Engineer

Event sourcing

å lagre endringer som hendelser og utlede tilstanden av dem

Hva det er

Event sourcing er en måte å lagre data på der beretningen om hva som har skjedd, er selve dataene. De fleste systemer har den gjeldende tilstanden til en ordre eller en konto liggende i en rad og overskriver den ved hver endring. Et system bygd med event sourcing lagrer i stedet endringene: hver av dem legges til i en logg som en hendelse, og den gjeldende tilstanden regnes ut fra loggen når den trengs. Martin Fowler, som beskrev mønsteret i 2005, oppsummerer det i én linje: «Capture all changes to an application state as a sequence of events.»

En hendelse beskriver noe som allerede har skjedd. Greg Youngs CQRS Documents krever at hver hendelse navngis med et verb i preteritum, for eksempel `CustomerRelocated` eller `CargoShipped`, fordi navnet er en del av språket virksomheten selv bruker om sitt eget arbeid. Det som har skjedd, kan ikke gjøres om, så hendelser endres og slettes aldri. Young sammenligner loggen med en regnskapsførers hovedbok: en feil postering viskes ikke ut, den rettes med en ny, og den samme regelen gjelder for sletting. Legger en kunde to par sokker i handlekurven og tar dem ut igjen, får loggen en hendelse om at de ble fjernet. Young kaller en slik postering en motpostering (reversal transaction): kurven ender som om sokkene aldri var lagt i, mens loggen fortsatt viser at de var der.

Hendelsene ligger i strømmer, én for hver ting hvis tilstand de beskriver. Young bygger på domenedrevet design og kaller tingen et aggregat. Hver hendelse bærer strømmens id og et versjonsnummer som er unikt og fortløpende innenfor strømmen, slik at rekkefølgen aldri er i tvil. For å laste inn en ordre leser systemet ordrens hendelser etter versjon og bruker dem én om gangen. Young påpeker hvor lite selve lageret trenger å kunne: i sin enkleste form har det to operasjoner, nemlig å lese alle hendelsene i én strøm i rekkefølge og å legge nye hendelser til den.

Dermed er loggen det autoritative registeret. Fowler skriver at både loggen og den gjeldende tilstanden kan være den offisielle registreringen. Med event sourcing er det loggen, og alt annet, en tabell med gjeldende saldoer, en søkeindeks, en rapport, er kopier avledet av den som kan kastes og bygges på nytt.

Bruk begrepet i en beslutning

Bruk «Hva er event sourcing?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Delen brukerne aldri ser: hvor dataene ligger, hvilke regler som gjelder, og hvordan andre systemer får lov til å snakke med den.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Designet og driftet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tvers av tre databasemotorer

Designet og driftet 6 ETL/ELT-pipelines Ta kontakt →

Hvorfor det er viktig

Fowler nevner tre ting en komplett logg gjør mulig. Full gjenoppbygging: hele applikasjonens tilstand kan forkastes og gjenskapes ved å spille av alle hendelser på en tom applikasjon. Tidsspørringer: tilstanden på et hvilket som helst tidspunkt er resultatet av å spille av hendelsene fram til det tidspunktet. Og ny avspilling: viser en hendelse seg å være feil, kan systemet rulle tilbake den og alle senere hendelser og spille av den rettede rekkefølgen. Den samme loggen er dessuten et revisjonsspor som ingen måtte huske å skrive, og Fowler trekker fram brukerstøtte og feilsøking som opplagte bruksområder: brukerstøtten kan se nøyaktig hva en bruker gjorde, og en utvikler kan spille av hendelsene fra produksjon i et testmiljø.

Young viser gevinsten med to team hos en stor nettbutikk. Begge får i oppgave å lage en rapport over varer kundene legger i handlekurven og tar ut igjen. Teamet som bare lagrer gjeldende tilstand, begynner å registrere slikt nå, og rapporten har data fra den dagen. Teamet som lagrer hendelser, skriver den samme rapportlogikken og kjører den over loggen fra begynnelsen, så den første rapporten strekker seg flere år tilbake. Et spørsmål ingen tenkte på da systemet ble bygd, kan likevel besvares, fordi systemet tok vare på hva som skjedde, og ikke bare hva det endte med.

