Engineer

Event sourcing

at gemme ændringer som hændelser og udlede tilstanden af dem

Hvad det er

Event sourcing er en måde at gemme data på, hvor beretningen om, hvad der er sket, er selve dataene. De fleste systemer har den aktuelle tilstand af en ordre eller en konto liggende i en række og overskriver den ved hver ændring. Et system bygget med event sourcing gemmer i stedet ændringerne: hver af dem føjes til en log som en hændelse, og den aktuelle tilstand regnes ud fra loggen, når der er brug for den. Martin Fowler, der beskrev mønstret i 2005, sammenfatter det i én linje: »Capture all changes to an application state as a sequence of events.«

En hændelse beskriver noget, der allerede er sket. Greg Youngs CQRS Documents kræver, at hver hændelse navngives med et udsagnsord i datid, for eksempel `CustomerRelocated` eller `CargoShipped`, fordi navnet er en del af det sprog, forretningen selv taler om sit arbejde med. Det, der er sket, kan ikke gøres ugjort, så hændelser rettes og slettes aldrig. Young sammenligner loggen med en bogholders regnskabsbog: en forkert postering viskes ikke ud, den modposteres, og samme regel gælder for sletning. Lægger en kunde to par sokker i kurven og tager dem ud igen, får loggen en hændelse om, at de blev fjernet. Young kalder den slags en modpostering (reversal transaction): kurven ender, som om sokkerne aldrig var lagt i, mens loggen stadig viser, at de var der.

Hændelserne ligger i strømme, én for hver ting, hvis tilstand de beskriver. Young bygger på domænedrevet design og kalder tingen et aggregat. Hver hændelse bærer strømmens id og et versionsnummer, der er entydigt og fortløbende inden for strømmen, så rækkefølgen aldrig er til diskussion. For at indlæse en ordre læser systemet ordrens hændelser efter version og anvender dem én ad gangen. Young gør opmærksom på, hvor lidt selve lageret skal kunne: i sin enkleste form har det to operationer, nemlig at læse alle hændelser i én strøm i rækkefølge og at tilføje nye hændelser til den.

Dermed er loggen det autoritative register. Fowler skriver, at både loggen og den aktuelle tilstand kan være den officielle optegnelse. Med event sourcing er det loggen, og alt andet, en tabel med aktuelle saldi, et søgeindeks, en rapport, er kopier afledt af den, som kan smides væk og bygges forfra.

Brug begrebet i en beslutning

Brug »Hvad er event sourcing?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Den del, brugerne aldrig ser: hvor data ligger, hvilke regler der gælder, og hvordan andre systemer må tale med den.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har designet og drevet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tre databasemotorer

Har designet og drevet 6 ETL/ELT-pipelines Kontakt os →

Hvorfor det betyder noget

Fowler nævner tre ting, som en komplet log gør mulige. Fuld genopbygning: hele applikationens tilstand kan kasseres og genskabes ved at afspille alle hændelser på en tom applikation. Tidsforespørgsler: tilstanden på et vilkårligt tidspunkt er resultatet af at afspille hændelserne frem til dette tidspunkt. Og genafspilning: viser en hændelse sig at være forkert, kan systemet rulle den og alle senere hændelser tilbage og afspille den rettede rækkefølge. Den samme log er desuden et revisionsspor, som ingen skulle huske at skrive, og Fowler peger på support og fejlfinding som oplagte anvendelser: supporten kan se præcis, hvad en bruger gjorde, og en udvikler kan afspille hændelserne fra produktion i et testmiljø.

Young viser gevinsten med to teams hos en stor netbutik. Begge får til opgave at lave en rapport over varer, som kunderne lægger i kurven og tager ud igen. Teamet, der kun gemmer den aktuelle tilstand, begynder at registrere den slags nu, og rapporten har data fra den dag. Teamet, der gemmer hændelser, skriver den samme rapportlogik og kører den over loggen fra begyndelsen, så den første rapport rækker flere år tilbage. Et spørgsmål, ingen tænkte på, da systemet blev bygget, kan stadig besvares, fordi systemet har gemt, hvad der skete, og ikke kun, hvad det endte med.

