NTNU
Det skapende universitet

Bruk av Event Sourcing for logging og visualisering av bruk av nettressurser

Andreas Haugen Pedersen

Master i datateknologi
Innlevert: juli 2015
Hovedveileder: Hallvard Trætteberg, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

Det skal sees på et system for å overvåke og registrere bruk av nettressurser i forbindelse med faget TDT4100. På nettsidene til faget ligger det flere forskjellige tester man kan kjøre i Eclipse. Ønsket er å kunne registrere data om kjøring av testene for å vise bruken av disse nettressursene. Ved å visualisere dataene kan man synliggjøre bruken av nettressursene slik at andre brukere kan få økt motivasjon.

Forskjellige visualiseringer av bruken skal kunne vises, som for eksempel hvor mange ganger en test er kjørt eller når testen sist ble kjørt. Det må vurderes hvilke visualiseringer som er hensiktsmessige å bruke, og hvilke data som er nødvendig å registrere for å få til visualiseringen. Her ønskes det å sees på bruk av Event Sourcing i systemet. Event Sourcing bygger på registrering og bruk av hendelser for å finne tilstandene til et system. Det må også vurderes hvordan lagring av dataene som registreres skal skje, og forskjellige løsningsalternativ må diskuteres. Det kan være en fordel å se på eksisterende systemer med liknende funksjonalitet og muligheten for å bruke disse i en løsningsskisse. En evaluering av løsningsskissen bør gjennomføres med studenter i faget, og dette vil innebære en implementasjon av systemet.

Sammendrag

I faget *TDT4100 Objektorientert Programmering* brukes det tester i øvingsopplegget for å se om studentene har fullført en øving. Data fra disse testene er registrert og lagret i et system som en annen student, Stein Kjetil Sørhus, har laget i sin masteroppgave. I dette systemet er det også lagret all tekst og kode som studenter i faget har skrevet, sammen med feil som er funnet i koden. Det er også laget en visning av dataene i systemet, som viser resultat av tester og feil i kode over tid. I denne oppgaven har vi sett på systemet til Stein opp mot Event Sourcing for å se hvordan systemet kan bruke Event Sourcing. Det ble gjort et studie av Event Sourcing som har lagt grunnlaget for beskrivelsen av systemet hans. Vi har ut i fra studiet og beskrivelsen vurdert noen alternative løsninger som kan gjøres i systemet. I tillegg har vi sett på hvor fleksibelt systemet er når det kommer til å lage nye visninger.

Fra studiet av Event Sourcing fant vi at systemet er likt et system med Event Sourcing og CQRS. I systemet er dataene først lagret som rådata i Git repositorier, og deretter i en grafdatabase som det lages visning av dataene fra. Vi så at dette var den beste måten for å lage visning, ved å lagre dataene først i grafdatabasen og lage visning derfra, i stedet for å lage en visning rett fra rådataene i Git. Dataene i grafdatabasen er hentet ut og analysert fra rådataene i Git. Vi så at i stedet for å hente ut dataene fra Git for så å analysere de, var det mulig å la systemet sende de direkte videre til analyse samtidig som de lagres i Git. Da slipper vi steget med å hente dataene ut fra Git. For å lage visninger av dataene i grafdatabasen trengs å lages spørringer som henter ut dataene. Vi så at det var fleksibelt å lage nye spørringer, og dermed også visninger. Dette ble illustrert i kode som ble implementert i Java.

Abstract

In the course *TDT4100 Object-Oriented Programming*, tests are used in the exercises to see whether a student has completed an exercise or not. Data from these tests are registered and stored in a system made by another student, Stein Kjetil Sørhus, in his masters degree. In that system all text and code, which students of the course have produced, are also saved along with errors found in the code. There has also been made a view of the data in the system, which show test results and errors in code over time. In this report, we have reviewed Steins system against Event Sourcing to see how the system can make use of Event Sourcing. There has been conducted a study of Event Sourcing which has laid a foundation for the review of his system. From the study of and the review, we have reviewed some alternative solutions for the system. In addition, we have looked at how flexible the system is when it comes to make additional views.

From the study of Event Sourcing, we found that the system has the likes of a system with Event Sourcing and CQRS. In the system, the data is stored first as raw data in Git repositories, and then they are stored in a graph database from which there is made views of the data. We found that this was the best solution to make a view, by first storing the data in the graph database and make a view from that, instead of making a view from the raw data stored in Git. The data in the graph database are extracted and analyzed from the raw data in Git. We saw that instead of extracting the data from Git for analyzation, it was possible to let the system instead pass the data directly on for analyzation at the same time they are stored in Git. This way we eliminate the step of extracting the data from Git. To make views of the data in the graph database one needs to construct queries that fetch the data. We saw that it was flexible to make new queries, and thus new views. This was illustrated in code written in Java.

Innholdsfortegnelse

Kapittel 1: Introduksjon 1
Motivasjon 1
Kontekst 1
Problemstillinger 2
Framgangsmåte 2
Resultater 2
Overblikk over rapporten 2
Kapittel 2: Event Sourcing 3
Introduksjon til event sourcing 3
Hendelser og kommandoer 6
Additiv arkitektur 7
Relaterte teknikker 9
MemoryImage 9
CQRS og Event Sourcing 10
Kapittel 3: Event Sourcing i systemet 15
Kapittel 4: Alternative løsninger i systemet 20
Dataene i grafdatabasen 20
Visning av data fra grafdatabasen mot visning av data fra Git 23
Effektivisering av analysen av rådata i Git 24
Lage nye visninger fra data i grafdatabasen 25
Kapittel 5: Konklusjon 27
Oppsummering 27
Diskusjon og evaluering 27
Fremtidig arbeid 28
Referanser 29
Vedlegg A 30

Liste med figurer

Figur 1: Hendelser og tilstander 3
Figur 2: Hendelser og tilstander i en bilnavigasjon 4
Figur 3: Reversible hendelser 5
Figur 4: Kommandoer og hendelser 7
Figur 5: MemoryImage 10
Figur 6: Skrivemodell i CQRS 11
Figur 7: Lesemodell i CQRS 12
Figur 8: Lese- og skrivemodell i CQRS 13
Figur 9: Projeksjoner 14
Figur 10: Kommandoer, hendelser og tilstander i systemet 16
Figur 11: Skrivemodell i systemet 17
Figur 12: Lesemodell i systemet 18
Figur 13: Lese- og skrivemodell i systemet 19
Figur 14: Grafen i grafdatabasen 20
Figur 15: «User»- og «Repo»-node 21
Figur 16: «Repo»- og «File»-node 21
Figur 17: «File»-, «FileState»- og «Category»-node 22
Figur 18: «FileState»-, «Test»- og «Marker»-node 22
Figur 19: Flere visninger i systemet 26

Kapittel 1: Introduksjon

Faget *TDT4100 Objektorientert Programmering* underviser studenter innen programmering. I øvingsopplegget til faget brukes det tester for å sjekke om programkoden som studentene har skrevet gir riktig resultat. I forbindelse med dette har Stein Kjetil Sørhus gjort et arbeid i sin masteroppgave *Applying Learning Analytics in the course TDT4100 at NTNU* [7] der det har blitt registrert data fra kjøring av tester hos studenter i faget. I tillegg har Stein i sitt system også registrert alle endringene studentene har lagret i kode og tekst sammen med eventuelle feil i koden etter hver lagring. Disse dataene blir vist i en visning som gir en oversikt over testresultater og antall feil studentene har i koden over tid.

I denne oppgaven ønsker vi å se på hvordan Event Sourcing kan brukes som en tankegang i et slikt system som Stein har laget, og om Steins system følger denne tankegangen. Oppgaven vil først se på Event Sourcing og hvordan Event Sourcing fungerer. Deretter vil Steins system beskrives i lys av Event Sourcing, og det vil bli utført en analyse av alternative løsninger i systemet.

Motivasjon

I forhold til oppgaveteksten har oppgaven endret seg noe underveis. Vi ønsket opprinnelig å lære mer om bruk av Event Sourcing i forbindelse med læringsanalyse for faget TDT4100, men så oss etter hvert tjent med å se på systemet som Stein har laget i sin masteroppgave, da Event Sourcing også ble relevant der.

Systemet som Stein har laget, har et lager for data som registreres fra studenter i faget TDT4100. Dataene registreres og lagres i lageret når bestemte handlinger utføres. Event Sourcing bygger på en tankegang der man registrerer og lagrer hendelser som skjer i et system for å representere tilstander. Vi ønsker å lære mer om Event Sourcing for å se hvordan Event Sourcing kan brukes i systemet til Stein. Systemet har også en visning av de registrerte dataene og vi ønsker også å se hvor fleksibelt systemet er med tanke på å lage nye visninger.

Kontekst

Systemet som Stein har laget henter inn data fra studenter i faget TDT4100. Registreringen av data skjer fra programmet Eclipse som studentene bruker i øvingsopplegget i faget. Når en student kjører en test eller lagrer en endring i kode eller tekst lagres henholdsvis resultatet av testen og endringen i koden eller teksten på en server. Resultatet av testen inkluderer også hvilken metode som ble testet i koden. Når en endring i kode lagres, blir det også lagret eventuelle feil som Eclipse finner i koden på det tidspunktet. Disse dataene er lagret på serveren først som rådata i Git repositorier i tillegg til i en grafdatabase som fungerer som del av en projeksjon av rådataene der utvalgte data vises i en egen visning. Disse to versjonene av data er del av systemet vi skal se på.

Problemstillinger

Når vi ser på systemet til Stein, har det likheter med Event Sourcing hvor man registrerer hendelser for å kunne representere tilstander. I hvilken grad følger Stein sitt system en tankegang om Event Sourcing, og hvordan kan Event Sourcing brukes i systemet? Når vi ser på systemet i lys av Event Sourcing kan vi også se på noen alternative løsninger for systemet. Er det bedre å lage visning av data rett fra rådataene i stedet for å gå om grafdatabasen, og hvor effektivt er systemet slik det er nå? Det er allerede laget en visning av data fra grafdatabasen, men hvor lett er det å lage nye visninger av data i systemet?

Framgangsmåte

For å besvare problemstillingene beskrevet over vil arbeidet i første omgang innebære et studie av Event Sourcing og relevante områder knyttet til Event Sourcing. Dette studiet vil danne grunnlag for å kunne beskrive Stein sitt system ut i fra Event Sourcing. Deretter vil systemet evalueres i forhold til Event Sourcing, og alternative løsninger i systemet vil bli diskutert. Til slutt vil det sees på hvor lett det er å lage nye visninger ved å implementere funksjonalitet for nye visninger med kode i Java.

Resultater

Studiet av Event Sourcing vil resultere i et eget kapittel om Event Sourcing i rapporten. Kapitlet vil inneholde figurer og modeller som illustrerer Event Sourcing og hvordan Event Sourcing fungerer. Disse vil bli brukt for å beskrive Stein sitt system opp mot Event Sourcing. På grunnlag av det vi har lært om Event Sourcing, og slik Stein sitt system ser ut, ser vi på hvordan alternative løsninger i systemet kan gjøres. Dette vil gi forslag til nye eller bedre løsninger av systemet. Til slutt vil det også produseres kode som viser hvordan man kan lage nye visninger.

Overblikk over rapporten

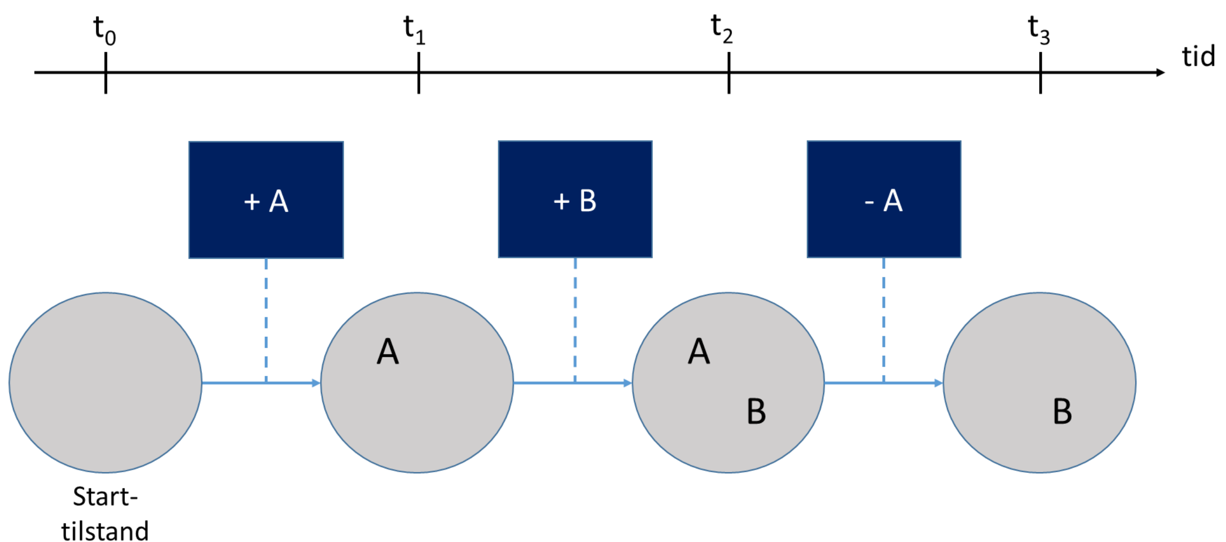
Rapporten vil fortsette med Kapittel 2 der det vil gjøres et studie av Event Sourcing og relaterte områder til Event Sourcing. Deretter vil systemet til Stein beskrives i Kapittel 3 i lys av studiet av Event Sourcing som er gjort Kapittel 2. Videre vil Kapittel 4 diskutere og beskrive alternative løsninger for systemet. Rapporten vil avslutte med Kapittel 5, der alle resultatene oppsummeres og evalueres. Her vil det også sees på mulige videreføringer av systemet. Kode som blir skrevet blir lagt i vedlegg helt til sist i rapporten.

Kapittel 2: Event Sourcing

I dette kapitlet beskrives Event Sourcing og tankegangen i Event Sourcing. Dette gjøres med hjelp av figurer. Relaterte områder der Event Sourcing kan brukes beskrives også.

