Rapport:

Index

Det som er utfordrende med programmering er ikke å løse enkelte små problemer, men å jobbe på større prosjekter over lengre tid uten å miste oversikten. Det skjer lett at det blir mange kobliner mellom de ulike delene av koden slik at endringer som må gjøres, fører til store omstruktureringer. Siden vi har lite erfaring med å jobbe i gruppe på kodeprosjekt, antar vi at vi vil gjøre feil og tar steg for å begrense hvor lang tid det vil ta å fikse problemene.

Prosess

Vi bestemmer oss for domain driven development. Ved å splitte opp applikasjonen i ulike moduler, eller lag, med klart definerte grensesnitt, kan hver modul forstås seperat. Dette gjør at programmet ikke eksploderer i kompleksitet etter hvert som det vokser. Ved å følge dette «black box» prinsippet, kan deler av applikasjonen trygt endres innvendig, eller byttes ut uten at det påvirker andre deler av applikasjonen så lenge det ikke påvirker grensesnittene. Grensesnittene definerer hva data som går inn og ut av modulene og format på denne. Hver modul dekker ulike lag av kommunikasjonen mellom klient og database og er separert der det er knutepunkter i informasjonsflyten. Vi begynner arbeidet ved at alle gruppemedlemmene sammen diskuterer den generelle strukturen til appen. Dette gjør at alle får

en oversikt og eierskap over prosjektet.

I vårt første møtet definerer vi hva problemer appen skal kunne løse og vi begrenser appen til hva kjærnefunksjonalitet som skal til for å fylle kravene i oppgavebeskrivelsen. Vi bestemmer oss for hvilken teknoligi som da blir nødvendig.

Teknologivalg

For å holde på brukerdata trengs en database. Ettersom buisnissmodellen krever data med mange interne koblinger, velger vi å bruke MySQL som database. En relasjonsdatabase som MySQL kan normaliseres slik at data ikke lagres flere steder, noe som reduserer rissikoen for inkonsistens og sparer plass. MySQL skalerer bra med mange brukere og kan effektivt koble sammen tabeller med ulik data. MySQL sikrer også consistency ettersom commits er det siste utføres etter at alle endringer er gjort.

Vi velger å legge businesslogikken i en server som står for kommunikasjonen mellom databasen og klienten. Serveren er python flask[(4)]. Vi bestemte oss for å bruke python til backend programmeringen siden vi alle hadde jobbet med dette tidligere og var komfortable med å bruke det. Vi velger å bruke flask av samme grunn, dessuten er flask velegnet for relativt små prosjekter slik som vårt.
Ved å skrive serveren i python, som er svakt typet, får vi stor fleksibilitet i hva data som sendes og mottas. Hvis et grensesnitt endres, kan mottaker sjekke om inndata er av det gamle eller nye formatet og behandle dataen deretter. Ved å skrive bakoverkompatibel kode blir overgangen mellom endringer av grensesnitt mykere. Det negative med at python er svakt typet og tolket er at om et grensesnitt endres og den ene enden av informasjonsoverføringen ikke er klar, får vi ikke feilmeldinger før kjøretid.
Å velge et rammeverk som flask som server sparer både produksjonstid, og gir sikkerhet ettersom det er vel utprøvd, i forhold til om vi skrev en server fra bunn. Python flask er begynnervennlig og rask å sette opp slik at vi raskt får kommunikasjon med databasen og satt i gang med en prototype.
Python serveren har også ansvaret for å varsle brukeren om en viktig hendelse nærmer seg. Serveren skal sende brukeren en mail i forkant av hendelsen en tid bestemt av brukeren. Denne delen av applikasjonen kjører uavhengig av resten av flask serveren og er dermed implementert som en egen «thread». For å sende mail til brukeren velger vi å bruke et python rammeverk kalt MIME[(5)]. Hvert 10-ende minutt utføres sjekker og email sendes. Det var utfordrende å få denne servicen til å kommunisere med samme databasetilkobling som resten av applikasjonen, og etter mange forsøk ble det besluttet at det enkleste var at denne fikk en egen tilkobling mot databasen.

På klientsiden er applikasjonen en single page web applikasjon.
Fordelen med en webapplikasjon er bred støtte av mobile eneter og datamaskiner. Alt du trenger for å kjøre appen er en nettleser, noe de fleste mobiler allerede har innstallert, altså trengs ingen ekstra nedlastinger eller installasjoner. En nettside kan også lett deles via url, eller søkes på via søkemotorer. Andre fordeler er standariserte (og dermed forutsigbare for brukeren) brukergrensesnitt, automatiske oppdateringer av browsere og bra dokumentasjon. Dette sparer utviklingstid. Desverre krever webapplikasjoner kontinuerlig nettilgang for å fungere, men ettersom brukerdata lagres på nett, er nettilgang nødvendig uansett.
GUI bygges med html og styles med css. Javascript brukes til å kommunisere med server og har alle fordelene med svakt typede og tolkede skriftspråk. Vi bruker json for å kommunisere fra klienten til serveren. Javascripts objekter sendes som json tekst til serveren hvor teksten dekodes til pythons dictionaries. Ettersom visnig av kommende hendelser og oppgaver til brukeren er viktig, laget vi en

tidslinje i javascript fra bunnen av. Dette ga oss mer frihet (enn om vi brukte et rammeverk) til å utforme hvordan brukeren kan se kommende hendelser og oppgaver, fortelle applikasjonen når oppavene skal gjøres og sette opp nye hendelser. Vi ender altså opp med en «thick» applikasjon. Fordelen med at så mye som mulig skjer på klientsiden er kortere responstid, dette er mer brukervennelig. Ved å lage en single page app, vil kun de delene av html-documentet som er nødvendig å bytte ut, byttes ut. Informasjon kan lastes ned og opp i bakrunnen, noe som minimerer ventetid. Ved å lage vår egen SPA lærer vi mer om hvordan teknologien fungerer grunnleggende, i stede for å kun lære hvordan et rammeverk fungerer. Vi får også mer fleksibilitet og kunnskap over rammeverket som vi selv lager, det gjør det enklere å utvide det og tilpasse det etter produktseiers behov. På den negative siden krever dette mer jobb fra oss, og dessuten er standariserte rammeverk bedre testet og gir gjerne brukeren mer kjente grensesnitt. Det kan også være etablerte kommunikasjonsprotokoller mellom standariserte grensesnitt. Likevel velger vi å lage vårt eget, men vi bruker standariserte rammeverk i andre deler av applikasjonen. Mer om hvordan SPA påvirker html på frontend kan leses under overskriften «Frontend», underoverskriften «SPA». Mer om hvordan SPA gjøres i javascript kan leses under «Single page app med ajax»

I forhold til at applikasjonen er en single page app, brukes javascript til å endre html elementene dynamisk via ajax kall. For å få til dette er tilbakekompatibel og konsekvente protokoller mellom ajax funksjonene i javascript og html koden ekstremt viktig. Derfor dokumenterte vi disse protokollene slik at de som jobber med html lett kan finne syntaks til det de vil lage, og de som skriver dokumentasjonen blir mer ansvarlige for å ikke bryte tilbakekompatabiliteten. Et mål er også å gjøre grensesnittet så enkelt som mulig. Om grensesnittet begynner å bli unødvendig komplisert, oppdager de som skriver dokumentasjonen dette tidlig og kan gjøre noe med dette. Om grensesnittet er så komplisert at det er vanskelig å forklare det kortfattet i tekst, er det gjerne for avansert.