Hendelser er også måten andre systemer får vite om en endring på. Young bemerker at mange systemer lagrer tilstanden sin og deretter sender en hendelse til en kø, noe som strengt tatt krever en tofasecommit mellom de to,

for går systemet ned midt imellom, er de uenige. Alternativet hans er å la hendelseslageret selv være køen: et løpenummer på hver hendelse lar en separat prosess sende hendelsene videre i rekkefølge, slik at det bare er én skriving. Og fordi lageret bare legger til, skalerer det lettere. Young påpeker at et lager som bare legger til, har langt færre låser å holde styr på enn et som oppdaterer på stedet, og at aggregatets id er den eneste nøkkelen et hendelseslager noen gang må partisjoneres etter.

Kostnadene er reelle, og Fowler nevner dem selv. Å pakke hver endring inn som en hendelse er en programmeringsstil som ikke alle er komfortable med, og mønsteret er svært vanskelig å innføre i ettertid i et system som ikke er bygd for det. Lageret svarer dessuten bare på én type spørsmål, nemlig «gi meg denne strømmen». Youngs eget eksempel er at et hendelseslager ikke kan fortelle hvilke brukere som heter Greg til fornavn. Slike spørsmål krever en separat lesemodell, bygd fra hendelsene, og derfor behandler Young event sourcing og CQRS, atskillelsen av modellene for skriving og lesing, som et par.

Slik ser det ut i praksis

Det minste hendelseslageret som fungerer, er én tabell i PostgreSQL, nær det oppsettet Young beskriver: en aggregat-id, hendelsens data og en versjon. Her holder den lagerbeholdningen for én vare som er til salgs. Hver hendelse bærer en endring i antallet enheter til salgs og ikke den nye summen, for den formen er ifølge Fowler den enkleste å rulle tilbake: «legg til ti» oppheves ved å trekke fra ti, mens «sett til 110» har mistet verdien den erstattet.

```
CREATE TABLE stock_events (
    item_id  text    NOT NULL,
    version  integer NOT NULL,
    event    text    NOT NULL,
    order_id text,
    units    integer NOT NULL,  -- endring i enheter til salgs
    PRIMARY KEY (item_id, version)
);

INSERT INTO stock_events VALUES
    ('item-881', 1, 'ItemStocked',   NULL,   100),
    ('item-881', 2, 'UnitsHeld',     'A-17',  -2),
    ('item-881', 3, 'HoldExpired',   'A-17',   2),
    ('item-881', 4, 'UnitsHeld',     'B-02',  -1),
    ('item-881', 5, 'HoldConfirmed', 'B-02',   0);

-- gjeldende tilstand: summer hele strømmen
SELECT sum(units) AS on_sale FROM stock_events
```

```
WHERE item_id = 'item-881';                    -- 99

-- tidsspørring: varen slik den sto etter versjon 2
SELECT sum(units) AS on_sale FROM stock_events
WHERE item_id = 'item-881' AND version <= 2;   -- 98
```

Primærnøkkelen gjør jobben Young gir en optimistisk samtidighetskontroll. To utsjekkinger som begge har lest strømmen ved versjon 5, og som begge vil reservere en enhet, prøver hver å legge til versjon 6:

```
INSERT INTO stock_events VALUES ('item-881', 6, 'UnitsHeld', 'C-40', -1);
INSERT INTO stock_events VALUES ('item-881', 6, 'UnitsHeld', 'D-11', -1);
-- den andre feiler: (item-881, 6) finnes allerede
```

PostgreSQL-dokumentasjonen beskriver en primærnøkkel over flere kolonner som et krav om at kombinasjonen er unik, og en innsetting som ville gjentatt den, gir en feil. Utsjekkingen som tapte, laster strømmen på nytt, ser etter om det fortsatt er en enhet til salgs, og prøver igjen som versjon 7. Ingen av dem har solgt en enhet som ikke fantes, og loggen viser begge beslutningene i den rekkefølgen de ble tatt.

Notatet Å selge den siste billetten nøyaktig én gang arbeider med det samme lagerproblemet og viser hvor denne formen kommer under press. Det første svaret holder tallet i én rad og endrer det med én betinget `UPDATE`, og notatet fraråder optimistisk låsing på en het teller, fordi en versjonskolonne gjør kampen om raden til nye forsøk. For den mest etterspurte varen legger det hver forespørsel på én Kafka-partisjon, som leses av én enkelt konsument som holder tallet i minnet og lagrer avgjørelsene sine i batcher, og det kaller dette LMAX-formen. Fowlers artikkel om LMAX-arkitekturen, som notatet viser til, beskriver event sourcing i sin reneste form: en handelsmotor helt uten database, der hele tilstanden kan utledes av hendelsene den mottar, og der en journalfører lagrer dem varig slik at de kan spilles av igjen. Utsjekkingen i notatet bruker hendelser av en annen sort. Hvert steg skriver en melding i en outbox-tabell, change data capture sender den videre, og tabellene er fortsatt det autoritative registeret. Der kunngjør hendelsene endringer; i event sourcing er hendelsene registeret.

