Engineer

Event sourcing

état gardé comme journal ordonné des événements le modifiant

Ce que c’est

L’event sourcing est une manière de stocker les données où l’historique de ce qui s’est passé constitue les données elles-mêmes. La plupart des applications conservent l’état actuel d’une commande ou d’un compte dans une ligne, écrasée à chaque modification. Une application fondée sur l’event sourcing conserve les changements : chacun est ajouté à un journal comme événement, et l’état actuel est calculé à partir de ce journal chaque fois qu’on en a besoin. Martin Fowler, qui a décrit le modèle en 2005, le résume ainsi : « Enregistrer toutes les modifications de l’état d’une application sous forme d’une séquence d’événements. »

Un événement rapporte quelque chose qui a déjà eu lieu. Les CQRS Documents de Greg Young demandent de nommer chaque événement avec un verbe au passé, comme `CustomerRelocated` ou `CargoShipped`, car ce nom appartient au langage que l’entreprise emploie pour parler de son activité. Un fait passé ne peut être annulé ; les événements ne sont donc jamais modifiés ni supprimés. Young compare le journal au grand livre comptable : une écriture erronée n’est pas effacée, mais corrigée par une nouvelle. Il applique la même règle à la suppression. Lorsqu’un client retire deux paires de chaussettes de son panier, un événement de retrait s’ajoute au journal. Young appelle cette écriture une contre-passation : le panier se retrouve comme si les chaussettes n’avaient jamais été ajoutées, mais le journal montre toujours qu’elles l’ont été.

Les événements sont conservés dans des flux, un par objet dont ils décrivent l’état. Young, qui s’appuie sur la conception pilotée par le domaine, appelle cet objet un agrégat. Chaque événement porte l’identifiant du flux et un numéro de version unique et séquentiel au sein de ce flux. L’ordre des événements ne fait donc jamais de doute. Pour charger une commande, l’application lit ses événements par version et les applique successivement. Young souligne le peu

de travail demandé au stockage lui-même : dans sa forme la plus simple, il propose deux opérations, lire tous les événements d'un flux dans l'ordre et lui en ajouter de nouveaux.

Le journal devient ainsi la source de référence. Fowler précise que le journal ou l'état actuel peuvent constituer le registre officiel. Dans un système reposant sur l'event sourcing, c'est le journal. Tout le reste, table des soldes actuels, index de recherche ou rapport, est une copie dérivée que l'on peut jeter et reconstruire.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que l'event sourcing ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. La partie invisible pour les utilisateurs : où résident les données, quelles règles s'appliquent et comment les autres systèmes peuvent y accéder.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 6 pipelines ETL/ELT conçus et exploités — 6 pipelines ETL sur trois moteurs de bases de données

6 pipelines ETL/ELT conçus et exploités Contactez-nous →

Pourquoi c'est important

Fowler cite trois possibilités offertes par un journal complet. La reconstruction complète : on peut supprimer l'état de l'application et le recréer en rejouant tous les événements sur une application vide. La requête temporelle : l'état à tout instant passé est le résultat du rejeu des événements jusqu'à cet instant. Enfin, le rejeu : si un événement s'avère erroné, le système peut inverser ses effets et ceux des événements suivants, puis rejouer la séquence corrigée. Le même journal est aussi une piste d'audit qui se constitue sans que personne ait à penser à la rédiger. Fowler cite l'assistance et le débogage parmi les usages courants : l'équipe d'assistance voit exactement ce qu'a fait l'utilisateur, et un développeur peut rejouer les événements de production dans un environnement de test.

Young illustre le bénéfice par deux équipes d'un commerçant en ligne, chargées du même rapport sur les articles retirés du panier par les clients. L'équipe qui ne conserve que l'état actuel commence à enregistrer les retraits ; son rapport dispose de données à compter de ce jour. Celle qui conserve les événements écrit le même gestionnaire de rapport et l'exécute sur le journal depuis le début. Son premier rapport couvre donc déjà plusieurs années. On peut encore répondre à une question que personne n'avait envisagée à la conception, car le système a conservé ce qui s'est passé plutôt que son seul résultat final.