Ytre arkitektur

Vi lager en webapplikasjon. Brukere kan registreres, kalendere kan opprettes og oppgaver kan planlegges. Her er en oversikt over de ulike elementene.

User

Det første en bruker av applikasjonen blir bedt om er å registrere seg. En «user» vil da bli opprettet. En bruker registrer en user med email og et nickname. Email brukes når brukeren logger inn og til å resette passord om brukeren glemmer det. Når brukeren har registrert en user vil en «calendar» opprettes automatisk.

Calendar

En user vil alltid ha en calendar, men kan også opprette flere eller redigere dem som er. Calendar inneholder informasjon over hva oppgaver brukeren skal gjøre. Denne informasjonen er i form av «events» og «tasks». Begge disse elementene kan opprettes av en user og de er knyttet til en kalender.

Task

En task er en oppgave som brukeren har registrert på en kalender. En task består av et navn, beskrivelse, startdato og et intervall. Intervallet betyr at en task vil komme igjen og igjen for eksempel hver andre tirsdag hver måden for eksempel. Strukturen til et intervall er består av et hovedintervall og sub-intervaller. Et hovedintervall er enten årlig, månedlig, ukentlig eller daglig. Et hovedintervall har

også en startdato, som vil være dagen den opprettes, og en modulus. Hvis hovedintervallet er månedlig og modulus er 3 for eksempel, vil intervallet gjennta seg hver tredje måned. Et sub-intervall brukes til å spesifisere innenfor et hovedintervall når tasken skal gjøres. Hvis hovedintervallet er årlig, kan sub-intervallet være månedlig, ukentlig, daglig osv. Et subintervall kan også ha subintervall, så hvis hovedintervallet er årlig, og et sub-intervall er månedlig, kan et nytt sub-intervall for eksempel være den tredje dagen i måneden, eller den andre mandagen osv. Du kan også la være å spesifisere og bare la intervallet være årlig eller månedlig. Når en task skal utføres, blir den gjort om til en event. Dette gjøres i en «timeline». I tidslinjen kan alle tasks fremover og bakover i tiden sees. Tasks kan også sees i tabben «my tasks». Etter at tasken er plassert de aktiveres den for nå. Når tiden går inn i neste intervall vil tasken bli aktivert på nytt.
En task består av «todos». For å fullføre en task, må alle todoene bli fullført (gjort om til events). Det kan også være null todoer, i så fall vil tasken selv kunne fullføres.

Event

Event har et starttidspunkt, slutttidspunkt, navn og beskrivelse. Events kan opprettes fra kalendere eller fra tasks som gjøres til events. Events kan sees på listeform under tabben «my events» eller i tidslinjen.

Indre Arkitektur

Applikasjonen består av fire hovedmoduler.

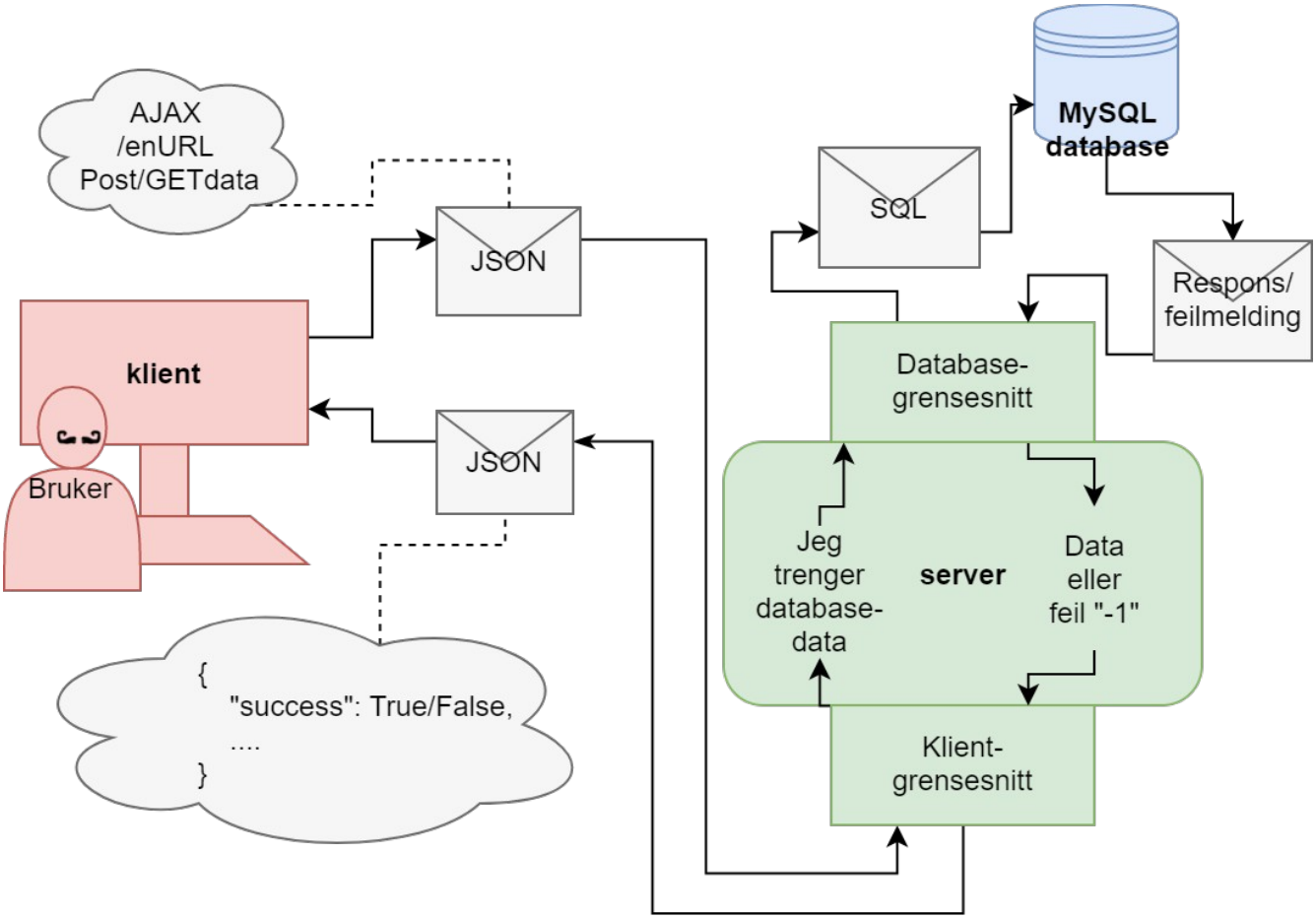
Innerste modul er databasen, her lagres data om brukere, kalendere, hendelser, oppgaver og koblinger dem i mellom. Det er lagt til restriksjoner i databasen som gjør at «DELETE»-statements ikke blir godtatt. Detter er for å hindre at data blir slettet ved uhell eller om uvedkommende får tilgang til databasen. Hvis elementer i databasen slettes, vil et flagg «Deleted» settes til 1. Dette gjør at slettet data kan gjenopprettes. Hvis, for eksempel, en hendelse blir slettet, slettes også hendelsen i alle kalendere den er med i. At ingenting slettes for godt er også bra for statistikken sin del. Vi beslutter at databasen ikke skal gjøre mer logikk enn det som er beskrevet over, ikke grunnet begrensninger i MySQL, men det er mer oversiktlig om buisnisslogikk blir tatt hånd om på serveren. Database-relaterte filer finnes i /db folderen.