Introduksjon til event sourcing

Når vi ser på event sourcing er det to veldig sentrale begreper som kommer fram. Det ene er *hendelser*, og det andre er *applikasjonstilstand*. Event sourcing er en måte å representere applikasjonstilstand på ved hjelp av hendelser [3]. Applikasjonstilstand er den tilstanden en applikasjon til enhver tid befinner seg i. Denne tilstanden vil endre seg ved bruk av applikasjonen. Et eksempel er når ruten i en bilnavigasjon endrer seg når føreren tar en avkjørsel, eller når føreren setter opp en ny rute i navigasjonen. Handlingene som føreren utfører lager hendelser i applikasjonen, og disse hendelsene fører til endringer i applikasjonens tilstand.



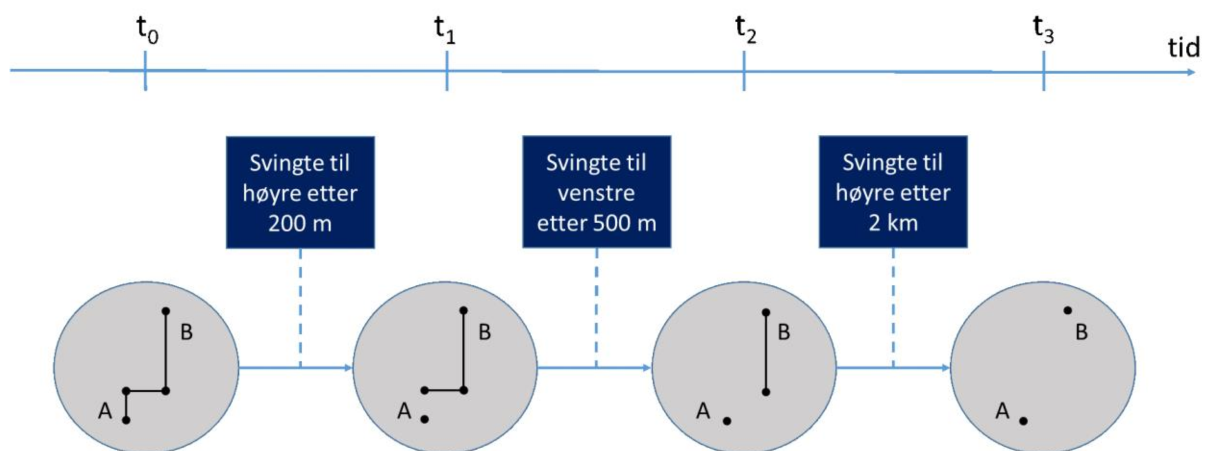
Figur 1: Hendelser og tilstander

I figuren over er Event Sourcing illustrert. Alle hendelser skjer over en tidslinje. Hendelsene er representert i de mørkeblå firkantene, mens sirklene viser applikasjonstilstanden. Vi begynner i en starttilstand ved tiden $\mathbf{t_0}$. Denne kan være en tom tilstand, det vil si før noen hendelser har skjedd, eller det kan være en bestemt tilstand etter at en eller flere hendelser har skjedd. Alle applikasjoner begynner i en tom tilstand, og får tilstanden sin endret når de blir brukt. Noen applikasjoner husker tilstanden de hadde fra forrige gang applikasjonen ble kjørt, for eksempel hvis du lagrer et spill og fortsetter fra der du lagret. De kan fortsette fra den tilstanden som starttilstand.

Vi ser i figuren hvordan hendelsene endrer på tilstanden. I figuren har vi begynt med en tom tilstand og i den første hendelsen ser vi at vi legger til **A**. Da ser vi en endring i applikasjonens tilstand hvor vi har fått **A**. Den neste hendelsen er at vi legger til **B**. Da ender vi opp med en

tilstand hvor vi har både **A** og **B**. Den siste hendelsen fjerner **A** og i den resulterende tilstanden sitter vi igjen med **B**. Dette er den siste tilstanden til applikasjonen, det vil si den nåværende tilstanden.

Det vanlige er at man ser på den nåværende tilstanden i en applikasjon. I det tidligere eksemplet med bilnavigasjon vil dette være hvor du befinner deg akkurat nå på ruten. Men noen ganger kan det hende at du også ønsker å se hvor du har vært, og da blir det nødvendig å vite ruten du har kjørt fram til dit du er. I en bilnavigasjon vil ruten oppdatere seg mens du kjører. Hvis du tar en avkjøring til høyre vil navigasjonen oppdatere hvor langt det er til neste avkjøring og om du skal ta til høyre eller venstre. Her har handlingen å ta avkjørselen til høyre ført til en endring i applikasjonens tilstand. Hvis vi tenker oss at vi lagrer unna alle disse hendelsene, så kan vi bruke de til å rekonstruere ruten vi har kjørt. Dette er essensen i Event Sourcing, og dette konseptet er vist i figuren under.



Figur 2: Hendelser og tilstander i en bilnavigasjon

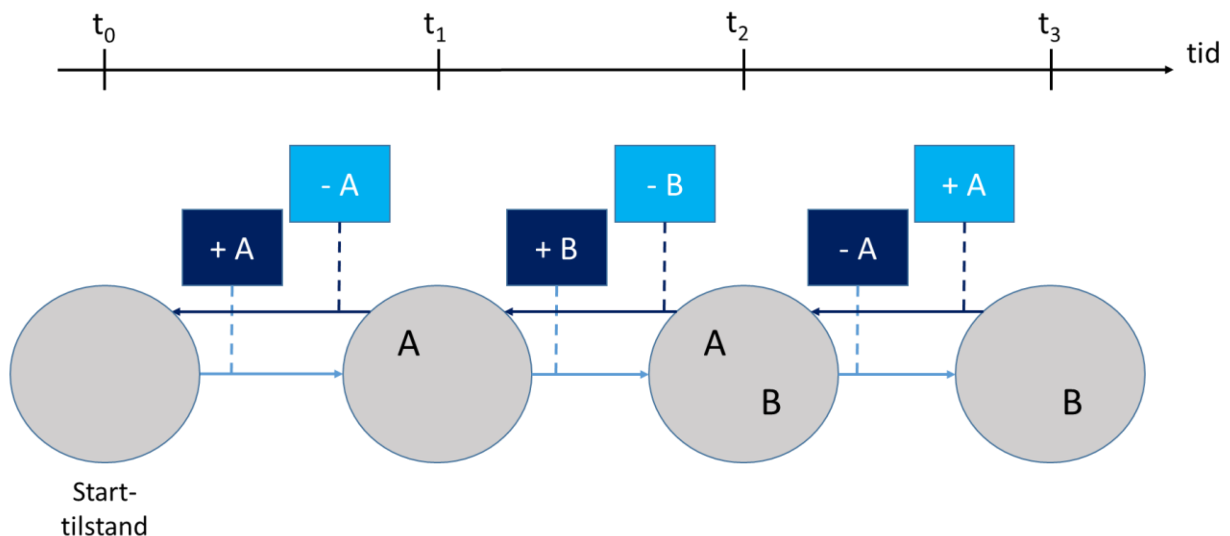
I figuren har vi forsøkt å representere en veldig kort og enkel rute i en bilnavigasjon. Alle hendelsene som skjer er representert i firkantene. De skjer over et tidsrom som er representert med tidslinjen. De forskjellige applikasjonstilstandene er representert i sirklene der vi kan se hvor mye av ruten som er igjen. Vi begynner i starttilstanden som viser hele ruten vi skal kjøre. Etter hvert som vi kjører endrer ruten seg når vi tar de forskjellige avkjørslene. Her kan vi se at endringen i ruten skjer etter at en hendelse har skjedd og at hendelsene utløser endringen i ruten, dvs. de fører til endringer i applikasjonens tilstand.

I figuren over kan vi enkelt se på alle tilstandene, eller delene av ruten. Den siste tilstanden på tidslinjene er tilstanden nå, og denne viser at vi har nådd destinasjonen vår. De andre tilstandene viser ruten på tidligere tidspunkt. La oss tenke oss at bilnavigasjonen lagrer alle endringene i ruten og tidspunktet de skjer på. Da vil du kunne gå tilbake og se hvor på ruten du var for eksempel ti minutter siden. Legg merke til at det er endringene i ruten som lagres, og ikke den gjenstående ruten. Hendelser er enkle å representere, for eksempel «tok den første avkjørselen til høyre». Den gjenstående ruten er mer komplisert å representere fordi vi da må representere alle de gjenstående hendelsene for å nå målet.

Dette er essensielt i Event Sourcing der man lagrer alle hendelsene og bruker de til å representere applikasjonstilstanden i stedet for å lagre applikasjonstilstanden i seg selv. Når

hendelsene lagres er det også viktig å vite i hvilken rekkefølge de oppstod. For eksempel tok du avkjørselen til høyre før du tok avkjørslene til venstre? I bilnavigasjonen er dette veldig viktig for å vite hvor du har vært. Hvis alle hendelsene lagres i en sekvens med den første hendelsen først, vil du ved å spille av denne sekvensen få alle hendelsene i riktig rekkefølge. Du kan også gi alle hendelsene et tidsstempel som viser når hendelsen oppstod. Da vil du også kunne representere applikasjonstilstanden på et bestemt tidspunkt i tid. Uten tidsstempel vil du i bilnavigasjonen kunne se hvor du var etter tre kilometer, mens med tidsstempel vil du også kunne vite hva tiden var på det tidspunktet, eller hvor langt du hadde kjørt etter 10 min.

I situasjonene over kan vi se hvor vi har vært ved å rekonstruere ruten fram til et bestemt punkt. Det betyr at vi begynner fra starten av ruten og spiller av hendelsene som har skjedd opp til det punktet. La oss si at vi kjører en lang rute på kryss av hovedstaden i Norge, Oslo. Her vil mest sannsynlig få en komplisert rute med mange avkjørsler. Dersom vi ønsker å vite hvor vi kjørte i starten av ruten er dette ganske greit å finne ved å rekonstruere ruten fra starten av. Men la si vi ønsker å vite hvor vi kjørte på slutten av ruten, da vil det være litt tungvint å rekonstruere ruten helt fra starten av. I stedet hadde det vært ideelt å kunne gå bakover fra slutten av ruten for å se hvor vi har kjørt. Vi kan oppnå det ved å reversere hendelsene vi har lagret, men da er det viktig at hendelsene er reversible. Dette konseptet forklares nærmere i figuren under.



Figur 3: Reversible hendelser

Vi ser her på samme figur som vi så på i starten av kapitlet. De mørkeblå firkantene er hendelsene som har skjedd, mens de lyseblå firkantene er de reverserte hendelse. I den første hendelsen blir **A** lagt til, og i den reverserte hendelsen blir **A** trukket fra. Den reverserte hendelsen er ikke gitt, den må finnes ved å reversere den opprinnelige hendelsen slik at man kommer tilbake til tilstanden før. Det er heller ikke alle hendelser som er reversible. For å illustrere dette ser vi på et annet eksempel med en bankkonto. La si du setter inn penger på kontoen og lager hendelsen «Oppdater beløpet på kontoen til 500 kroner». Denne hendelsen er ikke reversibel fordi du ikke vet hvor mye beløpet ble

oppdatert med. Hvis du i stedet lager hendelsen «Satte inn 200 kroner på kontoen», vil hendelsen være reversibel fordi du kan reversere den ved å trekke 200 kroner fra beløpet på kontoen. Hvis du i bilnavigasjonen lager en hendelse på at du ankom et sted, men det var flere mulige veier dit, vil hendelsen ikke være reversibel. Da må du i stedet lage en hendelse på at du ankom stedet via en bestemt vei.

Alle hendelser i Event Sourcing lagres i en *hendelseslogg*. Hendelsesloggen er en slags sekvens av hendelsene som har oppstått i en applikasjon, der alle hendelsene er sortert etter når de oppstod. Det er fra hendelsesloggen at vi spiller av hendelser for å representere en bestemt tilstand. I eksemplet med bilnavigasjon vil hendelsesloggen inneholde alle avkjøringer føreren har tatt til nå. Hendelsesloggen er et fast lager for hendelsene, det vil si at du kan slå av applikasjonen og fortsatt ha tilgang til de hendelsene som har skjedd neste gang du starter applikasjonen igjen. Det betyr at du ved oppstart av en applikasjon som bruker en hendelseslogg vil kunne rekonstruere den forrige tilstanden applikasjonen hadde før du slo den av. La oss se på et eksempel på dette i bilnavigasjonen.

La si du er på en bilferie fra Oslo til Paris, og ruten går over flere dager. Da ønsker du gjerne å fortsette på ruten fra dagen før. Ved å bruke en hendelseslogg kan vi lagre unna hendelsene som har skjedd i ruten i løpet av dagen, og neste dag kan vi spille av disse for å vite hvor på ruten vi er kommet. Men hvis du har fullført store deler av ruten allerede, kan dette bli litt tungvint. Da er et alternativ å lagre unna et bilde av ruten ved slutten av dagen når vi skrur av bilnavigasjonen. Det gjør det mulig å i stedet starte fra bildet av ruten og bygge videre på det. Da slipper vi å rekonstruere ruten, men vi vil fortsatt ha hendelsene fra ruten dagen før slik at vi har mulighet til å rekonstruere deler av den. Ved å lagre såkalte snapshots fra perioder med bruk vil det være lettere å starte opp igjen og fortsette ved et senere tidspunkt.

Hendelser og kommandoer

I Event Sourcing har vi hendelser som skjer i domenet, og disse kalles domenehendelser. Det er viktig i applikasjonen å skille mellom disse og kommandoer [8]. En kommando fører til en hendelse, og det er hendelsene vi er interessert i lagre og ikke kommandoene fordi hendelsene er noe som faktisk har skjedd.