Les événements permettent aussi aux autres systèmes de connaître un changement. Young observe que beaucoup de systèmes enregistrent leur état, puis publient un événement dans une file. Selon lui, cela exige en toute rigueur une validation en deux phases entre les deux : un plantage intermédiaire les laisserait en désaccord. Il propose de traiter le stockage d'événements lui-même comme une file. Un numéro de séquence sur chaque événement permet à un processus distinct de les publier dans l'ordre ; il n'y a ainsi qu'une écriture. Ce stockage n'acceptant que des ajouts, il est aussi plus facile à faire évoluer en capacité. Young note qu'il exige bien moins de verrous qu'un stockage mis à jour sur place, et que l'identifiant de l'agrégat est la seule clé nécessaire au partitionnement d'un stockage d'événements.

Les coûts sont réels, et Fowler les expose. Faire de chaque changement un événement est un style de programmation qui ne convient pas à tout le monde. Le modèle est très difficile à ajouter après coup à un système qui n'a pas été conçu autour de lui. Le stockage ne répond en outre qu'à une sorte de question : « Donnez-moi ce flux. » Dans l'exemple de Young, il ne peut dire quels utilisateurs s'appellent Greg. De telles questions nécessitent un modèle de lecture distinct, construit à partir des événements. C'est pourquoi il associe l'event sourcing à CQRS, qui sépare les modèles d'écriture et de lecture.

À quoi cela ressemble en pratique

Le plus petit stockage d'événements fonctionnel est une table PostgreSQL, proche de la structure proposée par Young : identifiant d'agrégat, données de l'événement et version. Ici, elle contient le stock d'un article en vente. Chaque événement indique une variation du nombre d'unités en vente plutôt qu'un nouveau total. Fowler juge cette forme la plus simple à inverser : « ajouter dix » s'annule en retranchant dix, tandis que « fixer à 110 » a perdu la valeur remplacée.

```
CREATE TABLE stock_events (
    item_id  text    NOT NULL,
    version  integer NOT NULL,
    event    text    NOT NULL,
    order_id text,
    units    integer NOT NULL,  -- change to the units on sale
    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);

-- current state: fold the whole stream
SELECT sum(units) AS on_sale FROM stock_events
WHERE item_id = 'item-881';                    -- 99

-- temporal query: the item as it stood after version 2
SELECT sum(units) AS on_sale FROM stock_events
WHERE item_id = 'item-881' AND version <= 2;   -- 98
```