Neste modul, databasegrensesnittet, står for kommunikasjon mellom server og database. Denne modulen sikrer at brukere ikke har tilgang til data på serveren som de ikke skal ha tilgang til ved at bruker-id brukes i sql-kallene som kreterie for å få ut data. Kun innloggede brukere kan se data tilhørende sin bruker. Denne modulen er den eneste som har noe direkte å gjøre med databasen, det betyr at om vi endrer database, vil dette være det eneste grensesnittet som må endres. Dette følger «single-responsebility prinsiple», et object oriented design prinsiple. De gamle filene relaterte til serverens databasegrensesnitt finnes i /application/old/ og har prefikset back_. Disse filene ble skrevet på ny og koden er nå i filen mf_database.py.

Serverens klientgrenesnitt mottar forespørsler fra klienten, utfører buisnisslogikk, og eventuellt viderefører informasjonsforespørselen til databasegrensesnittet. Data vil så sendes tilbake til klienten sammen med eventuelle feilmeldinger. De gamle filene som stod for kommunikasjonen finnes i /application/old/ og heter app.py og dataIO.py. Disse ble også skrevet om og de nye filene finnes i /application/ og heter mf_app.py og mf_dataIO.py.

«Frontend» står for kommunikasjon mellom brukeren og serveren. Inndata fra brukeren konverteres til

et format som kan tolkes av serverens klientgrensesnitt. Frontend kommuniserer med brukeren ved html og css. Html filene finnes i html og templates folderne under /application. Javascriptfilene og css er under /application/static.



Separation of concern

Ved å splitte applikasjonen inn i separerte lag som dekker hvert område i kommunikasjonen fra klient til database og tilbake, ender vi opp med et samlebånd av mooduler hvor hver modul kun trenger å tenke på grensesnittet mellom seg selv og de nærliggende modulene. Ettersom hver modul alltid konverterer dataen den mottar fra et grensesnitt før den sender det videre til et annet, vil format relaterte problemer oppdages tidlig. Hadde ikke dataen blitt konvertert og sjekket mellom hver modul, risikerer vi at en endring i datastrukturen i databasen, fører til at frontend, helt i andre enden, mottar data i feil format. Da er det vanskeligere å finne ut hvor feilen ligger, og vi risikerer at alle moduler må endres for å fikse problemet.

Samlebondstrukturen la også grunnlaget for arbeidsflyten av utviklingen. Modellen vi følger er Kanban.
Når vi begynner arbeidet på en ny funksjon til applikasjonen, begynner det med at klientmodulen definerer et nytt informasjonsbehov. Enten er det noe nytt som skal kunne lagres, for eksempel et nytt

«event», eller et behov for å hente informasjon fra serveren, for eksempel å kunne hente alle «events» innenfor en gitt tidsperiode. Dermed vil nye kommunikasjonsprotokoller bli laget mellom modulene. Når en ny funksjon legges til, går den gjennom flere steg.

Vi har en backlog av funksjonalitet som skal implementeres. Hver funksjon beveger seg gjennom flere steg i arbeidsflyten fra start til ferdig. Vi prøver å ikke ha flere enn en funksjon i hvert produksjonssteg. I det første steget blir den nye funksjonen planlagt. Dette steget blir gjort i ukentlige møter, der alle forklarer hva de trenger av hverandre.
I neste steg, implementerer hver modul sin seksjon av prosessen i forhold til sine grensesnitt. Enhetstesting blir gjort med dummy-data på inn- og ut-grensesnitt. Hvert gruppemedlem har hovedansvar for hvert sin modul og koder sin løsning i en egen branch i github.
Når hvert medlem er ferdige med sin del av implementasjonen av funksjonen går den over til neste steg. Alle møtes, og de ulike branchene blir merget sammen i en debelopment branch. Development branchen skal til en hver tid være noen lunde fungerende, så etter hvert mergingmøte skal mergekonflikter og misforståelser i grensesnittene mellom modulene løses. Når den nye funksjonen er ferdiggjort, og tilstrekkelig testet, regnes funksjonen som ferdig og development branchen merges inn i master branchen. Når en funksjon er ferdig i et steg er vi klare til å begynne på neste funksjon.

I begynnelsen møter vi for merging hver tirsdag. Vi innser etterhvert at en gang i uken er for lite ettersom problemer oppstår. Mer som dette i neste avsnitt.

Problemer oppstår

Et problem vi ganske raskt støtter på, er at mergingen tar for lang tid. Vi rekker ikke på de planlagte mergedagene å fikse konfliktene som oppstår. Dette fører til statig mindre kompatabilitet mellom de ulike delene av applikasjonen. Etter hvert som kompleksiteten øker, blir problemet statig forverret ved at også gamle funksjoner som tidligere fungerte, slutter å fungere på grunn av endrede grensesnitt. Etter hver som problemet forverres fører det til at vi utsetter merging, jobber mer separat og får en tedens til å miste fokus på produktet. Testing blir også vanskelig, for selv om vi gjør enhetstesting, er det vanskelig å teste hele stabelen fra klient til database og tilbake.
For å løse dette problemet, møtes vi 3 ganger i uken, i stedet for 1. Dette gjør at vi raskt får synkronisert modulene, og konflikter mellom grensesnitt løses. Når det gjelder problemet med at gamle funksjoner som før virket, slutter å fungere, skyldes det at når ny funksjonalitet blir lagt til, endrer vi grensesnitt på den ene enden, uten at grensesnittet på den andre enden er klar. For å løse problemet begyner vi med tilbakekompatibilitet. I stedet for å for eksempel sende lister med data mellom moduler, sender vi i stedet objekter eller dictionaries. Dette gjør at vi kan legge til eller fjerne elementer uten at grensesnittet slutter å fungerer. Det gjør også koden mer lesbar.

Eksempel på tilbakekompatibel navneendring:

```
return = {
    "success": False, # name changed from isSuccess to success
    "isSuccess": False
}
```

Under er et eksempel på mindre lesbar og ikke bakoverkompatibel kode ved bruk av lister:

```
var response = JSON.parse(responseText);
var start = response[0]
```

```
var end = response[1]
var name = response[2]
```

Her er et eksempel på mer lesbar, bakoverkompatibel kode ved bruk av dictionaries objekter:

```
var response = JSON.parse(responseText);
var start = response.start
var end = response.end
var name = response.name
```

Et annet problem som forsinker mergingen av de ulike branchene, er runtime-errors. Spesiellt om disse feilene oppstår i moduler en selv ikke jobber med direkte. Feilmeldingene som oppstår er lite spesifikke og er ofte langt fra den egentlige årsaken til feilen. Ved å lage egne feilmeldinger og tidligere sjekker for feil, går debuggingen raskere. I stedet for å få en runtime-errors og en lite lesbar feilmelding, implementerer vi i stedet bedre sjekker innad i modulene som sjekker om inndata er i det forventede formatet. Hvis formatet er feil printes eller sendes en mer informativ feilmelding.

Eksempel på en lite informativ feilmelding:

```
range = Tool.getNextInterval(new Date("04 jan 2017 00:00"), true, {weekNrInYear: 0,
   dayNrInWeek: 1});
> mf_Tool.js:99 Uncaught TypeError: Cannot read property 'getTime' of undefined
    ...
```

Eksemepel på mer informativ feilmelding:

```
range = Tool.getNextInterval(new Date("04 jan 2017 00:00"), true, {weekNrInYear: 0,
   dayNrInWeek: 1});
> There must be exactly one interval as input.
Error
    ...
```

I tillegg til å forbedre feilmeldinger, ble vi enige om hvilke verdier som blir returnert dersom feil oppstår. Feil i SQLen som sendes til databasen som fører til exceptions, blir catch-et på serveren og feilmeldingen blir printet i consollet. Når funksjoner innad i serveren feiler, returnerers «-1». Når feiler oppstår på serveren, vil hva som gikk galt logges i et «notifications» system. Feilloggen, sammen med attributtet «success», som enten er «True» eller «False», pakkes inn som json og sendes til klientgrensesnittet.