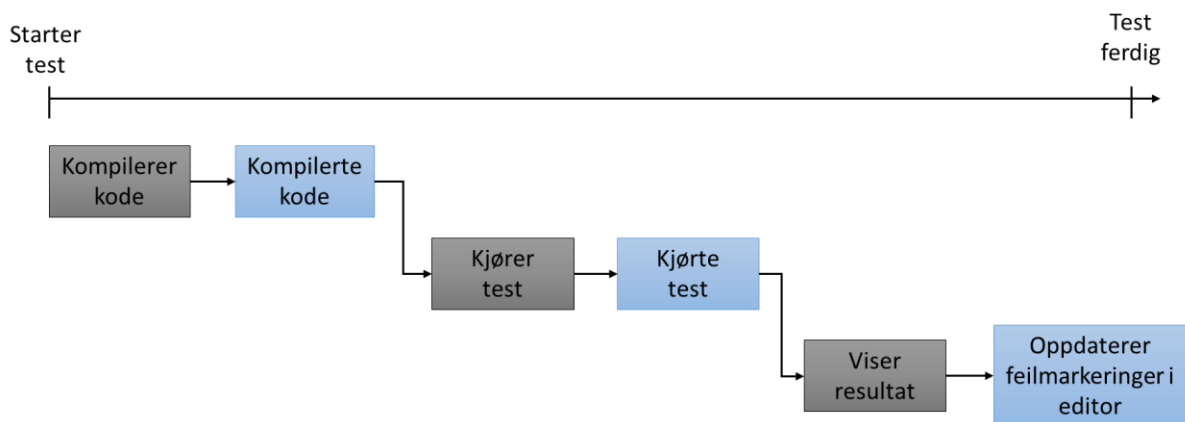
Det som først og fremst skiller hendelser fra kommandoer er navnet på de. «Åpner dokument» er en kommando, mens «Åpnet dokument» er en hendelse. Legg merke til hvordan de to er skrevet. Alle kommandoer skrives i presens (nåtid), mens alle hendelser skrives i preteritum (fortid). En hendelse i Event Sourcing er alltid noe som har skjedd, og ikke noe som kommer til å skje. Derfor er det naturlig å beskrive disse i fortid.

Hendelser kan heller ikke feile. De er noe som har skjedd, og noe som har skjedd kan ikke feile i ettertid. I en prosess kan derimot en kommando feile. For eksempel vil kommandoen «Åpner dokument» kunne feile fordi dokumentet ikke var tilgjengelig for lesing, men det gir ikke mening at hendelsen «Åpnet dokument» feiler fordi den beskriver at dokumentet faktisk ble åpnet. I Event Sourcing vil man aldri beskrive at en hendelse feilet, men man kan

beskrive at en kommando feilet ved å lage en hendelse for feilen som oppstod da kommandoen ble forsøkt utført.

En annen grunn til at det er viktig å skille mellom kommandoer og hendelser er sideeffekter. En sideeffekt er noe som skjer i tillegg og som utløses ved siden av når du utfører en handling. La si du har en nettside hvor kunder kan kjøpe varer. Den naturlige gangen i et kjøp av varer er at kunden først finner varer som legges i en handlekurv, og deretter går til kassen for å betale for de. Varene som kundene kan kjøpe i en nettbutikk ligger på et lager. Når en kunde legger en vare til i handlekurven kan en sideeffekt være at varen blir lagt til sides og reservert på lageret. Du ønsker kanskje ikke at denne sideeffekten skal skje når du spiller av hendelsene som har skjedd. Derfor det viktig å knytte sideeffekter bare til kommandoer og ikke til hendelser.

I eksemplet med en nettbutikk ovenfor ser vi at flere systemer er involvert ved kjøp av varer. Under betaling av varene vil et eget system i kundens bank uføre selve betalingen fra kunden til nettbutikken. Hver av disse systemene utfører sine egne prosesser. Hele prosessen for et kjøp av en vare kan modelleres som en enkelt hendelse, eller den kan deles opp i flere hendelser. Hvert system får en kommando som sier hva de skal gjøre, og deretter lager de en hendelse når det er utført. Under det vist et enkelt eksempel på en prosess der en test av programmeringskode kjøres.



Figur 4: Kommandoer og hendelser

Vi ser at prosessen starter med en kommando «Kompilerer kode». Den fører til en hendelse «Kompilerte kode». Neste steg i prosessen utføres med kommandoen «Kjører kode» som fører til hendelsen «Kjørte kode», og helt til slutt har vi kommandoen «Viser resultat» som fører til hendelsen «Oppdaterer feilmarkeringer i editor». Her ser vi et tydelig skille mellom kommandoer og hendelser, der hver del i systemet får en kommando som fører til en egen hendelse.

Additiv arkitektur

Event Sourcing bygger på en additiv arkitektur der det kun er lov å legge til informasjon [2]. Hendelsene i Event Sourcing lagres i en hendelseslogg der de legges til i en sekvens som inneholder alle hendelsene. Alt som trengs for å legge til en ny hendelse er å legge den til på

slutten av denne sekvensen. Det trengs ingen oppdateringer eller integrering av data. Alle hendelser er uforanderlige, det vil si at de aldri endrer seg [6]. En hendelse er noe helt konkret som har skjedd, og den kan ikke endres i ettertid, og derfor trengs det ingen oppdatering av hendelser.

Hendelser har gjerne med seg informasjon som dato på når hendelsen oppstod i tillegg til informasjon som forteller hva som skjedde. I et eksempel med en overføring av penger til en bankkonto vil det stå en dato i tillegg til hvilken konto beløpet skal overføres til, hvilken konto det ble overført fra og størrelsen på beløpet. Datoen for hendelsen vil aldri endre seg siden den forteller når overføringen skjedde. Beløpet vil heller ikke endre seg, fordi det forteller hvor mye som ble overført på det tidspunktet. Kontoen beløpet ble overført til og kontoen det ble overført fra vil også være uforandret. Her kan vi se at hendelsen er uforanderlig.

Men hva hvis vi overførte for eksempel 200 kroner og ønsker å reversere denne handlingen, det vil si føre de 200 kronene tilbake der de kom fra. Da kan man fort tenke seg at det enkleste er å bare slette hendelsen og føre de 200 kronene tilbake, men det blir helt feil måte å gjøre det på. I Event Sourcing registrerer man alle hendelser som skjer, og det å føre pengene tilbake er også en hendelse som registreres. Det betyr at man aldri sletter hendelser, men i stedet kun reverserer det som skjedde med en ny hendelse. Oppstår det en hendelse der noe legges til i et systemet, kan man senere fjerne det som ble lagt til i systemet, men da ved å lage en ny hendelse som forteller hva som ble fjernet.

Dersom hendelser blir slettet, bryter man serien med hendelser i hendelsesloggen, noe som gjør at man ikke lenger har mulighet for å spille av hele serien. Man mister informasjon om hvilken tilstand systemet var i, siden man ikke lenger har disse hendelsene. Ved å beholde alle hendelser vil man alltid kunne se hvilken tilstand et system har hatt til enhver tid. Dette er spesielt viktig dersom et system skulle oppnå en tilstand som det egentlig ikke hadde lov til å være i, siden man da kan gå tilbake og se akkurat hva som skjedde for at systemet skulle nå den tilstanden. Det kan man bruke til å forebygge slike hendelser i framtiden.

En ren additiv arkitektur som Event Sourcing er, er veldig enkel å forholde seg til. Den er mindre kompleks siden det kun trengs logikk for å legge til og hente ut hendelser. Den er også mer effektiv siden det å legge til og hente ut fra en serie er enkle og raske operasjoner. Dette gjør det også lettere å optimalisere. Hendelsesloggen i Event Sourcing kan for eksempel enkelt optimaliseres ved bruk av horisontal partisjonering.

Horisontal partisjonering er en måte å lagre data på der dataene er lagret på flere forskjellige steder. En nøkkel velges for å definere hvor dataene skal lagres. Hvis du for eksempel skulle ønske å lagre alle telefonnummer, kan du definere en nøkkel som er de to først sifrene i telefonnummeret. Da kan si at for eksempel alle telefonnummer som begynner på 96 skal lagres samme sted, osv. Men man kunne også definert at alle telefonnummer som kommer fra Østlandet skal lagres på samme sted.

Hva som er en god nøkkel kan ofte være vanskelig å definere siden det ofte avhenger av bruken og sammenhenger mellom data. I Event Sourcing er dette derimot mye enklere. Alle hendelser har en ID som øker med hver hendelse. Denne kan enkelt brukes som nøkkel for

hvor hendelsene skal lagres. Da vil du kunne si at for eksempel de først 1000 hendelsene lagres på samme sted, osv. Horisontal partisjonering av hendelsesloggen blir med det veldig enkelt å implementere.

Den additive arkitekturen har en stor fordel i det at du slipper å oppdatere data. Det vanlige i systemer er at man har to kopier av dataene og sammenligner dataene for å se hva som er endret. I Event Sourcing slipper du dette fordi du allerede vet hva som er endret. Det gjør at lagring av data i Event Sourcing er mye raskere.

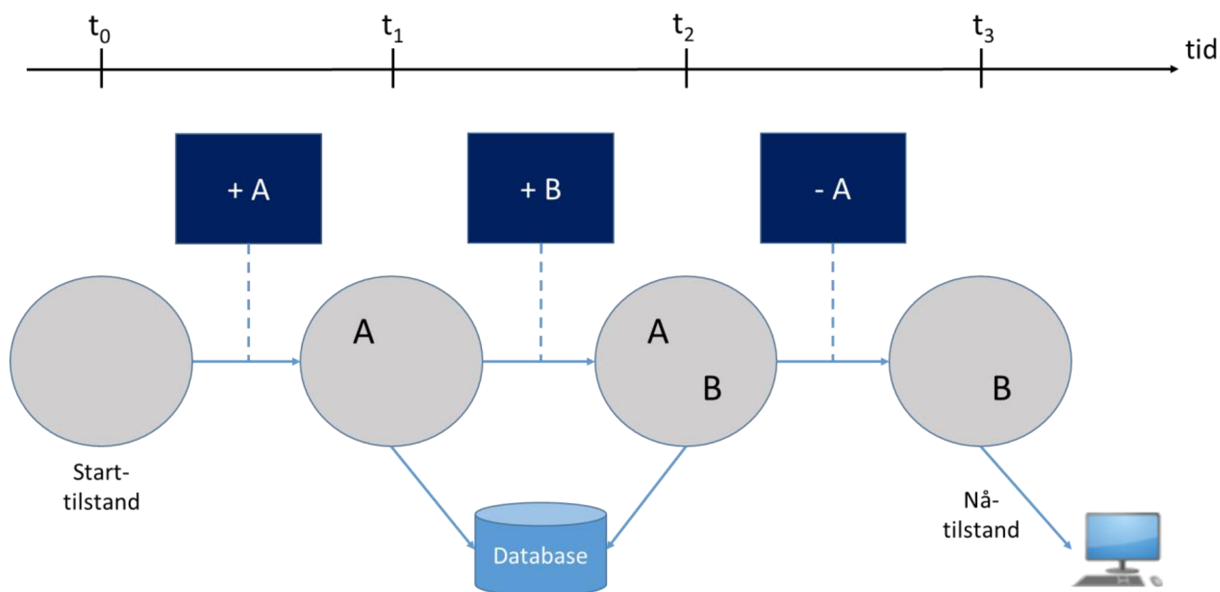
Også når det kommer til lasting av data har Event Sourcing et lite fortrinn. Det er ofte at man har store datasett der det å laste inn alle dataene ofte er veldig tids- og ressurskrevende. For å løse dette kan man laste inn kun de delene av datasettet som trengs. Dette kan fort bli komplekst for veldig store datasett der noen av dataene kan være spredd over forskjellige områder. Med Event Sourcing slipper man derimot dette problemet, da man i stedet bare laster inn hendelsene fra hendelsesloggen og spiller de av.

Men dette kan også bli litt problematisk for store systemer som har millioner av hendelser, siden det vil ta tid å laste inn og spille av alle hendelsene. Da kan man i stedet gjøre som beskrevet under det tidligere eksemplet med bilnavigasjon, hvor man lagrer et bilde av applikasjonstilstanden som kan fortsette fra. Disse bildene representerer tilstanden til en applikasjon på et gitt tidspunkt. Man kan starte fra et slikt bilde og spille av kun de nyeste hendelsene som har skjedd etter tidspunktet da bildet ble tatt. Da slipper man å spille av millioner av hendelser, og kan i stedet spille av kun de siste hendelsene i tid. Dette konseptet er illustrert med bruk i neste del av kapitlet.

Relaterte teknikker

MemoryImage

Når du skal lage en applikasjon som lagrer data, er det naturlig å tenke på databaser. MemoryImage baserer seg på å lagre dataene i minnet i stedet for å lagre dataene i en database [4][5]. For å ikke miste disse dataene kan man bruke Event Sourcing til å lagre endringene i dataene som hendelser i en hendelseslogg. Disse hendelsene kan man utføre på nytt i rekkefølgen de oppstod fra starttilstand for å bygge opp igjen dataene. Dersom det er mange hendelser, kan det bli tidkrevende å bygge opp dataene helt fra bunn av. Da kan man lagre øyeblikksbilder av minnet som man kan bygge opp fra med bare de siste hendelsene. Dette er vist i figuren under. Tilstandene ved t_1 og t_2, som er tidligere tilstander, kan lagres i en database. Tilstanden ved t_3, som er tilstanden nå, lagres i minnet på datamaskinen.



Figur 5: MemoryImage

En stor fordel med MemoryImage ar at man slipper å lagre selve dataene i en database, man kan i stedet lagre de rett i minnet. Dette er mye raskere fordi man slipper unna mye databaselogikk. Men siden alle dataene lagres i minnet, må man sørge for å ha nok minne. Det betyr at man må beregne mer minne enn man trenger. Man slipper ikke helt unna databaser da man må ha et sted å lagre hendelsesloggen. Denne ligger gjerne lagret i en database, men logikken for å lagre unna hendelsene er lettere og mindre kompleks fordi den bygger på en additiv arkitektur som vi har sett tidligere. Under er det beskrevet et eksempel på bruk av MemoryImage og Event Sourcing sammen.

