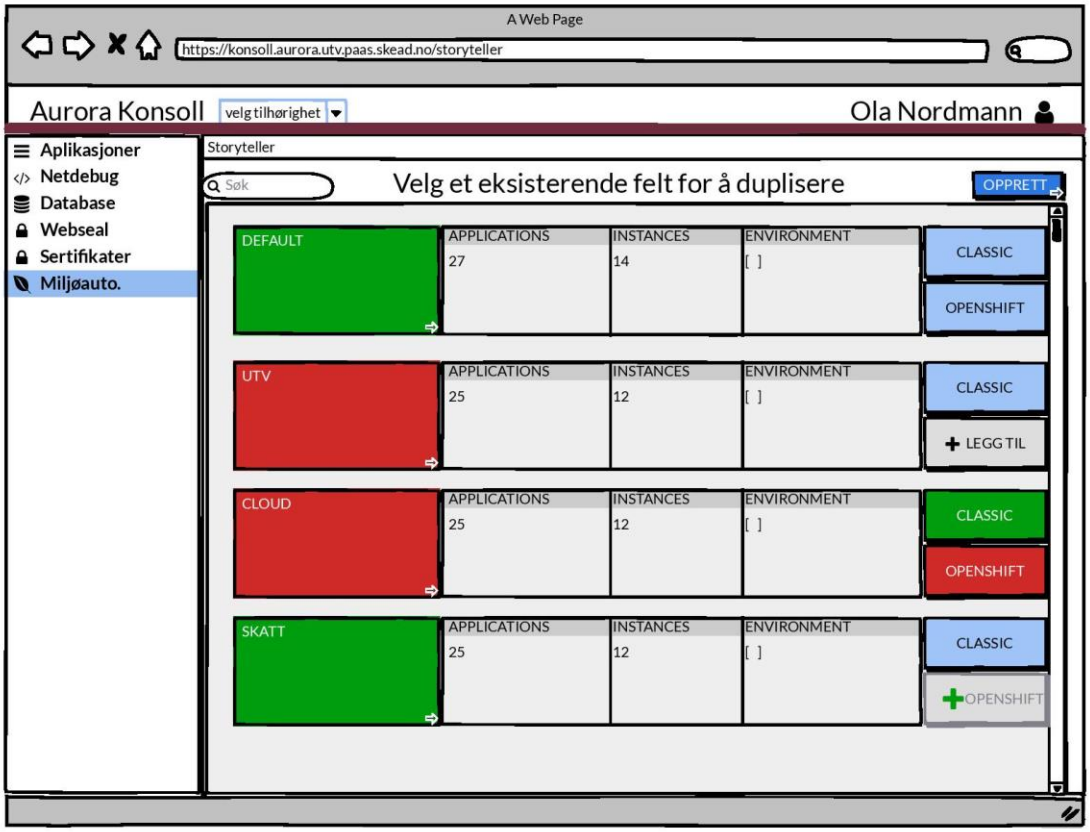
C.4

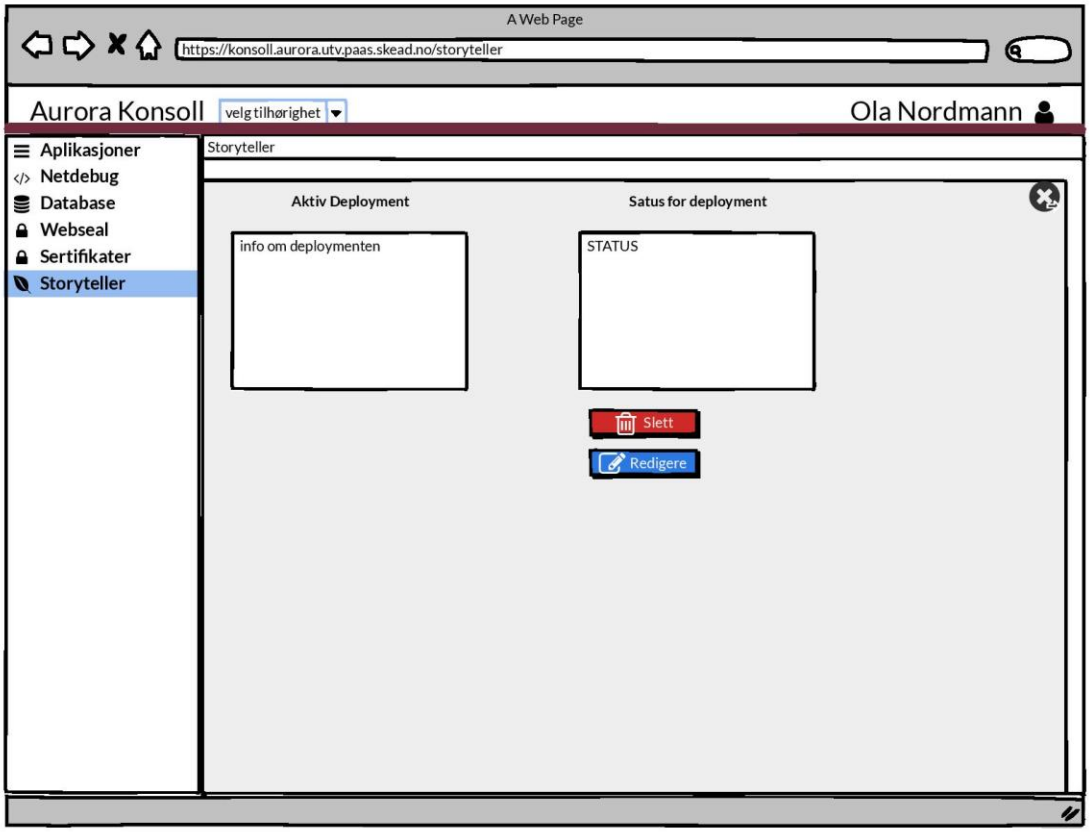

A Web Page
https://konsoll.aurora.utv.paas.skead.no/storyteller
Aurora Konsoll
velg tilhørighet
Ola Nordmann
Aplikasjoner
Netdebug
Database
Webseal
Sertifikater
Miljøauto.
Storyteller
Steg 1: Velg segment
PART
PART
ODIN
1337
GG


A Web Page
https://konsoll.aurora.utv.paas.skead.no/storyteller
Aurora Konsoll
velg tilhørighet
Ola Nordmann
Aplikasjoner
Netdebug
Database
Webseal
Sertifikater
Miljøauto.
Storyteller
search
OPPRETT
DEFAULT
APPLICATIONS
27
INSTANCES
14
ENVIRONMENT
[ ]
CLASSIC
OPENSHIFT
UTV
APPLICATIONS
25
INSTANCES
12
ENVIRONMENT
[ ]
CLASSIC
LEGG TIL
CLOUD
APPLICATIONS
25
INSTANCES
12
ENVIRONMENT
[ ]
CLASSIC
OPENSHIFT
SKATT
APPLICATIONS
25
INSTANCES
12
ENVIRONMENT
[ ]
CLASSIC
OPENSHIFT


A Web Page
https://konsoll.aurora.utv.paas.skead.no/storyteller
Aurora Konsoll
velg tilhørighet
Ola Nordmann
Aplikasjoner
Netdebug
Database
Webseal
Sertifikater
Miljøauto.
Storyteller
Søk
Opprett ny
Duplisere
OPPRETT
DEFAULT
APPLICATIONS
27
INSTANCES
14
CLASSIC
OPENSHIFT
UTV
APPLICATIONS
25
INSTANCES
12
ENVIRONMENT
CLASSIC
LEGG TIL
CLOUD
APPLICATIONS
25
INSTANCES
12
ENVIRONMENT
CLASSIC
OPENSHIFT
SKATT
APPLICATIONS
25
INSTANCES
12
ENVIRONMENT
CLASSIC
OPENSHIFT


A Web Page