Vi tror hvis vi skulle gjort noe annerledes med dette prosjektet, er det å bruke mer tid i starten av prosjektet på å etablere en skikkelig standard for hvordan de ulike lagene skal kommunisere seg i mellom. En idée for å sikre at standarden blir holdt, er å gjøre mellom alle grensesnittene, slik vi gjorde mellom html og ajax delen i frontend, å skrive ned dokumentasjon på grensesnittet. Da kan det ikke misforstås det nøyaktige formatet på data inn og ut av grensesnittene.

Tidslinjen

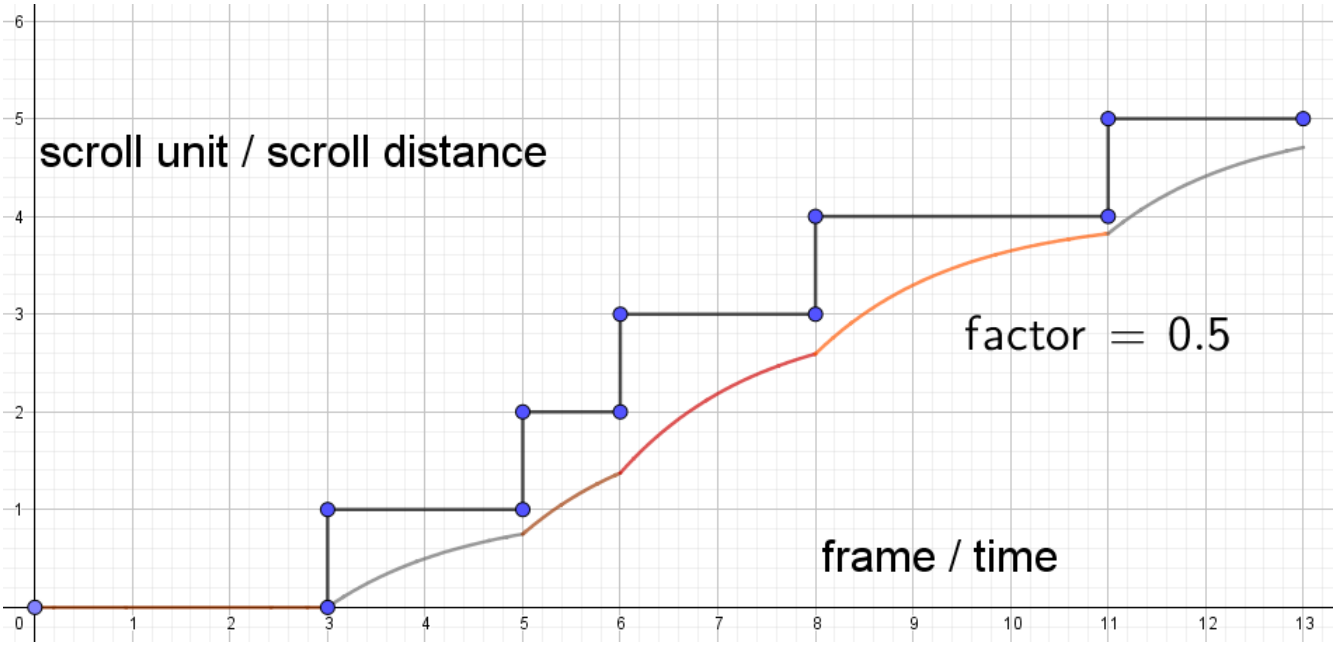
For å gi en oversikt over kommende, passerende og tidligere events og tasks brukes en tidslinje. Tidslinjen kjøres i javascript og koden finnes i filen /application/static/js/mf_timeline.js Tidslinjen kan tenkes på som et hjul, eller en slags tredemølle der brukeren kan bruke fingeren til å skyve tidslinjen

forover eller bakover for å se forover eller bakover i tiden. Dette ligner måten scrolling skjer til normalt i en nettleser på mobil, så det vil være intuitivt for brukeren. På tidslinjen vil kommende hendelser vises som fargede bokser med tekst. Størrelsen på boksene sier noe om når de begynner eller slutter. Brukeren kan også zoome inn og ut fra tidslinjen for å øke eller minske perspektivet.
I den ferdige versjonen av tidslinjen (bilde kan ses på side TODO) er det tidsenheter på bunnen. Her kan brukeren se hvilket år, måned, uke eller dag det er. I midt-delen kan brukeren se kommende og tidligere events og på toppen kan brukeren se kommende og tidligere tasks. Det er også to ulike moduser som kan skiftes mellom ved å trykke på knappen «change to timeline view». Her kan brukeren skifte til en modus der de neste taskene kan gjøres om til events som plasseres i tidslinjen.

Et alternativ til å zoome inn og ut kontinuerlig, var å bruke knapper for å skifte mellom daglig, ukentlig, månedlig eller årlig perspektiv. Vi gikk for continuerlig zooming ettersom det passer bedre med temaet årshjul og det er mer fleksibelt og enklere. I stedet for at brukeren må lære seg de ulike knappene og hvilket zoom invå de gir, lærer brukeren seg hvordan zoomen fungerer en gang, og kan dermed navigere seg helt fra minutt- til år-nivå. «Keep it simple stupid» . Knapper hadde også tatt mer plass. Det er altså ganske viktig at kontrollene for å zoome er så intuitive som mulig. Vi kommer tilbake til det under overskriften «brukertesting».

Til høyre er et eksempel på hvordan tidslinjen så ut til å begynne med. Forover i tiden er nedover. De horisontale linjene markerer starten på nye dager, uker, måneder eller år. Det større tidsinterval som starter, det tykkere og bredere er linjen. For å bevege seg forover i tiden, stryker brukeren fingeren vertikalt over skjermen. Dette fører til at linjene og teksten beveger seg med fingeren og gir en illusjon av at tidslinjen flytter på seg. Bevegelsen oppnås ved bruk av animasjon. 30 ganger i sekundet vil linjene og teksten flyttes en liten avstand i en retning. Html «Canvas» elementet brukes for grafikk. Ved å lage vår egen tidslinje, gir det oss stor kontroll og fleksibilitet til å legge til ny funksjonalitet, og gjøre appen så tilpasset produktseiers behov som mulig.

På pc brukes musens scroll for å bevege seg forover og bakover i tiden. Et problem som oppstod med scroll, er at for at sensitiviteten skulle være høy nok, måtte en scrollenhet flyttet tidslinjen ganske langt frem eller tilbake i tiden. Sensitiviteten kunne ikke settes lavere ettersom det gjorde at testpersonene klagde på at tidlinjen var for treg. Høy sensitivitet førte til for store hopp fremover eller tilbake i tiden, noe som gjorde det vanskelig å se hvilken retning tiden beveget seg i mens vi scrollet. En løsning på dette problemet er å i stedet for at scrollen direkte endret på hvilket tidspunkt som var i senter av skjermen, ble heller en målposisjonen endret. Den ordentlige posisjonen (tidspunktet) som vises på skjermen vil så følge målposisjonen eksponensiellt.