La oss si at vi har et system som støtter CRUD-operasjoner, det vil si oppretting, lesing, oppdatering og sletting av data. I dette systemet har vi dataobjekter som lagres i minnet. Her har man mulighet til å opprette, lese, endre og slette objekter. Alle dataobjektene er lagret i en samling med en identifikator som nøkkel for hvert objekt. Denne samlingen representerer applikasjonstilstanden. Operasjoner som oppretter, endrer eller sletter objekter blir utført som hendelser som lagres i en hendelseslogg i et varig lager på disk. Disse operasjonene vil endre tilstanden til applikasjonen, og ved å spille av hendelsene vil man kunne gjenskape en bestemt applikasjonstilstand. Dette har man bruk for dersom man skulle trenge å gjøre en omstart av systemet eller hvis systemet krasjer. Over tid vil man få veldig mange hendelser i loggen, og det vil ta tid å bygge opp igjen applikasjonstilstanden ved å spille av alle disse. Da kan man kan lage regelmessige øyeblikksbilder av minnet som man kan starte med når man bygger opp applikasjonstilstanden, slik at man kun trenger å spille av de nyeste hendelsene.

CQRS og Event Sourcing

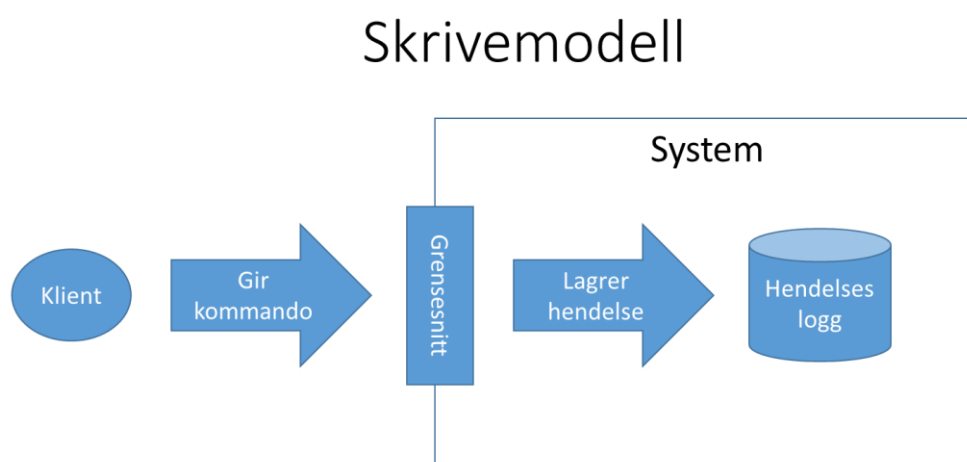
Når du skal lese og skrive data trenger du en datamodell. Det vanlige er at du lager funksjonaliteten for både å skrive og lese data i samme datamodell. Ofte kan dette gjøre at du får en stor og kompleks modell. Her kommer CQRS inn i bildet. CQRS, som står for

Command Query Responsibility Segregation, er et mønster for arkitekturen i en applikasjon der man skiller lesing og skriving av data fra hverandre i to forskjellige modeller [1].

CQRS bygger på CQS (*Command Query Seperation)* som har samme tankegang for et objekt. I CQS skiller man et objekts metoder inn i metoder som endrer på tilstand og metoder som kun leser eller spør etter data. Et godt eksempel på det er et objekt som holder en liste med navn. Funksjonalitet for å legge til et navn i listen vil være i en egen metode, mens funksjonalitet for å hente ut ett av navnene vil være i en annen metode. Legg merke til at du endrer tilstanden når du gjør endringer i listen. Å legge til navn i listen vil gjøre en endring i listen, men å hente ut et navn vil ikke gjøre noen endring i listen med mindre navnet fjernes fra listen etter at det er hentet ut. La si at du har en liste som representerer et lager med varer. Hvis du har en metode som henter en vare fra listen og deretter fjerner den fra listen, som i et lager hvor du tar en vare fra hyllen og gir varen til en kunde, vil du få en metode som både endrer tilstanden og henter ut data. En slik metode vil ikke følge CQS.

CQRS bruker samme prinsippet om å skille ut metoder som endrer tilstand ved å skille kommandoer fra spørringer etter data. Alle kommandoer i et system som fører til en endring i tilstand blir utført i en egen modell som kalles for *skrivemodell*. Alle spørringer som henter data og ikke endrer tilstand blir utført i en annen modell som kalles for en *lesemodell*. Ved å holde metoder som endrer tilstand for seg selv, kan man være helt trygg på at man ikke endrer på tilstand når man spør etter data. Det blir også mye enklere å gjøre endringer i systemet fordi man vet hvilke metoder som endrer på tilstand og bør endres på.

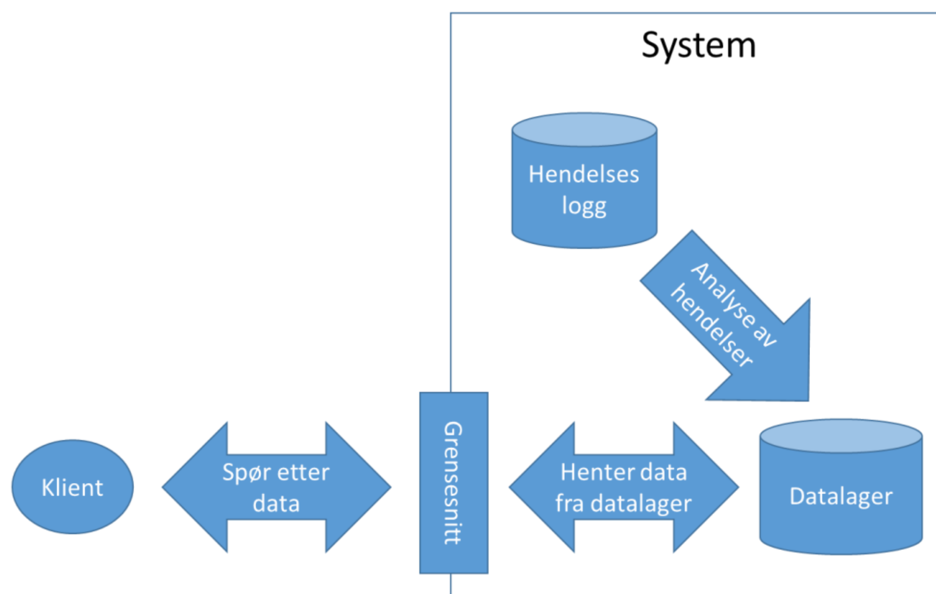
Vi kan fra dette se at CQRS og Event Sourcing passer godt sammen. I Event Sourcing lagres det hendelser når det skjer en endring i tilstand, og dette passer inn i skrivemodellen til CQRS. Da kan man lagre en hendelse i hendelsesloggen når en kommando blir utført og det skjer en endring i tilstanden. Hendelsesloggen blir til lageret for data. Dette er vist i en figur og beskrevet nærmere under.



Figur 6: Skrivemodell i CQRS

I figuren over ser vi skrivemodellen som er den ene delen av et system som bruker CQRS og Event Sourcing. Vi kjenner igjen hendelsesloggen og lagringen av hendelser fra Event Sourcing. En klient gir kommandoer til systemet via systemets grensesnitt, og disse generer da hendelser som lagres. Klienten får også et svar tilbake på at kommandoen er mottatt og utført. Grensesnittet består av forskjellige metoder eller funksjoner som er synlige for klienten. Når klienten kaller på disse, gis det kommandoer til systemet. Systemet utfører da tilhørende operasjoner, og en eller flere hendelser genereres. Disse lagres direkte inn i hendelsesloggen. Dette er akkurat som i Event Sourcing, der hendelsesloggen er lageret for data.

I skrivemodellen gir man kommandoer som generer hendelser, man hva hvis man ønsker å hente data fra systemet? Da trenger vi en lesemodell. Her kan man enkelt tenke seg at det er hendelsene som leses ut fra hendelsesloggen og sendes tilbake til klienten. Men hva hvis klienten skulle ønske å vite data som for eksempel hvor mange kunder i et system som har fornavn «Ola»? Det er et eksempel på data som ikke er like lett å hente ut dersom du kun har en hendelseslogg med hendelser. Da må du gå igjennom og analysere alle hendelsene for å finne ut hvor mange av kundene som har fornavnet «Ola», og det kan fort bli tungvint hvis du har veldig mange hendelser du må gå igjennom.

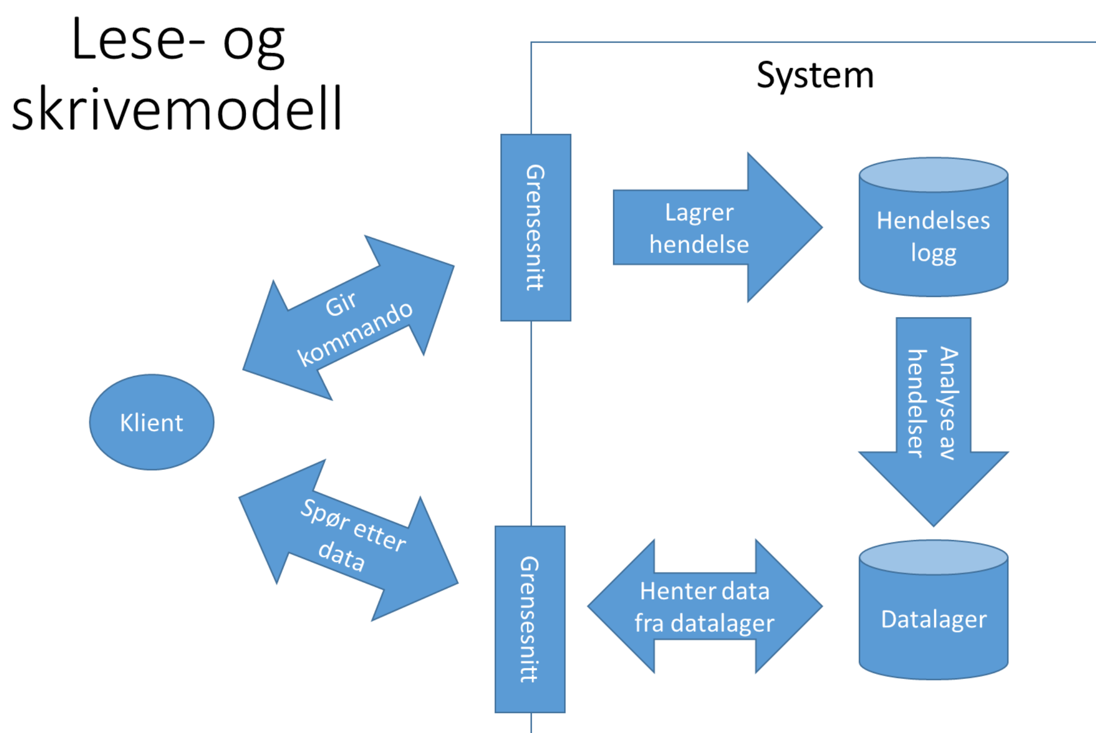


Figur 7: Lesemodell i CQRS

For å hjelpe med dette kommer lesemodellen i et system med CQRS og Event Sourcing inn i bildet. Hendelsene som lagres i hendelsesloggen sendes videre til lesemodellen som analyserer de for relevante data. Disse dataene lagres deretter i et datalager som det kan spørres etter spesifikke data fra. Når en klient etterspør data benytter den seg av et eget grensesnitt som henter dataene fra datalageret og returnerer de til klienten. Dette er vist i figuren over. Her ser vi at klienten spør etter data fra grensesnittet som sender en spørring

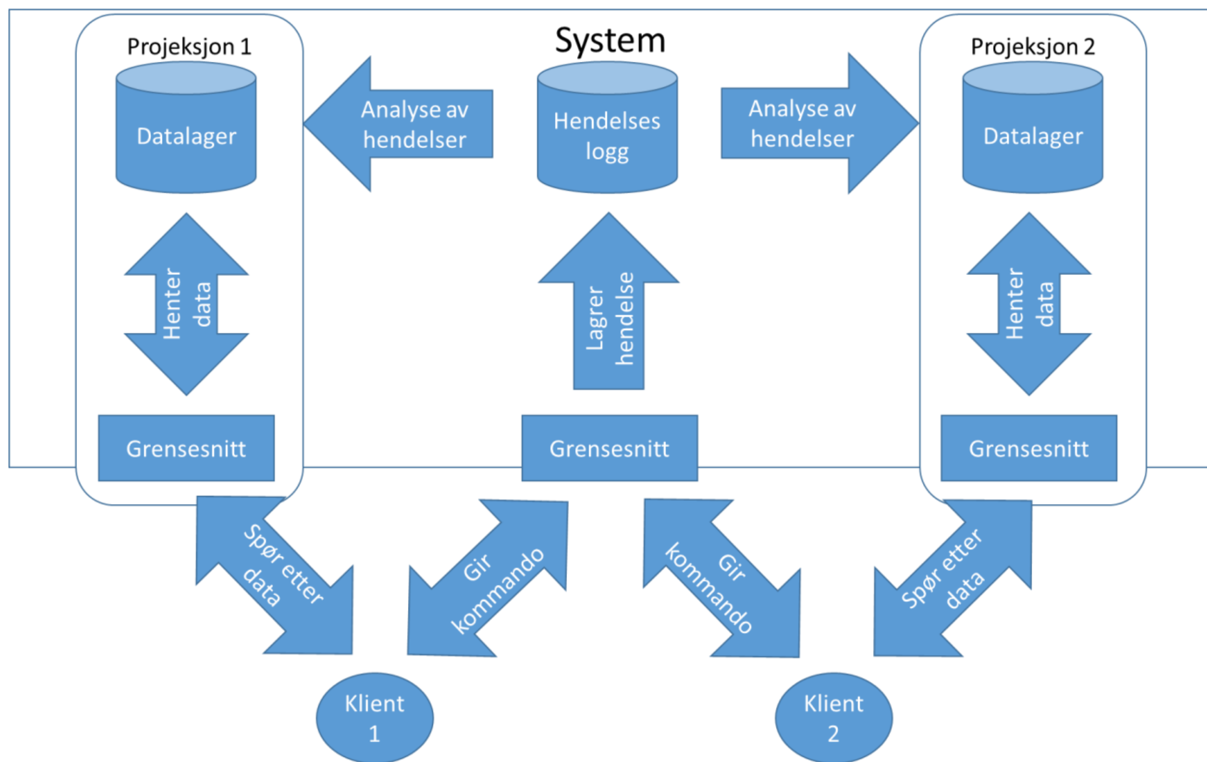
videre til datalageret for å hente ut data. Dataene blir da returnert fra datalageret til grensesnittet som sender de videre til klienten. Vi ser også at dataene som ligger i datalageret er et resultat av analyse av hendelsene i hendelsesloggen.