Hændelser er også den måde, andre systemer hører om en ændring på. Young bemærker, at mange systemer gemmer deres tilstand og derefter sender en hændelse til en kø, hvilket strengt taget kræver en tofasecommit mellem de to, for går systemet ned midt imellem, er de uenige. Hans alternativ er at lade

hændelseslageret selv være køen: et løbenummer på hver hændelse lader en separat proces sende hændelserne videre i rækkefølge, så der kun er én skrivning. Og fordi lageret kun tilføjer, skalerer det lettere. Young påpeger, at et lager, der kun tilføjer, har langt færre låse at holde styr på end et, der opdaterer på stedet, og at aggregatets id er den eneste nøgle, et hændelseslager nogensinde skal partitioneres efter.

Omkostningerne er reelle, og Fowler nævner dem selv. At pakke hver ændring ind som en hændelse er en programmeringsstil, som ikke alle er trygge ved, og mønstret er meget svært at indføre bagefter i et system, der ikke er bygget til det. Lageret svarer desuden kun på én slags spørgsmål, nemlig »giv mig denne strøm«. Youngs eget eksempel er, at et hændelseslager ikke kan fortælle, hvilke brugere der hedder Greg til fornavn. Den slags spørgsmål kræver en separat læsemodel, bygget ud fra hændelserne, og derfor behandler Young event sourcing og CQRS, adskillelsen af modellerne til skrivning og læsning, som et par.

Sådan ser det ud i praksis

Det mindste hændelseslager, der virker, er én tabel i PostgreSQL, tæt på det layout, Young beskriver: et aggregat-id, hændelsens data og en version. Her holder den lagerbeholdningen for én vare til salg. Hver hændelse bærer en ændring i antallet af enheder til salg og ikke det nye samlede antal, for den form er ifølge Fowler den nemmeste at rulle tilbage: »læg ti til« fortrydes ved at trække ti fra, mens »sæt til 110« har mistet den værdi, den erstattede.

```
CREATE TABLE stock_events (
    item_id  text    NOT NULL,
    version  integer NOT NULL,
    event    text    NOT NULL,
    order_id text,
    units    integer NOT NULL,  -- ændring i enheder til salg
    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);

-- aktuel tilstand: læg hele strømmen sammen
SELECT sum(units) AS on_sale FROM stock_events
WHERE item_id = 'item-881';                    -- 99
```

```
-- tidsforespørgsel: varen, som den stod efter version 2
SELECT sum(units) AS on_sale FROM stock_events
WHERE item_id = 'item-881' AND version <= 2;   -- 98
```