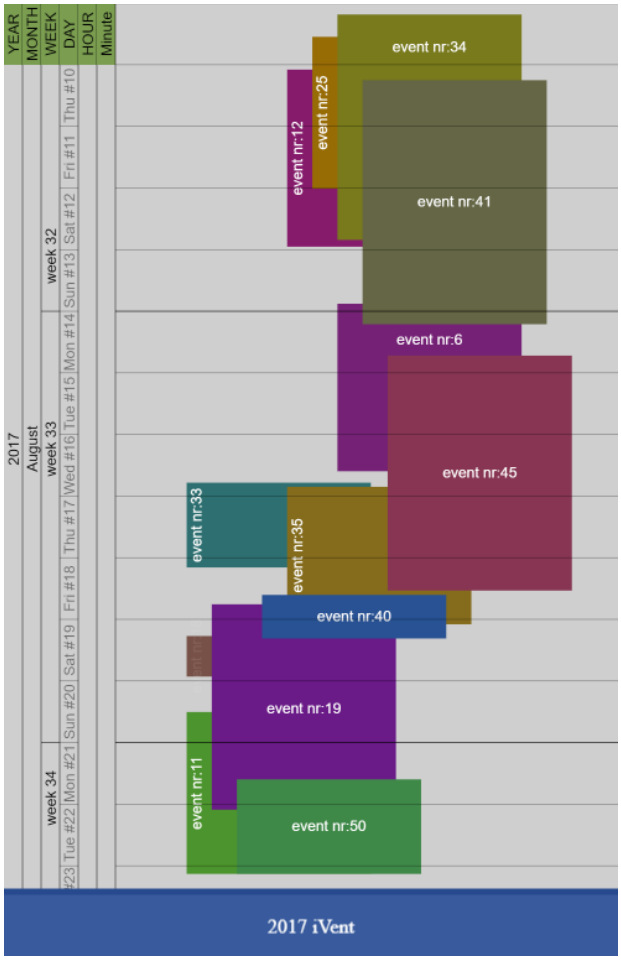
Y-aksen viser hvor langt tidslinjen beveger seg, mens x-aksen viser hvordan animasjonen går. Den kantede linja viser hvordan tidslinjen hadde flyttet seg om den fulgte scrollen. Den mykere kurven viser hvordan tidslinjen flytter seg over tid når posisjonen går eksponensiellt mot en målposisjon i stedet. Hvis «factor» er nærmere 1, vil den myke linjen ligne mer på den kantede, hvis «factor» er nærmere 0, vil den bli mykere. Høy verdi gir en mer responsiv bevegelse mens en lav verdi er mer behalelig å se på. Vi velger 0.1, det er et bra kompromiss.

Koordinater og enheter

Med scrolling har vi så vidt vært innom koordinater. Animasjonen av tidslinjen foregår i et javascript «interval».
Hvert 1/30 sekund vil funksjonen loop kjøre og end variable kalt «tick» vil øke. Tick er enheten som brukes for å holde styr på tiden slik som brukeren opplever den. 1 tick = 1/30 sekund.
Tidslinjen har en «posisjon» som sier noe om hvilket tidspunkt som for øyeblikket kan sees på midten av skjermen. Enheten til position er gitt i millisekunder fra 1. januar 1970. Dette gir en linær enhet som er lettere å kalkulere med enn å bruke datoer.
«zoom» er en annen variabel som bestemmer hvor stort tidsintervall som er synlig. Zoom er også gitt i millisekunder fra 1. januar 1970.
Tidslinjen visualiseres i et html canvas element. En posisjon i canvas er gitt i pixler. Funksjonene canvasCoordsToTime(coords) og timeToCanvasCoords(time) vil convertere mellom pixler på skjermen til millisekunder og tilbake. Disse funksjonene finnes i /application/static/js/mf_timeline.js.

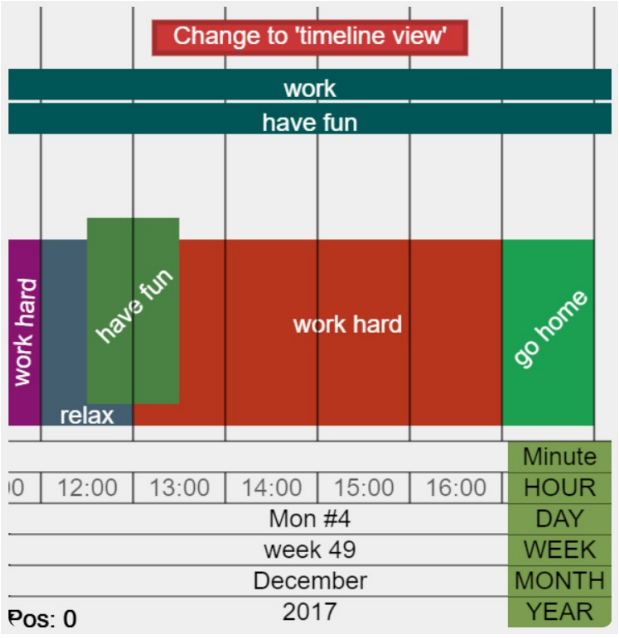
Brukertesting

Hvis brukergrensesnittet er vanskelig å forstå mister vi raskt oppmerksomheten til brukeren. Det er viktig at måten brukeren kontrollerer applikasjonen på og måten applikasjonen responderer på, er så intuitiv som mulig. Brukertesting er essensiellt for å sjekke om grensesnittet faktsik er så intuitivt som vi tror det er, eller om vi som utvikler det er så vandt med det at vi tror det er enkelt.



Underveis i utviklingen har vi fått gode tilbakemeldinger som gir en indikasjon på hvordan «80%» av brukerne vil bruke denne applikasjonen. Vi har prøvd å etterkomme disse tilbakemeldingene så godt som mulig. Dette angår utforming av skjema, standard valg og applikasjonsflyt (brukerreisen) (feks hvor en lander når en har laget et nytt event) og utforming at tidslinjen.

Når vi tester ut applikasjonen følger vi nøye med på hva det første forsøket testeren gjør på å bruke en funksjon. Et eksempel på dette er scrolling og zooming av tidslinjen.
For å zoome og scrolle tidslinjen begynte vi med å la musen, eller en finger om du bruker mobil, scrolle og zoome i tiden. Problemet med dette er at å dra skjermen med fingeren kolliderer med den vanlige scrollingen på en nettleser på mobil. For å løse dette problemet, lot vi to fingre scrolle og zoome på tidslinjen, mens en finger scroller nettsiden som vanlig. Når vi tester denne løsningen observerer vi at de fleste som bruker denne løsningen blir forvirret. De forsøker først å bruke en finger til å scrolle, ettersom de allerede er vandt med det fra nettleseren. Når de forsøker å zoome drar de fingeren mot seg, nesten som om de skulle dradd et papir som ligger på et bord mot dem for å se nærmere på det. Ettersom det var dette brukeren forsøkte, implementerte vi denne løsningen i stedet. Når det gjelder konflikten mellom å scrolle med en finger i nettleseren, eller å scrolle i tidslinjen, løste vi dette med at tidslinjen kan aktiveres ved et klikk. Du kan trykke hvor som helst på skjermen, noe som brukeren forsøker tidlig.



Oppe til høyre er et eksempel på en tidlig prototype. Forover i tiden i tidslinjen er nedover, og bakover i tiden er oppover. Til venstre i figuren kan vi se

hvilket tidsrom vi befinner oss i. Til høyre i figuren kan vi se dummy events. Toppen og bunnen av eventet viser når eventet starter og ender. Eventende blir også spredt utover slik at starttidspunkt, slutttidspunkt og navnet skal være synlig så lenge det er plass.

