Engineer

Event sourcing

att spara ändringar som händelser och härleda tillståndet

Vad det är

Event sourcing är ett sätt att lagra data där berättelsen om vad som har hänt är själva datan. De flesta system har det aktuella tillståndet för en order eller ett konto i en rad och skriver över det vid varje ändring. Ett system byggt med event sourcing sparar i stället ändringarna: var och en läggs till i en logg som en händelse, och det aktuella tillståndet räknas fram ur loggen när det behövs. Martin Fowler, som beskrev mönstret 2005, sammanfattar det på en rad: ”Capture all changes to an application state as a sequence of events.”

En händelse beskriver något som redan har hänt. Greg Youngs CQRS Documents kräver att varje händelse namnges med ett verb i preteritum, till exempel `CustomerRelocated` eller `CargoShipped`, eftersom namnet är en del av det språk verksamheten själv använder om sitt arbete. Det som har hänt går inte att göra ogjort, så händelser ändras och raderas aldrig. Young jämför loggen med en bokförares huvudbok: en felaktig post suddas inte ut utan rättas med en ny, och samma regel gäller för radering. Lägger en kund två par strumpor i varukorgen och tar ut dem igen, får loggen en händelse om att de togs bort. Young kallar en sådan post en reversering (reversal transaction): korgen blir som om strumporna aldrig hade lagts i, medan loggen fortfarande visar att de fanns där.

Händelserna ligger i strömmar, en för varje sak vars tillstånd de beskriver. Young bygger på domändriven design och kallar saken ett aggregat. Varje händelse bär strömmens id och ett versionsnummer som är unikt och löpande inom strömmen, så att ordningen aldrig är oklar. För att läsa in en order läser systemet orderns händelser efter version och tillämpar dem en i taget. Young påpekar hur lite själva lagret behöver kunna: i sin enklaste form har det två operationer, att läsa alla händelser i en ström i ordning och att lägga till nya händelser i den.

Därmed är loggen det auktoritativa registret. Fowler skriver att både loggen och det aktuella tillståndet kan vara den officiella uppteckningen. Med event sourcing är det loggen, och allt annat, en tabell med aktuella saldon, ett sökindex, en rapport, är kopior härledda ur den som kan kastas och byggas upp igen.

Använd begreppet i ett beslut

Använd ”Vad är event sourcing?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Den del användarna aldrig ser: där data lagras, vilka reglerna är och hur andra system får kommunicera med den.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Utformade och drev 6 ETL/ELT-pipelines — 6 ETL-pipelines över tre databasmotorer

Utformade och drev 6 ETL/ELT-pipelines Hör av er →

Varför det spelar roll

Fowler nämner tre saker som en fullständig logg gör möjliga. Fullständig återuppbyggnad: hela applikationens tillstånd kan kastas och återskapas genom att alla händelser spelas upp på en tom applikation. Tidsfrågor: tillståndet vid vilken tidpunkt som helst är resultatet av att spela upp händelserna fram till den tidpunkten. Och omspelning: visar sig en händelse vara fel kan systemet backa den och alla senare händelser och sedan spela upp den rättade ordningsföljden. Samma logg är dessutom ett revisionsspår som ingen behövde komma ihåg att skriva, och Fowler lyfter fram support och felsökning som självklara användningar: supporten kan se exakt vad en användare gjorde, och en utvecklare kan spela upp händelserna från produktion i en testmiljö.

Young visar vinsten med två team på en stor näthandel. Båda får i uppdrag att ta fram en rapport över varor som kunderna lägger i varukorgen och tar ut igen. Teamet som bara sparar aktuellt tillstånd börjar registrera sådant nu, och rapporten har data från den dagen. Teamet som sparar händelser skriver samma rapportlogik och kör den över loggen från början, så den första rapporten sträcker sig flera år bakåt. En fråga som ingen tänkte på när systemet byggdes går ändå att besvara, eftersom systemet sparade vad som hände och inte bara vad det slutade med.

Händelser är också sättet som andra system får veta om en ändring på. Young konstaterar att många system sparar sitt tillstånd och sedan skickar en händelse till en kö, vilket strängt taget kräver en tvåfascommit mellan de två, för går

systemet ner mittemellan är de oense. Hans alternativ är att låta händelselagret självt vara kön: ett löpnummer på varje händelse låter en separat process skicka vidare händelserna i ordning, så att det bara blir en skrivning. Och eftersom lagret bara lägger till skalar det lättare. Young påpekar att ett lager som bara lägger till har betydligt färre lås att hålla reda på än ett som uppdaterar på plats, och att aggregatets id är den enda nyckel ett händelselager någonsin behöver partitioneras efter.

Kostnaderna är verkliga, och Fowler nämner dem själv. Att paketera varje ändring som en händelse är en programmeringsstil som inte alla känner sig bekväma med, och mönstret är mycket svårt att införa i efterhand i ett system som inte är byggt för det. Lagret svarar dessutom bara på en sorts fråga, nämligen ”ge mig den här strömmen”. Youngs eget exempel är att ett händelselager inte kan tala om vilka användare som heter Greg i förnamn. Sådana frågor kräver en separat läsmodell, byggd ur händelserna, och därför behandlar Young event sourcing och CQRS, uppdelningen av modellerna för skrivning och läsning, som ett par.