https://konsoll.aurora.utv.paas.skead.no/storyteller
Aurora Konsoll
velg tilhørighet
Ola Nordmann
Aplikasjoner
Netdebug
Database
Webseal
Sertifikater
Miljøauto.
Storyteller
Name
Business Group
Instances
Applications
Storage Site
Env
// I tekstboksene over fyller man inn info i henhold til driftsplan.
Opprett
AVBRYT


A Web Page
https://konsoll.aurora.utv.paas.skead.no/storyteller
Aurora Konsoll
velg tilhørighet
Ola Nordmann
Aplikasjoner
Netdebug
Database
Webseal
Sertifikater
Miljøauto.
Storyteller
Søk
Velg et eksisterende felt for å duplisere
OPPRETT
DEFAULT
APPLICATIONS
27
INSTANCES
14
ENVIRONMENT
[ ]
CLASSIC
OPENSHIFT
UTV
APPLICATIONS
25
INSTANCES
12
ENVIRONMENT
[ ]
CLASSIC
LEGG TIL
CLOUD
APPLICATIONS
25
INSTANCES
12
ENVIRONMENT
[ ]
CLASSIC
OPENSHIFT
SKATT
APPLICATIONS
25
INSTANCES
12
ENVIRONMENT
[ ]
CLASSIC
OPENSHIFT


A Web Page
https://konsoll.aurora.utv.paas.skead.no/storyteller
Aurora Konsoll
velg tilhørighet
Ola Nordmann
Aplikasjoner
Netdebug
Database
Webseal
Sertifikater
Storyteller
Storyteller
Aktiv Deployment
Satus for deployment
info om deploymenten
STATUS
Slett
Redigere

Vedlegg D: Attest

Skatteetaten

ATTEST

Fredrik Haraldseth og Aleksander Andersen Skjelbred har i perioden 6.1.2020 til 25.5.2020 arbeidet med sin bachelor oppgave hos Skatteetaten.