En annen tilbakemelding gikk ut på at designet var vanskelig å forstå. Det er ikke intuitivt at tiden går nedover og mye av teksten må du legge hode på skakke for å lese. På grunn av dette velger vi å snu tidslinjen horisontalt. Dette er mer intuitivt ettersom forover i tiden blir til høyre, samme retningen vi leser. Navnene kan da også plasseres horisontalt. Bildet nederst på forrige side viser den nye horisontale versjonen av tidslinjen.

Etter hvert som vi la til ny funksjonalitet ble det, litt etter litt, vanskeligere å vedlikeholde brukergrensesnittet. Ny funksjonalitet ble lagt til og grensesnittet ble et lappeteppe av knapper og instillinger. Vi gjobbet med å gjøre det hele mer intuitivt ved å iterere mellom brukertesting og forbedring av grensesnittet, men ble ikke helt ferdige. Slik som produktet endte opp må vi fortsatt i tidslinjen skifte mellom et timeline view og et task view i stedet for å bare klikke på tasks i tidslinjen for å så plassere dem direkte, men det ble i det minste mye bedre enn slik det var før vi fikk tilbakemeldinger.

Animasjon

Animasjonen i tidslinjen skjer som nevnt tidligere i et intervall, der hele tidslinjen tegnes på nytt 30 ganger i sekundet. For å spare prosessorkraft, blir dette satt på pause dersom bildet ikke endres, det vil si, tidslinjen flytter seg hverken forover eller bakover i tid, zoomer ikke og ingen knapper trykkes på.

Når brukeren bruker musen, eller fingeren på mobil, til å flytte tidslinjen er det viktig at tidslinjen følger bevegelsen til fingeren både translatorisk og i forhold til zoom slik at illusjonen av at brukeren holder tidlinjen fast og flytter den, ikke brytes. For å få til dette må posisjonen som er i midten av skjermen flyttes litt når det zoomes.

```
if(this.mouseData.isDown){
    deltaZoom += (this.mouseData.pos.y - this.mouseData.pos0.y) / this.canvas.height *
        this.zoom * zoomSensitivity;
    deltaScroll -= (this.mouseData.pos.x - this.mouseData.pos0.x) / this.canvas.width *
        this.zoom * scrollSensitivity + deltaZoom * (this.mouseData.pos.x -
        this.canvas.width * 0.5) / this.canvas.width;
}
```

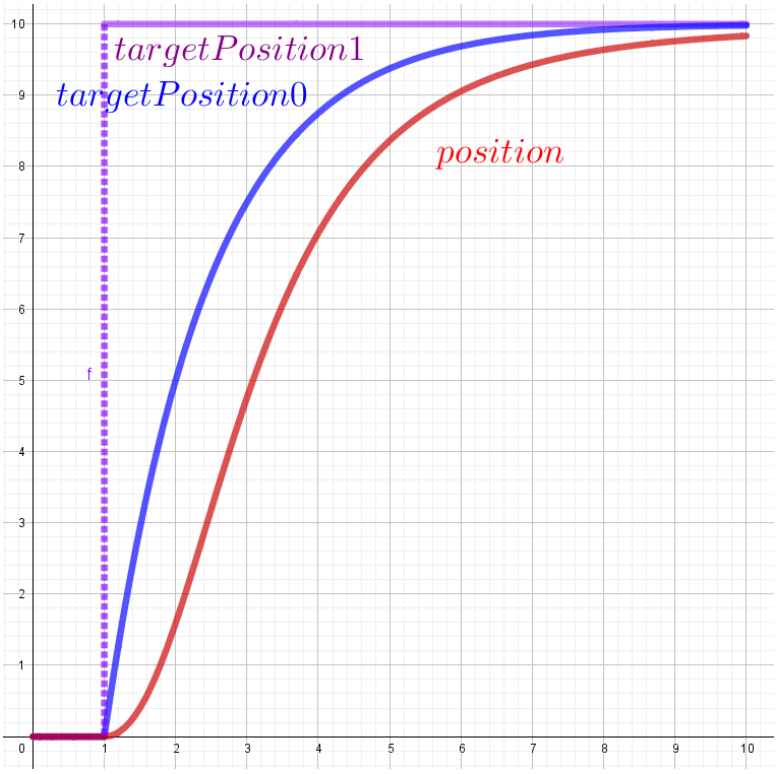
Koden er fra /applicationstatic/js/mf_timeline_loop.js Timeline.timelineControls(). «this.mouseData.pos» er vektorposisjon som beskriver hvor i canvas elementet musen befinner seg. «this.mouseData.pos0» beskriver hvor musen var for 1/30 sekund, eller ett tick, siden. «deltaZoom» er hvor mye zoom-nivået skal endres hvert tick (1/30 sekund). «deltaScroll» er hvor mye posisjonen som vi ser på skal endres per tick. DeltaScroll vil være avhengig av mengen det zoomes, slik at om brukeren holder fast på ene enden av tidslinjen og zoomer, vil sentrum bevege seg slik at posisjonen til musen/fingeren i tidskoordinater vil tilsvare pikselkoordinater. Dette legges til for å fikse problemet som oppstår når testbrukere av tidslinjen forsøker å zoome til for eksempel et event på ene enden av skjermen ved å flytte musen dit og dra eventet mot seg. Eventet vil da forsvinne ut av syne ettersom tidslinjen zoomer mot sentrum.

Flere steder i tidslinjen vil grafiske elementer flytte seg over tid. Dette skjer ved at elementer blir

tegnet, for så å bli vist til brukeren i 1/30 sekund, for deretter å bli visket ut og tegnet på nytt noen piksler til siden.

```
// clear
this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
...
// calculate target positions
task.targetPosition1 = ...
// move towards target positions
var movementSpeed = 0.1
task.targetPosition0 = Vec.lerp(task.targetPosition0, task.targetPosition1,
      movementSpeed);
task.position = Vec.lerp(task.position, task.targetPosition0, movementSpeed);
// render
this.drawRectOutline(task.position.x, ..., color)
```

Denne koden kjøres i /application/static/js/mf_timeline_render.js. I «calculate target positions» delen vil, for eksempel, en task i «taskview», regne ut sin nye målposisjon. I «move towards target positions» delen vil posisjonen til task-en endres slik at den er nærmere målposisjonen. Vec.lerp(a, b, factor) vil gi en vektor som er mellom vektorposisjonene «a» og «b», hvor «factor» bestemmer om vektoren skal være nermest a (factor = 0), eller b (factor = 1).
Det finnes to posisjoner «targetPosition0» og «targetPosition1». Disse to målposisjonene og «position», gjør at bevegelsen til elementene blir myke.



Grafet til høyre viser et eksempel på hvordan denne, og andre animasjoner utfolder seg. Først vil targetPosition1 endres til den nye posisjonen der det grafiske elementet skal flyttes. Dette skjer umiddelbart. TargetPosition0 vil så bevege seg eksponensiellt mot targetPosition1. Position vil så bevege seg eksponensiellt etter targetPosition0. Ettersom avstanden mellom position og targetPosition0 er 0 i begynnelsen, får animasjonen en myk begynnelse, og ettersom en eksponensiell funksjon med faktor mellom 1 og 0 endrer seg ganske langsomt etterhvert, vil animasjonen få en langsom avslutning. Selv om i kodeeksempelet på forrige side brukte task som eksempel, er denne måten å animere generelt for de fleste animasjoner som skjer i applikasjonen.