Når man setter skrivemodellen sammen med lesemodellen, får man hele systemet av CQRS med Event Sourcing. Dette er vist i figuren under der de to forrige figurene er satt sammen i samme figur. Her ser vi at både en hendelseslogg og et datalager ligger i bunn. Klienten snakker mot to forskjellige grensesnitt, der det ene representerer skrivemodellen og det andre representerer lesemodellen. Gangen i systemet blir at klienten sender kommandoer til det ene grensesnittet som genererer hendelser som lagres i hendelsesloggen. Når en hendelse lagres i hendelsesloggen blir den også sendt videre for å analyseres for data som lagres i et eget datalager. Klienten kan da senere hente disse dataene ut fra datalageret ved hjelp av spørringer mot det andre grensesnittet. Grensesnittene definerer hvilke kommandoer som kan brukes og hvilke data som kan etterspørres.



Figur 8: Lese- og skrivemodell i CQRS

Men ofte er det slik at forskjellige klienter ønsker forskjellige visninger av dataene som lagres. Med CQRS og Event Sourcing er dette mulig å gjøre ved å lage flere forskjellige grensesnitt hvor det kan etterspørres forskjellige data. Disse grensesnittene kan tilpasses klientens behov. Alt du trenger er å lage flere datalager, ett for hvert grensesnitt. Dataene i hvert datalager blir kun de dataene som kan etterspørres gjennom det tilhørende grensesnittet. Ved analysen av data avgjør du i hvilket datalager dataene skal lagres ved å se på dataene som hendelsene inneholder. Hvert datalager med tilhørende grensesnitt kalles for en projeksjon, og du kan ha flere projeksjoner som vist i figuren under.



Figur 9: Projeksjoner

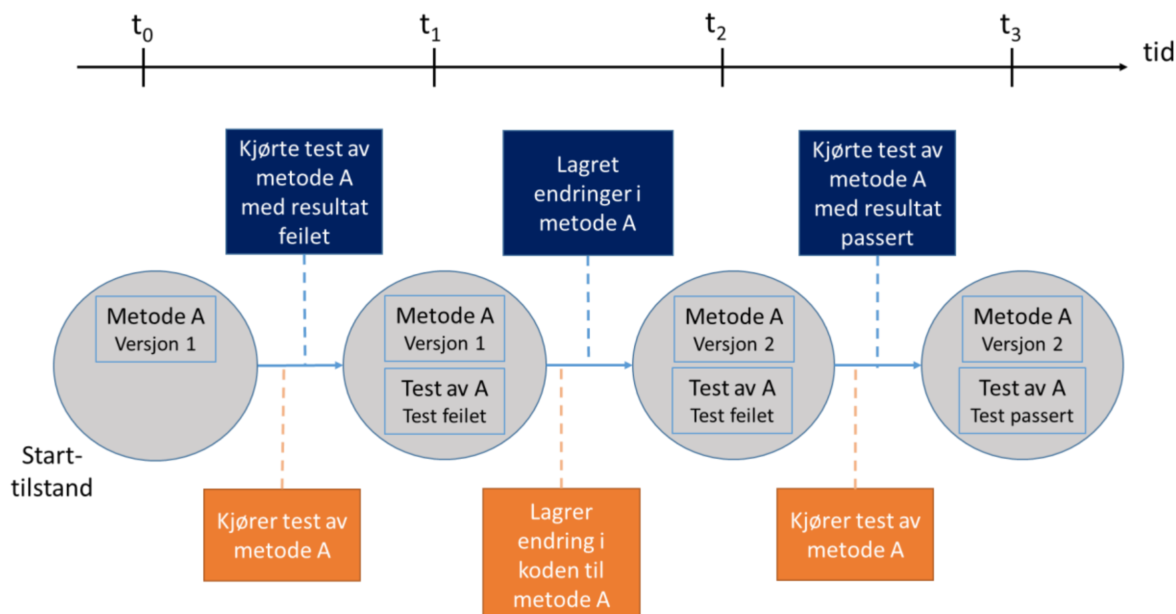
I figuren ser vi to klienter. Klient 1 spør etter data fra projeksjon 1, og klient 2 spør etter data fra projeksjon 2. Men vi ser at begge klientene snakker mot et felles grensesnitt for kommandoer. Det trengs kun ett grensesnitt for kommandoer da alle hendelsene lagres i samme hendelseslogg, men dataene man ønsker å få ut av hendelsesloggen kan være forskjellige slik at man får flere projeksjoner av data med hvert sitt datalager og grensesnitt. Vi ser kun to projeksjoner i figuren, men her er det ingen grense. Det kan være så mange projeksjoner som trengs, og flere klienter kan også spørre etter data fra samme projeksjon.

Kapittel 3: Event Sourcing i systemet

Dataene til Stein Kjell Sørhus som ble kort beskrevet i introduksjonen er lagret i to versjoner. Den første versjonen er rådata som er registrert og lagret som en samling av Git repositorier fra studenter i faget TDT4100. I den andre versjonen finner vi data som er hentet ut fra rådataene og lagret i en grafdatabase med noder og relasjoner. Grafdatabasen fungerer som prosesserte rådata som er lettere å jobbe med og vise. Vi skal i dette kapitlet se nærmere på hvordan disse to versjonene av de samme dataene relaterer seg til det vi har beskrevet om Event Sourcing i forrige kapittel.

Alle dataene som registreres fra studentene i faget lagres i Git repositorier på en server. Hvert Git repository inneholder alle prosjektmappene med all kode og tekst som hver student har skrevet i programmet Eclipse. Eclipse er et program som studentene bruker for å programmere og løse øvingsoppgavene i faget TDT4100. I tillegg inneholder Git repositoriene data om tester som er kjørt i Eclipse og data om feil som studentene hadde i koden sin da den ble lagret, og disse to dataene er lagret i hver sin egne fil innunder prosjektmappene. Hver gang en student kjører en test eller lagrer endringer i kode eller tekst i Eclipse, oppdateres et Git repository som er knyttet til den studenten med testresultatet eller endringen i tekst og kode i tillegg til eventuelle feil som Eclipse fant i koden.

Legg merke til at når Git repositoriet oppdateres er det kun endringer som lagres. Dette er akkurat som i Event Sourcing. Hvert Git repository er en egen hendelseslogg der alle endringer i tilstand i studentens prosjektmapper i Eclipse er lagret. Hvis vi husker tilbake til kommandoer og hendelser, ser vi at når en student kjører en test eller lagrer endringer som er gjort i kode, så blir dette akkurat som kommandoer som utløser en endring i tilstanden til filer. Disse kommandoene genererer hendelser, som er endringene som ble gjort, og hendelsene lagres i studentens Git repository. I figuren under vises det hvilke kommandoer og hendelser som oppstår og hvordan det endrer tilstanden i Git repositoriene.



Figur 10: Kommandoer, hendelser og tilstander i systemet

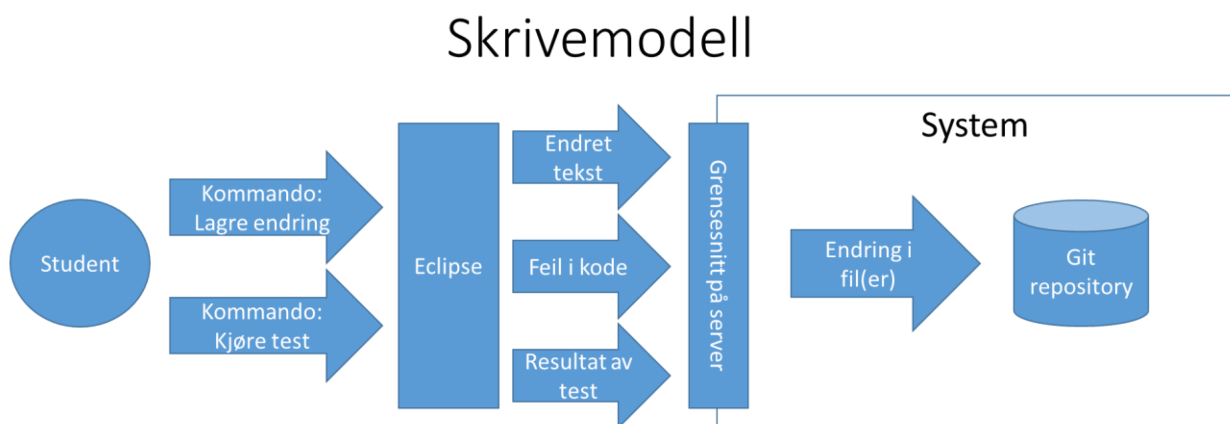
Sirklene representerer tilstandene i Git repositoriet for en student, og firkantene inni sirklene representerer filer med data. I starttilstanden $\mathbf{t_0}$ ser vi at metode A allerede er lagret i en fil Git repositoriet. De oransje boksene viser kommandoene og de mørkeblå boksene viser hendelsene. Alle kommandoene fører til hendelser som fører til en endring i tilstanden i Git repositoriet. Den første kommandoen kjører en test av metode A. Etter at kjøringen er ferdig lagres informasjon om testen med blant annet resultatet av testen og hvilken metode som ble testet. Som vi kan se inneholder Git repositoriet ikke noe data om kjøring av test for metode A i tilstanden t_0, så disse dataene blir opprettet slik at vi får tilstanden $\mathbf{t_1}$. Vi kan se i hendelsen at resultatet av testen var at den feilet, og vi kan også se dette i tilstanden t_1.

Den neste kommandoen som utføres er at studenten lagrer endringer han/hun har gjort i metode A, for eksempel for å prøve å rette opp metoden slik at den passerer testen. Når studenten lagrer endringen i koden oppstår det en hendelse som inneholder endringen, og denne lagres i Git repositoriet. Vi får da tilstanden $\mathbf{t_2}$ der vi kan se at metode A har blitt endret til versjon 2. Etter dette utføres det en ny kommando der testen av metode A kjøres på nytt. Denne kommandoen fører til en ny hendelse med et nytt testresultat, denne gangen at testen er passert. I den siste tilstanden $\mathbf{t_3}$ ser vi da at testen av metode A har fått endret resultat til passert.

Firkantene inni hver sirkel representere filer, og ut i fra det kan vi se at metodene og data om test av metodene lagres i hver sine filer. Det er en type data til som lagres i egne filer, og det er data om feil som Eclipse har funnet i koden ved lagring. Dette er ikke vist i figuren, men dataene om feil i koden lagres når endringer i koden lagres. Det betyr at dersom kommandoen for å lagre endringene i koden til metode A i figuren under hadde ført til en feil i koden til som Eclipse ville gjenkjent, ville det også bli lagret i Git repositoriet data om denne feilen i en egen fil. Hvis denne filen allerede eksisterte ville den blitt oppdatert, akkurat som for filen med informasjon om tester.

Legg merke til at når vi ser på den siste tilstanden ser vi at metode A har passert testen, men vi ser ikke noe informasjon i denne tilstanden om at metode A tidligere hadde feilet testen. Men Git er et system med versjonshåndtering som lagrer alle endringene som skjer. Disse kan spilles av akkurat som hendelser i Event Sourcing. Som følge av dette blir det da mulig å gå tilbake til en tidligere tilstand av Git repositoriet hvor vi kan se at testen av metode A feilet. Git lagrer også tidspunktet til de forskjellige endringene, og da blir det også mulig å se når i tid at testen for metode A feilet. Dette gjør Git til et smart system for å lagre unna testresultater, endringer i tekst og kode og feil i kode.

Når vi ser nærmere på dette systemet, så kan vi se at det er nesten helt likt skrivemodellen i CQRS med Event Sourcing. Studenten blir klienten som sender kommandoer til Eclipse ved å kjøre tester eller lagre endringer i tekst. Hvis en student lagret en endring i tekst, sender Eclipse den endrede teksten med eventuelle feil som ble funnet dersom teksten var kode videre til et grensesnitt på serveren som inneholder samlingen av alle Git repositoriene. Hvis det var en eller flere tester som ble kjørt, så er det resultat av testen(e) som Eclipse sender videre til grensesnittet på serveren. Grensesnittet lagrer dataene det får tilsendt som endringer i filer i det Git repositoriet som tilhører studenten det ble sendt data fra. Vi kan se dette illustrert i figuren under.