Så ser det ut i praktiken

Det minsta händelselager som fungerar är en tabell i PostgreSQL, nära den uppställning Young beskriver: ett aggregat-id, händelsens data och en version. Här håller den lagersaldot för en vara som säljs. Varje händelse bär en ändring av antalet enheter till försäljning och inte den nya summan, eftersom den formen enligt Fowler är den enklaste att backa: ”lägg till tio” upphävs genom att dra ifrån tio, medan ”sätt till 110” har tappat värdet det ersatte.

```
CREATE TABLE stock_events (
    item_id  text    NOT NULL,
    version  integer NOT NULL,
    event    text    NOT NULL,
    order_id text,
    units    integer NOT NULL,  -- ändring av enheter till salu
    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);

-- aktuellt tillstånd: summera hela strömmen
SELECT sum(units) AS on_sale FROM stock_events
```

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

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

Primärnyckeln gör det jobb som Young ger en optimistisk samtidighetskontroll. Två kassaflöden som båda har läst strömmen vid version 5 och båda vill reservera en enhet försöker vart och ett lägga till 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 andra misslyckas: (item-881, 6) finns redan
```

PostgreSQL-dokumentationen beskriver en primärnyckel över flera kolumner som ett krav på att kombinationen är unik, och en insättning som skulle upprepa den ger ett fel. Kassaflödet som förlorade läser in strömmen igen, kontrollerar att det fortfarande finns en enhet till salu och försöker på nytt som version 7. Inget av dem har sålt en enhet som inte fanns, och loggen visar båda besluten i den ordning de fattades.

Anteckningen Att sälja den sista biljetten exakt en gång arbetar med samma lagerproblem och visar var den här formen blir trång. Dess första svar håller antalet i en rad och ändrar det med en enda villkorad `UPDATE`, och den avråder från optimistisk låsning på en het räknare, eftersom en versionskolumn gör kampen om raden till nya försök. För den mest efterfrågade varan lägger den varje förfrågan på en Kafka-partition, som läses av en enda konsument som håller antalet i minnet och sparar sina beslut i omgångar, och den kallar det LMAX-formen. Fowlers artikel om LMAX-arkitekturen, som anteckningen hänvisar till, beskriver event sourcing i dess renaste form: en handelsmotor helt utan databas, vars hela tillstånd kan härledas ur de händelser den tar emot och som en journalförare sparar beständigt så att de kan spelas upp igen. Anteckningens kassaflöde använder händelser av ett annat slag. Varje steg skriver ett meddelande i en outbox-tabell, change data capture skickar det vidare, och tabellerna förblir det auktoritativa registret. Där meddelar händelserna ändringar; i event sourcing är händelserna registret.

Var det går fel

Omspelning upprepar sidoeffekterna. En hanterare som skickar en orderbekräftelse eller drar pengar från ett kort gör det igen när loggen spelas upp, och Fowler påpekar att externa system inte kan skilja en omspelning från verklig behandling. En återuppbyggd läsmodell kan då skicka förra årets mejl till varje kund. Lägg varje externt system bakom en gateway som vet när applikationen spelar upp och som stoppar utgående anrop under tiden, så att domänlogiken aldrig behöver veta om det. Frågor kräver samma omsorg: Fowlers exempel är en valutaomräkning, där en omspelad händelse måste använda kursen från sin egen dag och inte dagens, så gatewayen bör spara svaren den fick och lämna dem igen under en omspelning.

En livlig ström drunknar i nya försök. Optimistisk samtidighet förutsätter att konflikter är sällsynta. När många skriver till samma ström samtidigt betyder varje konflikt att strömmen måste läsas in igen, och vid en plötslig trafikökning blir de nya försöken själva belastningen, precis som anteckningen ovan varnar för. Dela upp strömmarna så att en sällan har två skribenter samtidigt: en ström per order snarare än en för en hel varugrupp. Där en enskild ström verkligen är het, ge den en enda skribent, som i LMAX-designen, så att det inte finns något kvar att krocka om.

Långa strömmar läses in långsamt. Att spela upp tusentals händelser för att läsa in ett objekt kostar tid vid varje anrop, och Fowler nämner just den långsamma återuppbyggnaden som skälet att spara en kopia av det aktuella tillståndet. Youngs svar är den rullande ögonblicksbilden (rolling snapshot): det serialiserade tillståndet för en ström vid en känd version, lagt i en egen tabell och skrivet i bakgrunden av en separat process, så att inläsning betyder den senaste ögonblicksbilden plus händelserna efter dess version. Håll ögonblicksbilderna utanför själva händelseloggen: Young beskriver hur processen som tar dem misslyckas med sin samtidighetskontroll om och om igen på en livlig ström. Han rekommenderar också att bygga utan ögonblicksbilder, eftersom de alltid kan läggas till senare för att göra inläsningen snabbare.

Läsmodellerna släpar efter. Varje läsmodell byggs ur händelserna efter att de har skrivits, så den ligger efter loggen. Fowler varnar för att system som läser loggen lätt kommer i otakt med den, och en användare som skickar ett formulär och sedan ser en lista utan ändringen skickar det igen. Låt varje läsmodell registrera den senaste händelse den har tillämpat, så att eftersläpningen kan

mätas och modellen kan byggas om från början, och visa användaren resultatet av användarens eget kommando utifrån kommandots svar snarare än från en läsmodell som kanske inte har hunnit ikapp.

Läs mer

- Event Sourcing — Martin Fowlers beskrivning av mönstret: fullständig återuppbyggnad, tidsfrågor, omspelning och gatewayer som håller omspelningar borta från andra system.
- CQRS Documents by Greg Young — händelser som fakta i preteritum, reversering i stället för radering, ett händelselager i två tabeller, rullande ögonblicksbilder och lagret som kö.
- The LMAX Architecture — en handelsmotor i minnet vars hela tillstånd härleds ur journalförda indatahändelser.
- PostgreSQL: Constraints — unikhets- och primärnyckelvillkor över flera kolumner, databasens halva av en optimistisk samtidighetskontroll.



Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är event sourcing?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

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

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/event-sourcing/