La clé primaire réalise le contrôle de concurrence optimiste décrit par Young. Deux passages en caisse ayant tous deux lu la version 5 du flux et décidé de réserver une unité tenteront chacun d'ajouter la 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);
-- the second fails: (item-881, 6) already exists
```

La documentation PostgreSQL décrit une clé primaire portant sur plusieurs colonnes comme garantissant l'unicité de leur combinaison. Une insertion qui la dupliquerait lève une erreur. Le traitement de passage en caisse dont l'insertion a échoué recharge le flux, vérifie qu'une unité est toujours en vente et réessaie avec la version 7. Aucun des deux n'a vendu une unité inexistante ; le journal montre les deux décisions dans leur ordre réel.

La note Vendre le dernier billet exactement une fois traite le même problème de stock et montre les limites de cette structure. Sa première solution garde le compteur dans une ligne et le modifie avec un seul `UPDATE` conditionnel. Elle déconseille le verrouillage optimiste pour un compteur très sollicité, car une colonne de version transforme la contention en nouvelles tentatives. Pour l'article le plus demandé, elle place chaque requête dans une seule partition Kafka, lue par un consommateur unique qui garde le compteur en mémoire et valide ses décisions par lots. Elle appelle cela la structure LMAX. L'article de Fowler sur l'architecture LMAX, une des sources de la note, décrit l'event sourcing sous sa forme la plus pure : un moteur de négociation sans aucune base, dont tout l'état dérive des événements d'entrée. Un processus de journalisation les stocke durablement pour permettre leur rejeu. Le passage en caisse de la note utilise des événements d'une autre nature. Chaque étape écrit un message

dans une table d'outbox ; la capture des changements le publie, et les tables restent la référence. Ces événements annoncent les changements ; dans l'event sourcing, ils constituent la référence.

Où les problèmes surviennent

Le rejeu répète les effets de bord. Un gestionnaire qui envoie un courriel de confirmation ou débite une carte le refait au rejeu du journal. Fowler souligne que les systèmes externes ne distinguent pas le rejeu du travail réel. Reconstruire un modèle de lecture peut alors envoyer à chaque client les courriels de l'année précédente. Placez chaque système externe derrière une passerelle qui sait quand l'application rejoue le journal et supprime les appels sortants pendant ce rejeu. La logique du domaine n'a ainsi pas à le savoir. Les requêtes demandent la même précaution : dans l'exemple de conversion monétaire de Fowler, l'événement rejoué exige le taux de change de son jour, pas celui d'aujourd'hui. La passerelle devrait donc enregistrer les réponses reçues et les restituer au rejeu.

Un flux très sollicité multiplie les nouvelles tentatives. La concurrence optimiste suppose des conflits rares. Si beaucoup de processus écrivent simultanément dans un flux, chaque conflit impose de le recharger et de réessayer. Lors d'un afflux, les nouvelles tentatives deviennent la charge ; c'est l'avertissement de la note sur les colonnes de version. Délimitez les flux pour que chacun ait rarement deux processus d'écriture au même instant : un flux par commande plutôt qu'un pour toute une gamme de produits. Si un flux est réellement très sollicité, confiez-le à un seul processus d'écriture, comme dans LMAX. Il ne reste alors plus de concurrence à résoudre.

Les longs flux sont lents à charger. Rejouer des milliers d'événements pour charger un objet coûte du temps à chaque requête. Fowler cite les reconstructions lentes parmi les raisons de garder une copie de l'état actuel. Young propose l'instantané glissant : l'état sérialisé d'un flux à une version connue, conservé dans une table distincte et écrit en arrière-plan par un processus séparé. Charger consiste alors à lire l'instantané le plus récent, puis les événements postérieurs à sa version. Gardez les instantanés hors du journal d'événements lui-même : Young décrit un processus d'instantanés échouant sans cesse au contrôle de concurrence sur un flux très actif. Il conseille aussi de construire sans instantanés ; on pourra toujours les ajouter ensuite pour accélérer le chargement.

Les lecteurs prennent du retard sur l'écriture. Chaque modèle de lecture est construit à partir des événements après leur écriture ; il suit donc le journal avec un décalage. Fowler prévient que les systèmes lisant le journal risquent de ne pas être synchronisés avec lui. Un utilisateur qui soumet un formulaire puis voit une liste sans son changement le soumettra de nouveau. Faites enregistrer à chaque modèle le dernier événement appliqué, pour mesurer son retard et pouvoir le reconstruire depuis le début. Montrez aux utilisateurs le résultat de leur propre commande à partir de la réponse à celle-ci, plutôt que d'un modèle de lecture qui peut ne pas être à jour.

Pour aller plus loin

- Event Sourcing — la description de Martin Fowler : reconstruction complète, requête temporelle, rejeu et passerelles empêchant les rejeux d'atteindre les autres systèmes.
- CQRS Documents, par Greg Young — les événements comme faits au passé, la contre-passation plutôt que la suppression, un stockage d'événements à deux tables, les instantanés glissants et le stockage comme file.
- L'architecture LMAX — un moteur de négociation en mémoire dont tout l'état dérive d'événements d'entrée journalisés.
- PostgreSQL : contraintes — les contraintes d'unicité et de clé primaire sur plusieurs colonnes, la part du contrôle de concurrence optimiste assurée par la base.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que l'event sourcing ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

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

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/event-sourcing/