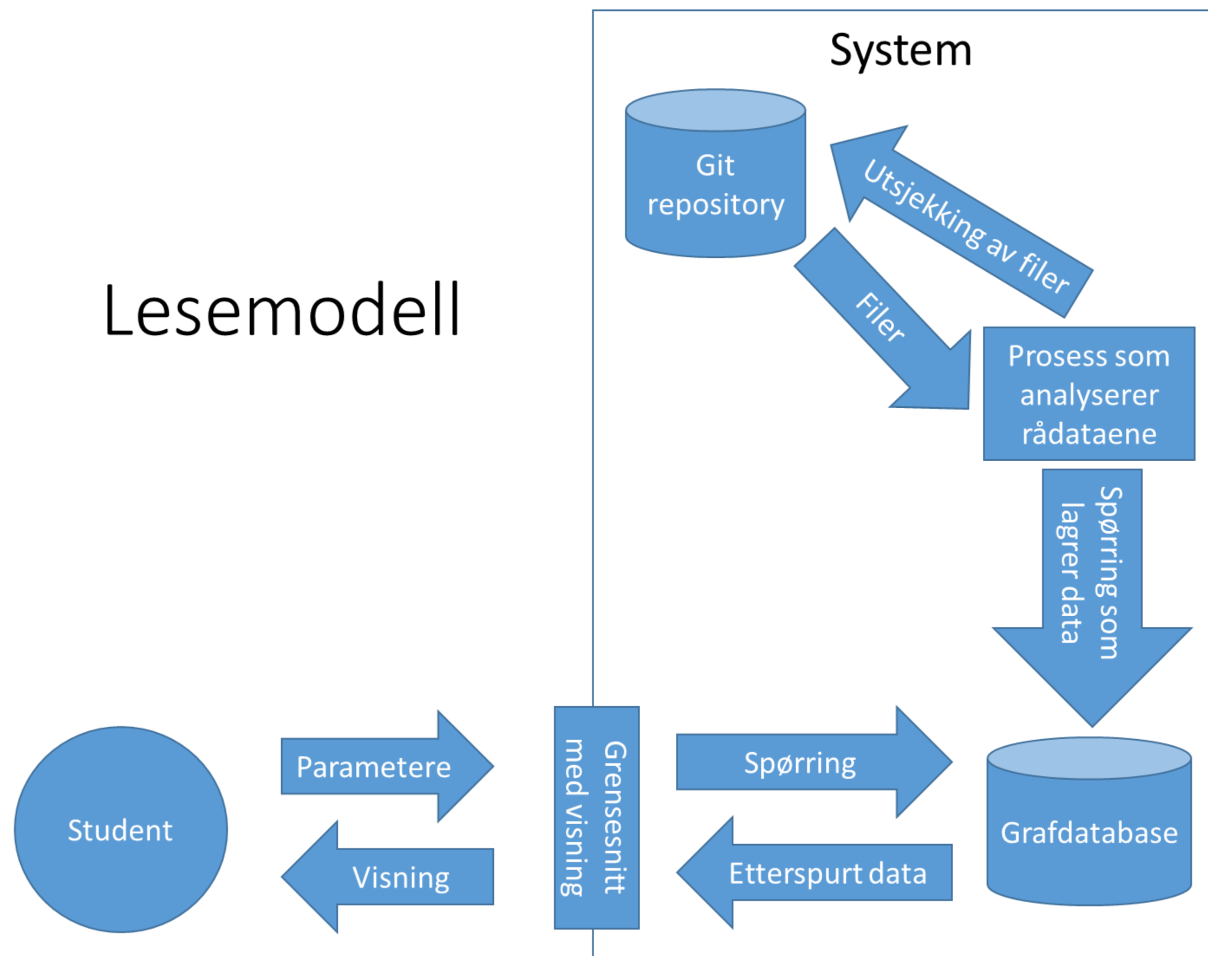


Figur 11: Skrivemodell i systemet

Som vi ser i figuren er det endringer i filer som lagres i Git repositoriet. For å hente ut en fil i seg selv fra Git, er man nødt til å sjekke ut en versjon av filen. Å sjekke ut en versjon av en fil betyr å hente ut filen i en bestemt tilstand. Disse tilstandene får man ved å bygge de opp fra endringene som er lagret i Git repositoriet, akkurat som i Event Sourcing der man kommer fram til en tilstand ved å spille av hendelser fra hendelsesloggen. Her blir endringene i filene til hendelser og Git repositoriet blir til en hendelseslogg.

Figuren viser hvordan rådataene som er lagret i Git repositorier passer inn under skrivemodellen i CQRS med Event Sourcing. Den andre versjonen av dataene inneholder data som er hentet ut fra rådataene, og er lagret i en grafdatabase. Disse dataene er metadata, det vil si «data om data», og de er produsert ved analyse av dataene i Git repositoriene. Grafdatabasen de er lagret i har mulighet til å gi en visning av dataene som noder med relasjoner eller som data i tabeller.

Hvis vi husker tilbake til lesemodellen for CQRS med Event Sourcing, så husker vi at for å hente ut data som for eksempel alle kunder med fornavnet «Ola» fra hendelsesloggen, måtte vi først analysere hendelsene. Analysen produserte data som ble lagret i egne datalager som en klient kunne spørre etter data fra. Det er akkurat slik grafdatabasen fungerer også. I Git repositoriene ligger alle hendelsene som rådata, og disse blir analysert for å hente ut relevante data som lagres i grafdatabasen. Klienten kan så spørre etter dataene fra grafdatabasen ved hjelp av spørringer. Vi ser her hvordan grafdatabasen passer inn i lesemodellen, og dette er illustrert i figuren under.



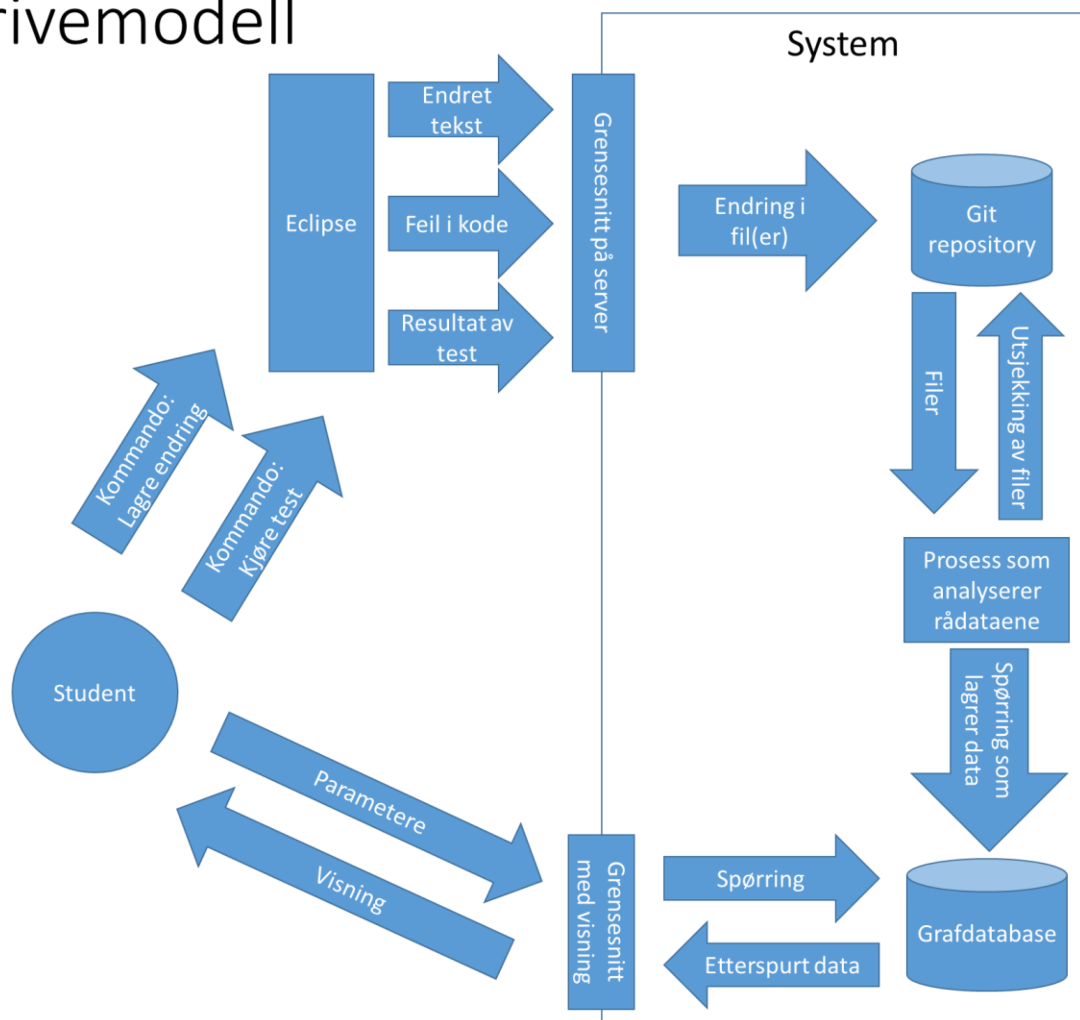
Figur 12: Lesemodell i systemet

Analysen av rådataene i Git skjer i en egen prosess som sjekker ut versjoner av filer fra Git repositoriene. Disse analyseres for data om tester, feil i kode og antall linjer i hver fil. I tillegg sees det på hvilken øving filene hører til, og det lages relasjoner mellom filer, versjoner av filer, øvinger, tester og feil i kode. Alt dette formuleres som data i spørringer som sendes til grafdatabasen som lagrer dataene i noder med attributter og relasjoner.

For å hente ut data fra grafdatabasen kan man bruke spørringer. Hvis man gjør dette mot grafdatabasen sitt egne grensesnitt i en nettleser kan man se resultatet som en graf med noder og relasjoner eller man kan se resultatet i en tabell. For en enklere visning har Stein laget et eget grensesnitt som henter dataene ut fra grafdatabasen med spørringer og lager

en egen visning av de. Dette blir akkurat som en projeksjon, der studenter eller andre brukere kan sende parametere til eget grensesnitt for å få en spesifikk visning av dataene.

Hvis vi setter sammen figuren for skrivemodellen med figuren for lesemodellen får vi figuren under som viser systemet i sin helhet. Vi ser at de to versjonene med data danner grunnlaget for et system som bygger på CQRS med Event Sourcing der Git repositoriene blir hendelsesloggen i systemet og grafdatabasen blir datalager for en projeksjon.



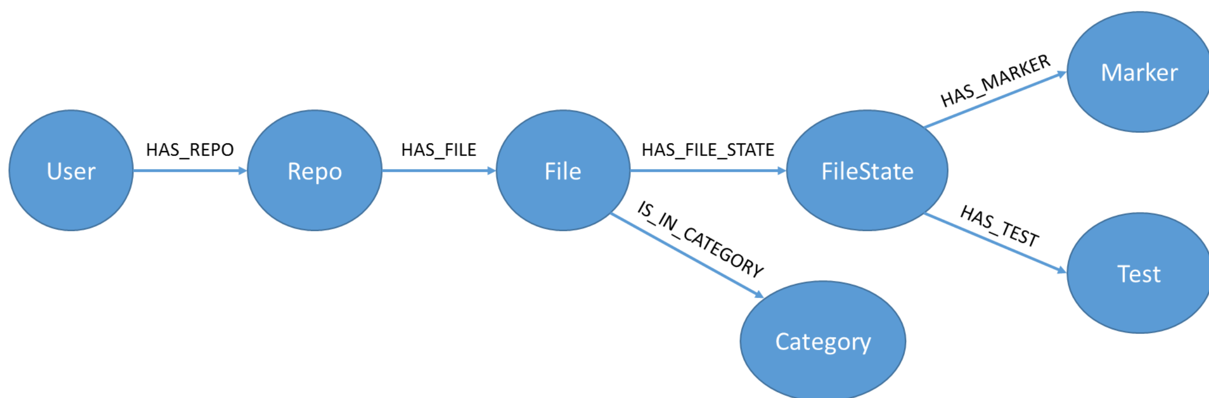
Figur 13: Lese- og skrivemodell i systemet

Kapittel 4: Alternative løsninger i systemet

I forrige kapittel så vi at Git repositoriene sammen med grafdatabasen følger tankegangen i CQRS med Event Sourcing. I dette kapitlet ser vi på alternative løsninger i systemet. Vi begynner med å se på muligheten for å lage visning av data rett fra Git repositoriene i stedet for å gå om grafdatabasen. For å gjøre dette begynner vi med å se mer i detalj på dataene som ligger i grafdatabasen. Etter det ser vi på hvor effektiv løsningen med grafdatabasen er i systemet og, til slutt ser vi på muligheten for å lage flere forskjellige visninger eller projeksjoner fra grafdatabasen.

Dataene i grafdatabasen

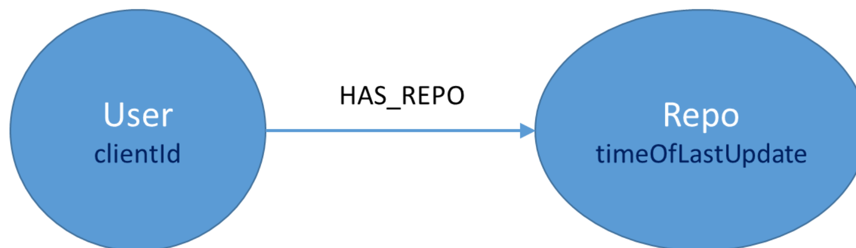
Dataene som er lagret i Git repositoriene er hovedsakelig filer med kode som ligger i prosjektmapper. Under prosjektmappene ligger det i tillegg filer som inneholder data om tester og feil i kode. Disse filene inneholder data som forteller hvilke tester en student har kjørt og med hvilket resultat, samt hvilke feil studenten har i koden. Det er først og fremst disse dataene som blir hentet ut og lagret i grafdatabasen. I grafdatabasen er alle dataene representert som noder med attributter, med relasjoner mellom noder for å vise sammenhenger. Relasjonene mellom nodene danner en graf der den første noden representerer en bruker som er en student. Dette er vist i figuren under.



Figur 14: Grafen i grafdatabasen

Setter man alle nodene sammen med relasjoner, får man grafen som er vist i figuren. Det finnes en slik graf for hver student det er registrert data fra. Grafen begynner med noden «User». I hver graf kan en node av typen «User» kun ha relasjon til én node av typen «Repo», da hver student er knyttet til ett Git repository. En node av typen «Repo» kan ha relasjon til flere noder av typen «File», siden et Git repository gjerne inneholder flere filer. Hver node av typen «File» har relasjon til kun én node av typen «Category», men flere noder av typen «File» fra forskjellige grafer kan ha relasjon til samme node av typen «Category». Det betyr med andre ord at en fil hører til en øving, men en øving kan ha flere filer, og filene kan være spredd over flere studenter. Alle noder av typen «File» har en eller flere relasjoner til noder av typen «FileState» som representerer tilstandene til filen. En fil har flere tilstander dersom den har blitt endret på. Hver node av typen «FileState» har relasjon til en

eller flere noder av typen «Marker» og «Test». I hver tilstand en fil har hatt, kan det ha vært kjørt en test på en eller flere metoder i koden i filen, og det kan ha vært en eller flere feil i koden til filen. Dette er representert i «Test» og «Marker» nodene som er de siste nodene i grafen. De har ingen relasjoner videre. Hver node er forklart nærmere i egne figurer under.