Frontend

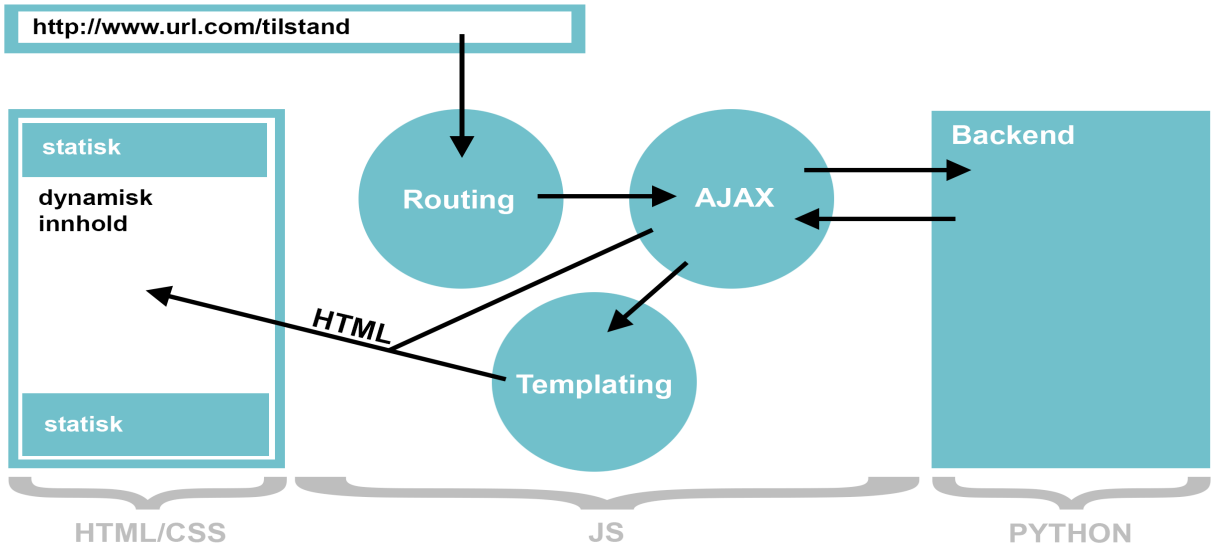
Introduksjon

Det at vi velger å lage en "Single Page App" (SPA) har store konsekvenser for frontend utviklingen av

applikasjonen. I motsetning til en tradisjonell webapplikasjon hvor HTML blir generert på serversiden og serveres til frontend som statiske sider der en URL tilsvarer en gitt HTML side, blir det i en SPA hele tiden generert innhold på siden og den tradisjonelle måten å tenke at en URL gir et bestemt innhold holder ikke lenger mål. I en SPA må en tenke at webapplikasjonen har en tilstand og ut ifra denne tilstanden så må den riktige informasjonen hentes og plasseres på siden hvor det passer. Samtidig er det viktig å ta hensyn til den tradisjonelle måten å tenke på og gi en illusjon av at bestemte URLer gir et bestemt innhold.

SPA

Webapplikasjonen vi lager er en tjeneste hvor i utgangspunktet all data ligger bak innlogging og er av privat karakter. Derfor trenger vi ikke å ta hensyn til at sidene skal indekseres av søkemotorer, noe en SPA i utgangspunktet ikke er egnet for. I tillegg vil vi konstruere en applikasjon med mulighet for å arbeide «offline» og dette vil en SPA ha støtte for gjennom å laste ned og «buffre» data fra serveren samt «buffre» data for å senere sende oppdateringer til serveren. Dette er funksjonalitet vi ikke fikk tid til å gjennomføre i prosjektet, men som enkelt kan videreutvikles på topp av den eksisterende applikasjonen.
En SPA gir i tillegg større muligheter for å holde data oppdatert på frontend ved å hele tiden hente ny data fra serveren uten at brukeren trenger å laste inn siden på nytt.



Arkitekturen

For å levere en SPA trenger vi endel forskjellige moduler for å håndtere både det å generere innhold direkte på siden, men også for å generere en tilstand ut i fra en URL. En av de mest grunleggende modulene vi må få på plass er en templater for html. Vi tenkte først å utvikle dette selv og hadde en veldig enkel modul på plass, men behovet for ekstra funksjonalitet som å kunne kjøre løkker og enkle betingelser, gjorde at vi i stedet gikk for å bruke en 3djeparts modul i stedet[(1)].

Den neste modulen vi trenger er en routing modul for å håndtere tilstander utifra URL. Her går vi direkte for en 3djepartsmodul med full funksjonalitet for dette. Etter noe prøving og feiling av

forskjellige moduler endte vi opp med denne[(2)].

Vi bruker også er tredjepart modul for å velge datoer og tid[(3)].

Resten av modulene for å kommunisere med backend og plassering av innhold i applikasjonen har vi utviklet selv. Dette gir oss mer frihet og kontroll til å gjøre akkuratt det vi ønsker. Andre rammeverk har ofte mye mer funksjonalitet innebygd, men krever også en gitt struktur på applikasjonen. Dette kan kanskje spare oss for en del tid, men gjør også at vi lærer mindre om hvorfor og hvordan en SPA er bygd opp og vi lærer i stedet hvordan et bestemt rammeverk fungerer. Det gir oss også mer fleksibilitet til å lage akkurat det produkteier spør etter. I fremtidige prosjekter vil vi kanskje bruke flere rammeverk der vi av erfaring ser at dette er tidsbesparende.

Modulen vi lager for å kommunisere med backend bygger vi først med veldig enkel funksjonalitet, den henter html og data fra backend, kjører dette evt gjennom en templater og puttet det videre inn i applikasjonen der forespørselen kom fra. Til å begynne med lager vi "hooks" i HTML strukturen som kaller på denne modulen ved hjelp av data attributter på tagger. For eksempel:

```
<div data-fill="/getTMPL?tmpl=calendar_list&data=calendar_list"></div>
```

Denne attributten fyller elementet med data fra backend (en liste med kalendere) og kjører dette gjennom templateren med calendar_list malen. Dette er grei funksjonalitet å bruke til å konstruere mer modulære maler, men vi trenger også å kunne "kaste" disse "hooksene" på elementer ved hjelp av feks klikk på knapper. Derfor utvider vi funksjonaliteten til å kunne gjøre dette ved å introdusere et ekstra attributt:

```
<button data-target="someId" data-fill="/getTMPL?tmpl=calendar_list&data=calendar_list">En knapp</button>
```

Slik kan vi nå endre innhold i elementer på siden dynamisk. Etterhvert dukker det opp flere behov og vi utvider denne modulen med funksjonalitet for dette. Resultatet er en modul som gjør akkuratt det den skal gjøre, den henter data fra backend og putter det inn på siden der det trengs utifra hendelsen som gjør kallet. For eksempel click, load, change. Modulen er også konstruert slik at den klarer å håndtere flere lag av innhold. Altså den klarer å identifisere "hooks" som finnes inni et hierarki av maler. Dette er nødvendig for å kunne gjøre malene så modulære som mulig. Mer om dette står under overskriften «Single page app med ajax».

Alle «forms» som sendes til backend får en respons. På klientsiden håndteres responsen i en "callback" funksjon knyttet til skjemaet som sendte requesten. Denne håndteringen skjer i filen /application/static/js/nm_callbacks.js. En slik håndtering kan være å sjekke om formet ble akseptert av serveren, sjekke hva slags notifikasjoner som følger med i responsen og erstatte deler av html-en via nye ajax kall.