De har hatt som oppgave å produsere en frontend-løsning for Storyteller.
Storyteller er et program for å automatisere opprettelse og sanering av testmiljøer internt i Skatteetaten.

Front-end løsningen er utviklet i React med typescript og hooks, og på Skatteetatens interne Openshiftbaserte plattform, Aurora.

Det har vært en glede å samarbeide med Fredrik og Aleksander i løpet av prosjektperioden. De er svært pliktoppfyllende og lærevillige, og har alltid møtt til jobb med et smil. Prosjektet de har utført har vært krevende, og under planleggingen ble vi enige i at de måtte sette seg inn i en del teknologier på forhånd slik at de var klare til oppstart. Dette gjennomførte de på en god måte, og stilte forberedt til første arbeidsdag. Senere i perioden ble vi alle nødt til å jobbe hjemmefra pga. Covid-19, og dette tok de også på strak arm. Sammen fant vi en god måte å fortsette samarbeidet, og det var fint å se at de fortsatt klarte å levere under uklare forhold.

Med dette går vi god for innholdet i Bachelor rapporten til Fredrik og Aleksander. Vi ønsker Fredrik og Aleksander lykke til med eksamen, videre i sin karriere og livet for øvrig.

Oslo 23.5.2020

Torhild Rørslett

Torhild Rørslett

Skatteetaten

Side 1 / 1

Vedlegg E: Kilder

Skatteetatens designsystem. (2020). *Designe og utvikle.* Hentet fra: https://skatteetaten.github.io/frontend-components/

Difi. (2020). *EUs webdirektiv.* Hentet fra: https://uu.difi.no/krav-og-regelverk/eus-webdirektiv-wad

Difi. (u.å.). *WCAG 2.1-standarden.* Hentet fra: https://uu.difi.no/krav-og-regelverk/webdirektivet-og-wcag-21/wcag-21-standarden

Skatteetaten Aurora. (u.å.). *The Aurora Openshift Platform.* Hentet fra: https://skatteetaten.github.io/aurora/documentation/openshift/

Openshift. (2020). *Openshift.* Hentet fra: https://www.openshift.com/

Docker. (2020). *Docker*. Hentet fra: https://www.docker.com/

Atlassian. (2020). *Jira.* Hentet fra: https://www.atlassian.com/software/jira

Atlassian. (u.å.). *Stand-ups for agile teams.* Hentet fra: https://www.atlassian.com/agile/scrum/standups

Atlassian. (u.å.). *Gitflow workflow.* Hentet fra: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Bitbucket. (2020). *Bitbucket.* Hentet fra: https://bitbucket.org/

React. (2020). *React*. Hentet fra: https://reactjs.org/

React. (2018). *Introducing Hooks.* Hentet fra: https://reactjs.org/docs/hooks-intro.html

Npm. (2020). *NPM.* Hentet fra: https://www.npmjs.com/

Typescript. (2020). *Typescript.* Hentet fra: https://www.typescriptlang.org/

Jest. (2020). *Jest.* Hentet fra: https://jestjs.io/

Github. (2020). *Enzyme*. Hentet fra: https://github.com/enzymejs/enzyme

Node.js. (2020). *Node.js.* Hentet fra: https://nodejs.org/en/

Swagger. (2020). *Swagger UI.* Hentet fra: https://swagger.io/tools/swagger-ui/

Postman. (2020). *Postman.* Hentet fra:
https://www.postman.com/

Visual Studio Code. (2020). *Visual Studio Code.* Hentet fra:
https://code.visualstudio.com/

Finn. (22.05.2020). *Søk på "react".* Hentet fra:
https://www.finn.no/job/fulltime/search.html?q=react

Finn. (22.05.2020). *Søk på "angular".* Hentet fra:
https://www.finn.no/job/fulltime/search.html?q=angular

Finn. (22.05.2020). *Søk på "vue".* Hentet fra:
https://www.finn.no/job/fulltime/search.html?q=vue

Github. (2020). *Create React App.* Hentet fra:
https://github.com/facebook/create-react-app

Single-spa. (2020). *single-spa*. Hentet fra:
https://single-spa.js.org/

Balsamiq. (2020). *Balsamiq wireframes.* Hentet fra:
https://balsamiq.com/wireframes/

Express. (2017). *Express.* Hentet fra:
https://expressjs.com/

Github. (2020). *Axios.* Hentet fra:
https://github.com/axios/axios

Redux. (2020). *Redux.* Hentet fra:
https://redux.js.org/

Ace. (2016). *Ace.* Hentet fra:
https://ace.c9.io/

Github. (2020). *CodeMirror.* Hentet fra:
https://github.com/codemirror/CodeMirror

ESLint. (2019). *ESLint.* Hentet fra:
https://eslint.org/

Styled components. (u.å.). *styled components.* Hentet fra:
https://styled-components.com/

Github. (2020). *React-hooks-testing-library.* Hentet fra:
https://github.com/testing-library/react-hooks-testing-library