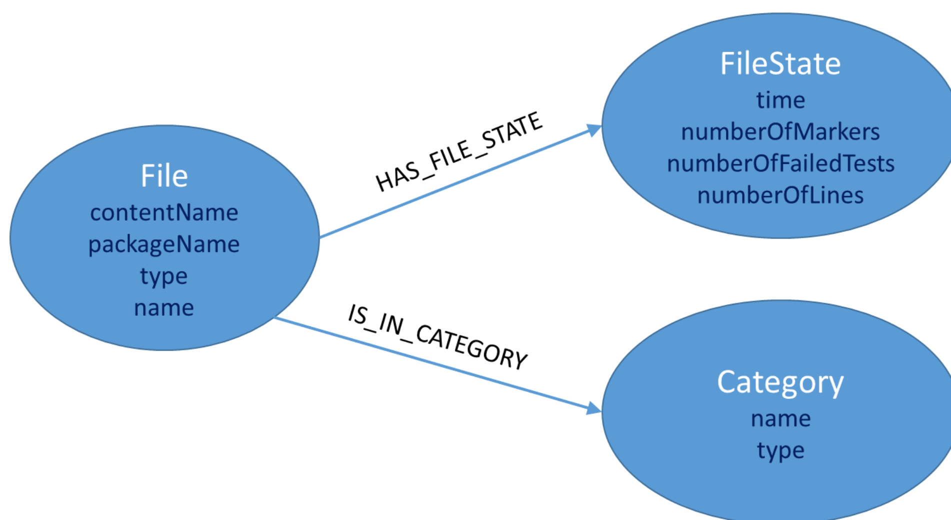


Figur 15: «User»- og «Repo»-node

Grafen starter med «User»-noder som representerer studenter i faget TDT4100 som det er registrert data fra. Disse nodene er startnoder i grafen, og de har ingen relasjoner mellom hverandre, men de har relasjon videre til hver sin egen «Repo»-node. Hver node som representerer en student har en unik identifikator som brukes for å identifisere hver enkelt student. Identifikatoren er lagt til som en attributt på noden. Hver «Repo»-node som «User»-nodene har relasjon til representerer brukerens Git repository. Disse nodene har et tidsstempel for når Git repositoriet sist var endret. Hver bruker har kun ett Git repository, og derfor har hver bruker-node relasjon til kun én «Repo» node. I et repository befinner det seg flere filer og hver av disse er representert i egne «File»-noder.

Figur 16: «Repo»- og «File»-node

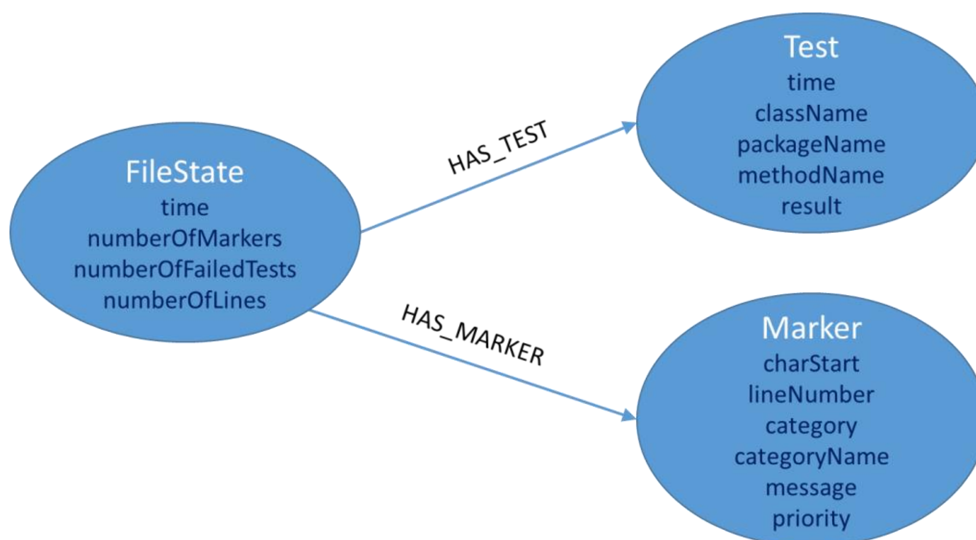
«File»-nodene inneholder navnet på filen, som er satt til filstien den har i repositoriet. I tillegg har disse nodene med attributter for pakkenavn (som er det samme som mappen filen ligger i), type (klasse, grensesnitt, e.l.) og navn på innholdet. En «Repo»-node kan ha relasjoner til en eller flere «File»-noder.



Figur 17: «File»-, «FileState»- og «Category»-node

Hver «File»-node har relasjon videre til noder som representerer de forskjellige tilstandene til filen i tid. Hver «FileState»-node har tid som hoved-attributt, som forteller tidspunktet da en fil hadde denne tilstanden. I tillegg har «FileState»-nodene attributter som forteller om antall linjer i filen og antall feil i koden til filen. Det er også et attributt for antall tester for filen som har feilet, men verdien for attributtet ser ut til å være ukjent.

Noen av filene hører også til øvinger eller kategorier. De «File»-nodene som hører til en kategori eller øving har en relasjon til én «Category»-node. Flere filer kan høre til samme «Category»-node, og noden har attributtene navn og type som beskriver hvilken kategori eller øving den representerer.



Figur 18: «FileState»-, «Test»- og «Marker»-node

«FileState»-nodene har relasjon videre til noder for feil i kode og noder for tester. En fil kan ha flere tester knyttet til seg, og det kan være flere feil i koden. Det kan også være ingen av delene for filen. «Test»-nodene har attributter for tid, klassenavn, pakkenavn, metodenavn

og resultat. Resultat forteller hvorvidt testen passerte eller ikke, og tiden er det tidspunktet da testen ble kjørt. For feil i koden har hver «Marker»-node attributtene linjenummer, posisjon på linja, kategori, kategorinavn, feilmelding og prioritet. Feilmeldingen beskriver feilen som ble funnet og prioriteten beskriver hvor alvorlig feilen er. Linjenummer og posisjon på linja forteller hvor i koden feilen ble funnet, mens kategori og kategorinavn beskriver hvilken type feil det var.

Visning av data fra grafdatabasen mot visning av data fra Git

Visningen av data i systemet skjer gjennom å hente dataene fra grafdatabasen. Her fungerer grafdatabasen som et mellomsteg der de dataene som er relevante for visning lagres. Grafdatabasen er del av projeksjonen som gir en visning av data. Men hva hvis vi ønsker å i stedet lage en visning direkte fra rådataene i Git, uten grafdatabasen som mellomsteg? Hvor effektiv vil en slik løsning være mot å bruke grafdatabasen til å lage visning, slik det allerede er i systemet?

Som nevnt i forrige kapittel er data om tester som er kjørt og data om feil i koden lagret i hver sin egne filer i Git innunder prosjektmappene. De ligger lagret i filene «tests.json» og «markers.json». Ved å sjekke ut forskjellige versjoner av disse filene kan man se på endringene i tilstanden til de. Dette kan brukes til å se utviklingen av hvor mange tester som er passert og hvor mange feil brukeren har i koden. Men dette innebærer å måtte sjekke ut filene for hver tilstand og gå igjennom filene for å hente ut de dataene som trengs.

Denne prosessen er allerede gjort for dataene som er lagret i grafdatabasen. I grafdatabasen er dataene om tester og feil i koden relatert til tilstander som en fil har hatt. Her kan man se alle tilstandene med tidspunkt for når en fil hadde en tilstand, og man kan se hvilke tester eller feil i kode som hører til hver tilstand. Dette er data som må prosesseres ut fra Git i flere steg der man sjekker ut hver versjon av filene.

Grafdatabasen inneholder også andre data som kan være aktuelle. Hver «FileState»-node i grafdatabasen inneholder et nummer for antall linjer med kode som filen har i den representerte tilstanden. Dette nummeret kan man ikke finne direkte i Git. For å komme fram til nummeret i Git må man først sjekke ut filen, gå gjennom den og telle antall linjer. I en slik prosess må man ta hensyn til blant annet tomme linjer og krøllparenteser som står på egen linje, og det kan fort bli komplekst. Hvis man ikke lagrer unna nummeret for antall linjer noe sted, må man prosessere det ut på nytt ved senere bruk, noe som kan skape mye unødvendig arbeid.

I tillegg til de dataene som nodene inneholder, kan relasjonene også gi nyttig informasjon. Ved å se på relasjonene kan man enkelt finne alle testene som er knyttet til en fil. Denne type data er vanskeligere å hente ut fra Git da den eneste måten man kan finne ut hvilken fil en test tilhører er ved å se på filstiene som ligger lagret i filen «tests.json» i Git. Hver fil har også en relasjon til en «Category»-node i grafdatabasen som forteller hvilken øving en fil tilhører. Siden flere filer kan høre til samme øving, kan man i grafdatabasen se ale filene som hører til en øving, også på tvers av brukere. Disse dataene ligger ikke direkte i Git, og må prosesseres ut.

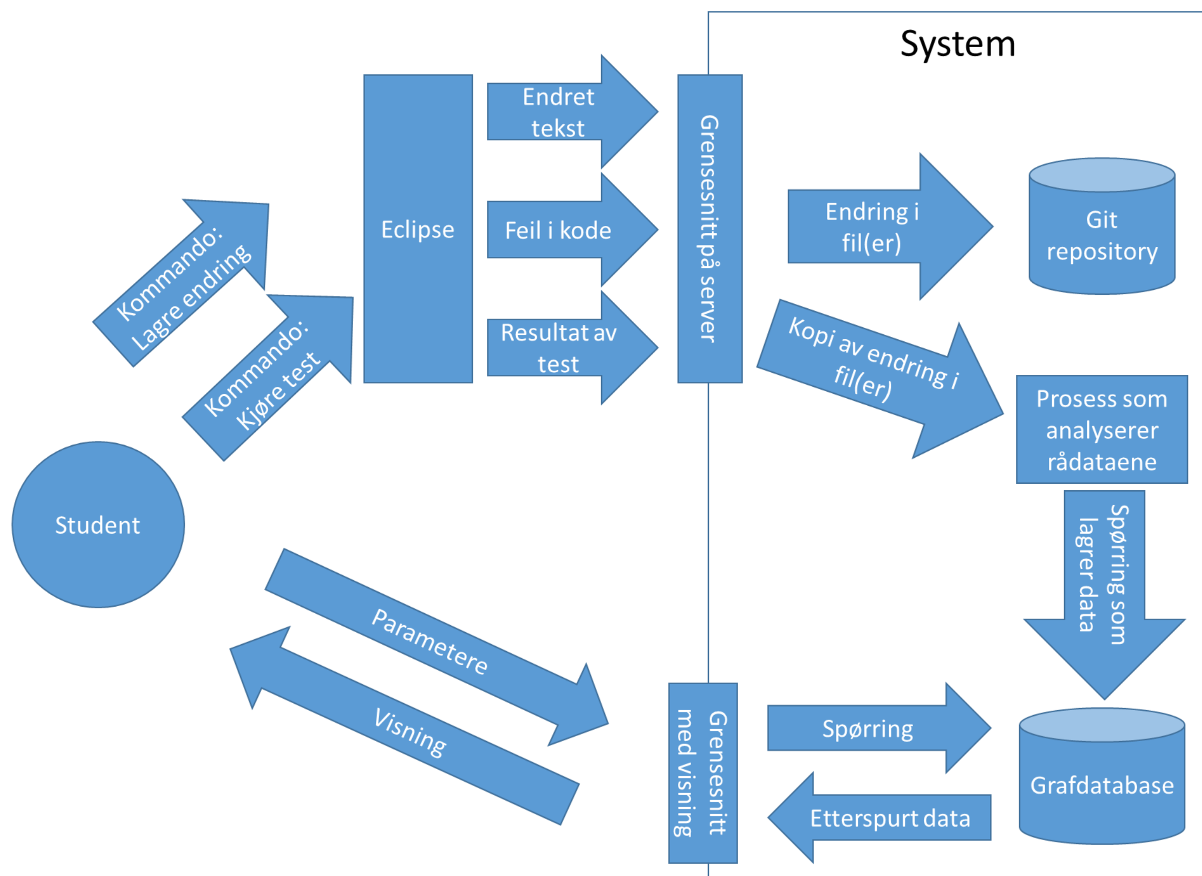
La oss se på et eksempel hvor du ønsker å se data om antall tester passert for en spesifikk fil fra en spesifikk øving. For å finne disse dataene i Git må du finne filen «tests.json» som inneholder data om tester som er kjørt på filer i prosjektmappen. Deretter må du gå igjennom denne filen for å finne eventuelle tester som tilhører filen du vil ha testresultat for. Ønsker du å finne alle tester passert over tid må du sjekke ut forskjellige versjoner av «tests.json» for å se endringene i testresultatet, og hver av disse versjonene må prosesseres.

I grafdatabasen er all prosesseringen som er beskrevet i avsnittene over allerede gjort. Visningene som lages fra dataene i grafdatabasen bruker spørringer mot grafdatabasen for å hente ut dataene. Dersom visningene baserer seg på data som hentes rett fra rådataene Git ville det krevd mye ekstra prosessering som vi slipper unna med å gjøre kun en gang når vi lagrer dataene i grafdatabasen. Vi ser at det er mer effektivt å bruke grafdatabasen som et mellomsteg fordi det krever mindre arbeid.

Men hva hvis skulle ønske å vise relevante data som ikke finnes i grafdatabasen. Det kan for eksempel være data som beskriver mål på kodekvalitet, og som er data som kan analyseres fra rådataene i Git. Da har vi et valg hvor vi kan velge å hente ut å vise dataene «ved siden av» grafdatabasen, eller vi kan utvide grafdatabasen med disse dataene. Ved å utvide grafdatabasen vil man kunne representere disse dataene med noder, relasjoner og attributter, men det er ikke sikkert at dette er den beste representasjonen av dataene. Da kan det være bedre å hente dataene rett fra Git i stedet.