Teknologi

For frontenden har vi holdt oss til gjeldende standarder innen HTML5, CSS og JS. Ved å bruke den semantiske standarden i HTML5 gjør vi applikasjonen vår bedre tilgjengelig for brukere med funksjonsnedsettelse (universell utforming) i tillegg til å bruke fargevalg som tilfredsstiller kravene for kontrast for denne gruppen.

I tillegg har vi utviklet applikasjonen med hensyn til mobilbruk og innholdet er responsivt slik at det tilpasser seg forskjellige skjermflater.

Single page app med ajax

For at html delen i SPA-en vår skal fungere trengs det javascript som leser html koden og tolker attributter og andre koder og gjør den nødvendige templatingen og spørringene mot serveren. Frontend er altså delt i to deler der den ene delen består av html og den andre av javascript templating og ajax kall. Templatingen er beskrevet under overskriften «Frontend» så det gjenstår bare å forklare hvordan ajax kallene fungerer, men først en mer detaljert oversikt over syntaksen mellom html- og javascript-delen.

For å fylle et element med html.

```
<div data-load="/getHTML?html=someContent"></div>
```

For å få en knapp til å fylle et html element.

```
<button data-target="toBeFilled" data-fill="/getHTML?html=someContent">Button that will fill the h1</button>
<h1 id="toBeFilled">
    Some h1 to be filled
</h1>
```

For å få en knapp til å submitte et form.

```
<form id="someForm" action="/handleLogin">
    <input type="text" name="email" id="email"><br>
    <input type="text" name="password" id="password"></br>
</form>
<button data-formid="someForm" data-callback="callBackExample">Hello there</button>
```

Det siste eksempelet krever litt mer forklaring. Knappen på bunnen har attributtet data-formid. Attributtet har id-en til et form som igjen har en action. Action på formet sier hvilken adresse formet blir submitted til. Når serveren responderer, blir funksjonen i data-callback kalt, i dette eksempelet heter funksjonen «callBackExample». Her er et eksempel på hvordan slik en funksjon kan se ut:

```
function callBackExample(response){
    console.log("Success: " + response.success);
}
```

«response» er et object som sendes fra serveren som json. Det konverteres til et javascript object før det sendes til callback funksjonen. Disse funksjonene finnes i /application/static/js/nm_callbacks.js.

En full oversikt over hele syntaksen finnes i /application/html/htmlAjaxInstructions.txt. Dette dokumentet definerer grensesnittet mellom html delen og javascript delen. Koden for SPA finnes i /application/static/js/mf_ajax.js. Måten mf_ajax fungerer på er at etter at indekssiden er lastet inn, vil den søke dokumentet for data-xxx attributter hvor xxx står for load, fill, target, replace, before, after... etc. Når ny html lastes inn, vil den nye html snutten søkes gjennom for nye attributter, eventlisteners legges til knapper og script med «data-run» attributtet vil bli kjørt.

Testing

Når nye funksjoner lages, trengs det gjerne endringer i ulike moduler. Vi kan ikke teste hele stacken før alle modulene er ferdige. For å teste at koden vi skriver fungerer, må dermed hver modul gjøre enhetstesting der dummydata sendes inn og ut av grensesnittene. På python serveren ble filer som test_event.py og test_user.py brukt for enhetstesting av både server og database. På frontend ble mf_tester.js brukt til å teste at både frontend virket, men også hele stacken etter at de nye funksjonene er ferdig implementerte. Når nye funksjoner er laget er testing nyttig for å sjekke at tidligere funksjoner ikke er blitt ødelagte.

Frontend testes som sagt via scriptet mf_tester. Når siden lastes inn, vil mf_tester legge inn eventlisteners på alle knapper, tekstfelt og andre elementer som kan endres. Når ting på siden endres vil dette lagres. Hvis ctrl + shift + c trykkes, vil opptaket lagres i clipboardet som en tekststreng. Opptaket kan så pastes inn i filen /application/static/js/mf_testerData.js og lagres. Neste gang siden lastes inn på samme plass som opptaket startet vil kommandoen ctrl + shift + s starte en animasjon hvor kommandoene lagret i mf_testerData vil spilles tilbake. Mer om hvordan denne testingen brukes finnes i /application/static/js/mf_testHandelerInstructions.txt. Når animasjonen spilles av, vil en animert cursoren lete etter elementer basert på id-er eller plasseringen til elementet i hirearkiet av html-elementer. Animering av cursoren gjør det enklere å se hva, hvor og når ting går galt. Hvis deler av html strukturen endres vil testen fortsatt fungere, så lenge ikke html som direkte påvirker kommandoene som spilles tilbake endres. Grunnen til at vi bruker et eget rammeverk for testing er at vi har mange egne elementer, som for eksempel tidslinjen, som krever egen logikk for å fungere i testeren.

Oppsummering

Vi gjorde mye rett i begynnelsen når det gjalt å bruke mye tid på å planlegge buisnisslogikken sammen. Det førte til at vi fikk en generell forståelse av applikasjonens grunnstruktur. Vi splittet deretter opp applikasjonen i ulike lag der hver av oss hadde hovedansvar for hver vår del, dette førte til ansvarsfølelse og frihet til å løse problemer selvstendig.

Et problem som oppstod er at etterhvert som prosjektet progresserte, mistet vi denne oversikten ettersom hver av oss jobbet nesten eksklusivt med vår egen modul og vi mistet fokuset på selve sluttproduktet. Dette gjorde at selv om vi definerte grensesnitt mellom modulene til å begynne med, endret disse seg uten at vi snakker nok sammen. Problemet kunne blitt redusert ved at vi hjalp mer til på hverandres kode og fikk identifisert flaskehalser tidligere. Ved å sette seg inn i andres kode ville vi også fått en bedre forståelse av hva de andre driver med, noe som reduserer sjangsen for at vi ødelegger noe for dem ved å endre noe et annet sted.
De fleste problemene oppstod i grensesnittet mellom frontend og serverens klientgrensesnitt, og mellom serverens klientgrensesnitt og serverens databasegrensesnitt. Hadde vi fra begynnelsen logget konfliktene og hvor de oppstod, ville vi for eksempel i en pareto scatter graf eller lignende identifisert flaskehalsen tidligere, men vi gjorde ikke dette. Vi kunne også redusert problemet ved å innføre strengere tilbakekompatabilitet fra begynnelsen og dokumentering av grensesnittene. Når vi begynte med dette, hjalp dette mye, men det vi gjorde som definitivt var til størst hjelp var da vi begynte å møtes minst 3 ganger i uken i stedet for bare ett fast møte. Problemene førte til at vi mistet mye tid, men vi lærte mye fra disse erfaringene. Alt tar mer tid enn det vi tror det vil ta, og sammarbeid er en kunst.
Desverre ble vi ikke ble ferdige med alle funksjonene vi hadde planlagt. Brukergrensesnittet kunne blitt

pusset mer på og det mangler noen funksjoner, for eksempel, å kunne gjenopprette slettede kalendere eller deling av filer, men vi ble ferdige med kjernefunksjonaliteten.

Kilder

(1):
Absurd TemplateEngine, https://github.com/krasimir/absurd/blob/master/lib/processors/html/helpers/TemplateEngine.js
(2):
Navigo, https://github.com/krasimir/navigo
(3):
Flatpickr, https://chmln.github.io/flatpickr/
(4):
Flask http://flask.pocoo.org/
(5):
MIME: https://docs.python.org/2/library/email.mime.html, https://tools.ietf.org/html/rfc2046