Hvor det går galt

Ny avspilling gjentar sideeffektene. En handler som sender en ordrebekreftelse eller belaster et kort, gjør det igjen når loggen spilles av, og Fowler påpeker at eksterne systemer ikke kan skille en avspilling fra ekte behandling. En gjenoppbygd lesemodell kan da sende hver kunde fjorårets e-poster. Legg hvert eksternt system bak en gateway som vet når applikasjonen spiller av, og som dropper utgående kall i mellomtiden, slik at domenelogikken aldri trenger å vite det. Spørringer krever samme omhu: Fowlers eksempel er en valutaomregning, der en avspilt hendelse må bruke kursen fra sin egen dag og ikke dagens, så gatewayen bør lagre svarene den fikk, og levere dem igjen under en avspilling.

En travel strøm drukner i nye forsøk. Optimistisk samtidighet forutsetter at konflikter er sjeldne. Når mange skriver til den samme strømmen samtidig, betyr hver konflikt at strømmen må lastes på nytt, og ved en plutselig økning i trafikken blir de nye forsøkene selve belastningen, slik notatet over advarer om. Del opp strømmene slik at én sjelden har to skrivere samtidig: én strøm per ordre framfor én for en hel varegruppe. Der en enkelt strøm virkelig er het, gi den én skriver, som i LMAX-designet, så det ikke er noe igjen å kollidere om.

Lange strømmer lastes sakte. Å spille av tusenvis av hendelser for å laste ett objekt koster tid ved hver forespørsel, og Fowler nevner nettopp den langsomme gjenoppbyggingen som grunnen til å lagre en kopi av den gjeldende tilstanden. Youngs svar er det rullerende øyeblikksbildet (rolling snapshot): den serialiserte tilstanden til en strøm ved en kjent versjon, lagt i en egen tabell og skrevet i bakgrunnen av en separat prosess, slik at innlasting betyr det nyeste øyeblikksbildet pluss hendelsene etter versjonen dets. Hold øyeblikksbildene utenfor selve hendelsesloggen: Young beskriver hvordan prosessen som tar dem, på en travel strøm feiler samtidighetskontrollen sin igjen og igjen. Han anbefaler også å utvikle uten øyeblikksbilder, fordi de alltid kan legges til senere for å gjøre innlastingen raskere.

Lesemodellene henger etter. Hver lesemodell bygges fra hendelsene etter at de er skrevet, så den ligger bak loggen. Fowler advarer om at systemer som leser loggen, lett kommer i utakt med den, og en bruker som sender et skjema og deretter ser en liste uten endringen, sender det igjen. La hver lesemodell registrere den siste hendelsen den har brukt, slik at forsinkelsen kan måles og modellen kan bygges på nytt, og vis brukeren resultatet av vedkommendes egen kommando ut fra kommandoens svar framfor fra en lesemodell som kanskje ikke har tatt den igjen ennå.

Les mer

- Event Sourcing — Martin Fowlers beskrivelse av mønsteret: full gjenoppbygging, tidsspørringer, ny avspilling og gatewayer som holder avspillinger unna andre systemer.
- CQRS Documents by Greg Young — hendelser som fakta i preteritum, motpostering i stedet for sletting, et hendelseslager i to tabeller, rullerende øyeblikksbilder og lageret som kø.
- The LMAX Architecture — en handelsmotor i minnet der hele tilstanden utledes av journalførte inndatahendelser.
- PostgreSQL: Constraints — unikhets- og primærnøkkelbegrensninger over flere kolonner, databasens halvdel av en optimistisk samtidighetskontroll.

Ordliste

Velg ut fra deres faktiske rammer

Beskriv arbeidsflyten, eksisterende grensesnitt og brukerne som trenger endringen. Vi kan definere en testbar leveranse og en overlevering som gjør teamet i stand til å vedlikeholde løsningen.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er event sourcing?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

welcome@engineer.company +45 50 10 61 84 København, Danmark

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/event-sourcing/