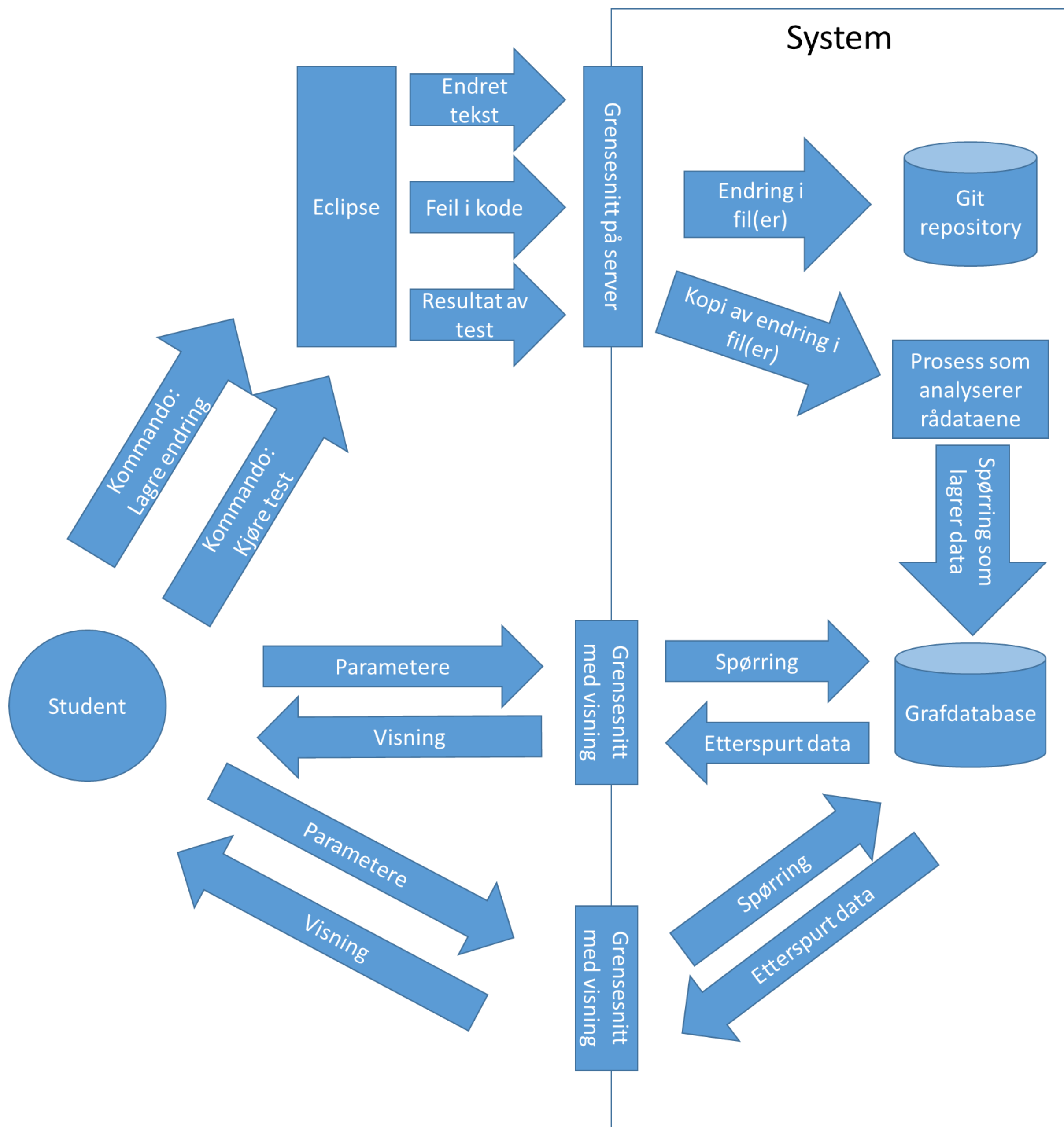
Effektivisering av analysen av rådata i Git

Slik som systemet er nå hentes dataene som er lagret i grafdatabasen fra Git repositoriene før de analyseres og lagres i grafdatabasen. I denne prosessen er man nødt til å hente ut dataene fra Git ved å sjekke et versjoner av de. Vi får da analysering av data i ettertid av at de er lagret i Git, og dette blir en tyngre jobb enn om de hadde blitt analysert før de ble lagret i Git, da man slipper unna å måtte hente de ut fra Git. Her er det mulig å effektivisere systemet ved å sende en kopi av dataene som lagres i Git videre til prosessen som analyserer de. Da slipper denne prosessen å i ettertid hente ut dataene fra Git, og grafdatabasen vil alltid være oppdatert med de nyeste dataene. Denne løsningen er vist i figuren under.



Lage nye visninger fra data i grafdatabasen

I grafdatabasen brukes språket «Cypher Query Language» for å lese, skrive, oppdatere og slette data i databasen. Spørringene konstrueres slik at de likner selve grafen i grafdatabasen, med både noder og relasjoner som begge kan ha attributter. Ved at spørringene konstrueres likt som grafen, blir de relativt enkle å konstruere når man vet hvordan grafen i grafdatabasen ser ut. I tillegg vil mange spørringer likne en del på hverandre i struktur, slik at man har mulighet til å gjenbruke spørringer ved å kun endre noen av parameterne. Dette gjør at det er ganske effektivt og fleksibelt å lage nye spørringer mot grafdatabasen, og siden visningene i systemet får dataene sine gjennom spørringer, gjør dette at det også er ganske fleksibelt å lage nye visninger. I figuren under ser vi hvordan en ny visning fra grafdatabasen vil se ut i systemet. Her har vi også tatt med endringen beskrevet i forrige avsnitt der det sendes kopi av endring i filer til prosessen som analyserer i stedet for at dataene hentes ut fra Git.



Figur 19: Flere visninger i systemet

Visningene blir som projeksjoner med hvert sitt grensesnitt, men som deler samme datalager. For å se hvordan visningen kan implementeres med kode er det laget et lite program i Java der grafdatabasen kjører i en integrert løsning. Koden til dette programmet ligger som vedlegg i denne rapporten. I koden kan vi se at grensesnittet er en metode som tar inn parametere fra klienten og returnerer data i tabellform. Spørringen som sendes til grafdatabasen konstrueres ved at en tekststreng som inneholder «skjelettet» til spørringen blir fylt inn med de parameterne som klienten sender inn.

Kapittel 5: Konklusjon

I dette kapitlet oppsummeres resultatene fra de to forrige kapitlene, og arbeidet i denne rapporten evalueres. Kom vi fram til det vi beskrev i introduksjonen, og nådde vi de målene vi hadde satt oss? Til slutt ser vi også på fremtidig arbeid som kan bygge på arbeidet gjort i denne oppgaven.

Oppsummering

I Kapittel 3 så vi at systemet til Stein følger CQRS med Event Sourcing der Git repositoriene fungerer som hendelseslogg i systemet og grafdatabasen fungerer som projeksjon av dataene i hendelsesloggen. Vi så at Git repositoriene fungerer akkurat som i Event Sourcing ved at det er endringene som lagres. Grafdatabasen blir datalageret i en projeksjon og inneholder metadata som er analysert fra rådataene i Git. For å lage visninger fra grafdatabasen i projeksjonen henter man ut data fra grafdatabasen med spørringer. Vi så at språket for spørringer mot grafdatabasen gjør det fleksibelt å lage nye spørringer, og dermed også fleksibelt å lage nye visninger.

Dersom man ønsker å vise annen data enn det som ligger i grafdatabasen, må man hente ut disse dataene fra hendelsesloggen i stedet. Vi så at å lage en visning av data fra hendelsesloggen var mye mer tungvint enn å lage en visning av data fra grafdatabasen, da man må gjøre en prosessering av hendelsene først. Dette gjøres kun en gang før man lagrer dataene i grafdatabasen, men må gjøres flere ganger dersom man henter data til visningen rett fra Git. Grafdatabasen fungerer derfor godt som et mellomlager mellom rådataene og visningen. Det trenger heller ikke være en grafdatabase som brukes. Den kunne vært byttet ut med en annen relasjonell database som ville fungert på samme måte.

For å legge inn data i grafdatabasen, blir data hentet ut av Git for så å analyseres før de legges inn i grafdatabasen. Her så vi at en effektivisering ville være å sende en kopi av dataene som sendes til Git rett til prosessen som analyserer dataene før de legges inn i grafdatabasen. Da slipper man steget å hente dataene ut fra Git, og man vil få de nye data inn i grafdatabasen med en gang.

Diskusjon og evaluering

I Introduksjon beskrev vi at vi ønsket å se hvordan Event Sourcing kan brukes i systemet til Stein. Vi ser at systemet allerede ligner på et system med Event Sourcing og CQRS. Hvis vi skulle ha fulgt en ren tankegang om bare Event Sourcing ville visningen av data basert seg på hendelsene i hendelsesloggen, og som vi har sett ville det vært mye mer tungvint enn slik systemet er med grafdatabasen. Vi ønsket å se hvor fleksibelt det var å lage nye visninger. Ved a ha grafdatabasen ser vi at det er fleksibelt å lage nye spørringer og dermed også nye visninger.

Metoden vi har brukt baserer seg på å først studere artikler om Event Sourcing som befinner seg på nett. Flere av artiklene er skrevet av kjente personligheter innen programmering. Basert på det vi har lært gjennom disse artiklene har vi skrevet tekst og laget figurer som beskriver Event Sourcing, og det er dette som har dannet grunnlaget for beskrivelsen vi har gjort av systemet til studenten og alternative løsninger i systemet. Totalt sett gir det oss grunn til å tro på de resultatene vi har kommet fram til, da artiklene kommer fra pålitelige kilder.

Fremtidig arbeid

Fremtidig arbeid som kan gjøres i forhold til denne oppgaven kan være å lage nye projeksjoner gjennom å lage nye spørringer som beskrevet i Kapittel 4. Det kan også være ønskelig å lage projeksjoner av andre data enn de som befinner seg i databasen. Ved å enten utvide den eksisterende grafdatabasen eller lage en ny database med nye data, kan man analysere dataene som sendes til Git repositoriene for nye data som man lagrer som et mellomsteg. Ut fra disse kan man da lage projeksjoner av nye data.

Koden som er laget for å spørre etter data i grafdatabasen returner dataene i tabellform. En visning for dataene på tabellform er ikke like lett å få oversikt over, og her vil det kunne være aktuelt å lage kode som er mer rettet mot en bedre visning av dataene. For eksempel kan dette være kode skrevet i JavaScript som får dataene i tabellform fra koden skrevet i denne oppgaven, og viser de på en nettside i et diagram.

Referanser

1. *CQRS and Event Sourcing* (2010). Tilgjengelig fra: https://cqrs.wordpress.com/documents/cqrs-and-event-sourcing-synergy/ (Hentet: 24. mars 2015)

2. *Event Sourcing Basics* (2015). Tilgjengelig fra: http://docs.geteventstore.com/introduction/event-sourcing-basics/ (Hentet: 12. mars 2015)

3. Fowler, Martin (2005) *Event Sourcing*. Tilgjengelig fra: http://martinfowler.com/eaaDev/EventSourcing.html (Hentet: 01. februar 2015)

4. Fowler, Martin (2011) *MemoryImage*. Tilgjengelig fra: http://martinfowler.com/bliki/MemoryImage.html (Hentet: 01. februar 2015)

5. Rozendaal, Erik (2012) *Simple Event Sourcing – introduction (part 1)*. Tilgjengelig fra: http://blog.zilverline.com/2012/07/04/simple-event-sourcing-introduction-part-1/ (Hentet: 01. februar 2015)

6. Stenberg, Jan (2014) *The Basics of Event Sourcing and Some CQRS*. Tilgjengelig fra: http://www.infoq.com/news/2014/09/greg-young-event-sourcing (Hentet: 01. februar 2015)

7. Sørhus, S.K. (2015) *Applying Learning Analytics in the course TDT4100 at NTNU*. Siv.ing. avhandling. IDI, NTNU

8. Young, Greg (2014) *CQRS and Event Sourcing*. Tilgjengelig fra: https://www.youtube.com/watch?v=JHGkaShoyNs (Hentet: 23. mars 2015)

Vedlegg A

```
package prototype;

import java.util.HashMap;
import java.util.Map;

import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Result;
import org.neo4j.graphdb.Transaction;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;

public class View {

    private String DB_PATH = "C:/neo4j/graph.db/";
    private GraphDatabaseService graphDb;

    public View() {

        graphDb = new
GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);
        registerShutdownHook(graphDb);
    }

    public Map<String, Object> getTests(String clientId,
                                        Boolean testPassed,
                                        String category,
                                        String startTime,
                                        String endTime) {

        Map<String, Object> results =
            new HashMap<String, Object>();

        try (Transaction tx = graphDb.beginTx()) {
            String query = getQuery(clientId,
                                    testPassed,
                                    category,
                                    startTime,
                                    endTime);

            try (Result result = graphDb.execute(query)) {
                while (result.hasNext()) {
                    results.putAll(result.next());
                }
            }

            tx.success();
        }

        return results;
```

```
    }

    private String getQuery(String clientId,
                            Boolean testPassed,
                            String category,
                            String startTime,
                            String endTime) {

        String query = "match (u:User)-[:HAS_REPO]-(r:Repo)-
[:HAS_FILE]-(f:File)-[:HAS_FILE_STATE]-(fs:FileState)-
[:HAS_TEST]-(t:Test), "
                + "(f)-[:IS_IN_CATEGORY]-(c:Category) "
                + "where (u.clientId = \"%s\") "
                + "and (t.result = \"%s\") "
                + "and (c.name = \"%s\") "
                + "and (fs.time >= %s) "
                + "and (fs.time <= %s) "
                + "return f, fs, count(t) as count;";
        if (testPassed == null) {
            return String.format(query, clientId,
                "OK\" or t.result = \"Failure\" or t.result =
\"Error",
                category, startTime, endTime);
        } else if (!testPassed) {
            return String.format(query, clientId,
                "Failure\" or t.result = \"Error",
                category, startTime, endTime);
        } else {
            return String.format(query, clientId,
                "OK", category, startTime, endTime);
        }
    }

    private void registerShutdownHook(
        final GraphDatabaseService graphDb )
    {
        Runtime.getRuntime().addShutdownHook( new Thread()
        {
            @Override
            public void run()
            {
                graphDb.shutdown();
            }
        } );
    }

    public void shutdownDb() {
        graphDb.shutdown();
    }

}
```