Primærnøglen klarer det, Young kalder et optimistisk samtidighedstjek. To checkout-forløb, der begge har læst strømmen ved version 5 og begge vil reservere en enhed, forsøger hver at tilføje version 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 anden fejler: (item-881, 6) findes allerede
```

PostgreSQL-dokumentationen beskriver en primærnøgle over flere kolonner som et krav om, at kombinationen er entydig, og en indsættelse, der ville gentage den, giver en fejl. Det forløb, der tabte, indlæser strømmen igen, ser efter, om der stadig er en enhed til salg, og prøver igen som version 7. Ingen af dem har solgt en enhed, der ikke fandtes, og loggen viser begge beslutninger i den rækkefølge, de blev truffet.

Noten At sælge den sidste billet præcis én gang arbejder med det samme lagerproblem og viser, hvor denne form kommer under pres. Dens første svar holder tallet i én række og ændrer det med én betinget `UPDATE`, og den fraråder optimistisk låsning på en varm tæller, fordi en versionskolonne gør kamp om rækken til gentagne forsøg. Til den mest efterspurgte vare lægger den hver forespørgsel på én Kafka-partition, som læses af en enkelt forbruger, der holder tallet i hukommelsen og gemmer sine afgørelser i batches, og den kalder det LMAX-formen. Fowlers artikel om LMAX-arkitekturen, som noten henviser til, beskriver event sourcing i sin reneste form: en handelsmotor helt uden database, hvis samlede tilstand kan udledes af de hændelser, den modtager, og som en journalfører gemmer varigt, så de kan afspilles igen. Notens checkout bruger hændelser af en anden slags. Hvert trin skriver en besked i en outbox-tabel, change data capture sender den videre, og tabellerne er fortsat det autoritative register. Her annoncerer hændelserne ændringer; i event sourcing er hændelserne registret.

Hvor det går galt

Genafspilning gentager sideeffekterne. En handler, der sender en ordrebekræftelse eller trækker et beløb på et kort, gør det igen, når loggen afspilles, og Fowler påpeger, at eksterne systemer ikke kan kende forskel på en genafspilning og ægte behandling. En genopbygget læsemodel kan så sende hver kunde sidste års mails. Læg hvert eksternt system bag en gateway, der ved, hvornår applikationen afspiller, og som dropper udgående kald imens, så domænelogikken aldrig behøver at vide det. Forespørgsler kræver samme omhu: Fowlers eksempel er en valutaomregning, hvor en genafspillet hændelse skal bruge kursen fra sin egen dag og ikke dagens, så gatewayen bør gemme de svar, den fik, og levere dem igen under en genafspilning.

En travl strøm drukner i nye forsøg. Optimistisk samtidighed forudsætter, at konflikter er sjældne. Når mange skriver til den samme strøm samtidig, betyder hver konflikt, at strømmen skal indlæses igen, og ved en pludselig trafikstigning bliver de gentagne forsøg selve belastningen, som noten ovenfor advarer om. Skær strømmene, så én sjældent har to skrivere på samme tid: én strøm pr. ordre frem for én for en hel varegruppe. Hvor en enkelt strøm reelt er varm, så giv den én skriver, som i LMAX-designet, så der ikke er noget tilbage at kollidere om.

Lange strømme indlæses langsomt. At afspille tusindvis af hændelser for at indlæse ét objekt koster tid ved hver forespørgsel, og Fowler nævner netop den langsomme genopbygning som grunden til at gemme en kopi af den aktuelle tilstand. Youngs svar er det rullende øjebliksbillede (rolling snapshot): den serialiserede tilstand af en strøm ved en kendt version, lagt i sin egen tabel og skrevet i baggrunden af en separat proces, så indlæsning betyder det nyeste øjebliksbillede plus hændelserne efter dets version. Hold øjebliksbillederne ude af selve hændelsesloggen: Young beskriver, hvordan processen, der tager dem, på en travl strøm fejler sit samtidighedstjek igen og igen. Han anbefaler også at udvikle uden øjebliksbilleder, fordi de altid kan tilføjes senere for at gøre indlæsningen hurtigere.

Læsemodellerne halter efter. Hver læsemodel bygges ud fra hændelserne, efter de er skrevet, så den ligger bag loggen. Fowler advarer om, at systemer, der læser loggen, let kommer ud af trit med den, og en bruger, der sender en formular og derefter ser en liste uden ændringen, sender den igen. Lad hver læsemodel registrere den seneste hændelse, den har anvendt, så forsinkelsen kan måles og modellen kan bygges forfra, og vis brugeren resultatet af vedkommendes egen kommando ud fra kommandoens svar frem for fra en læsemodel, der kan være et skridt bagud.

Læs mere

- Event Sourcing — Martin Fowlers beskrivelse af mønstret: fuld genopbygning, tidsforespørgsler, genafspilning og gateways, der holder genafspilninger væk fra andre systemer.
- CQRS Documents by Greg Young — hændelser som fakta i datid, modpostering i stedet for sletning, et hændelseslager i to tabeller, rullende øjebliksbilleder og lageret som kø.
- The LMAX Architecture — en handelsmotor i hukommelsen, hvis samlede tilstand udledes af journalførte inputhændelser.
- PostgreSQL: Constraints — entydigheds- og primærnøglebegrænsninger over flere kolonner, databasens halvdel af et optimistisk samtidighedstjek.

Ordliste

Vælg ud fra jeres faktiske vilkår

Beskriv arbejdsgangen, de eksisterende grænseflader og de brugere, der har brug for ændringen. Vi kan definere en testbar leverance og en overdragelse, der gør teamet i stand til at vedligeholde løsningen.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er event sourcing?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

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

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/event-sourcing/