Event Sourcing 101: When to Use and How to Avoid Pitfalls

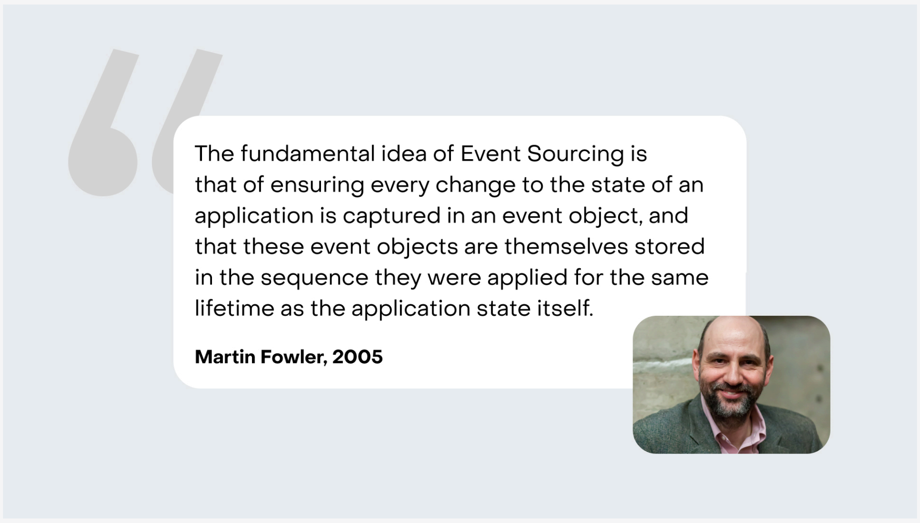
Learn the basics of event sourcing and its benefits, challenges, and practical use cases to decide if it's the right fit for your software project.

By Vladyslav Rybnikov · Mar. 14, 25 · Analysis

Recently, I delivered a lecture to my colleagues on event sourcing, and I realized that this introductory information could be valuable to a broader audience.

This article is useful for those interested in the concept of event sourcing and who want to decide if it's a good fit for their projects while avoiding common pitfalls. So, let's dive in.

Event sourcing is an approach where, instead of storing the current state of the system, all changes are saved as events, which become the main data source. The approach gained popularity around 2005 after Martin Fowler's article on the topic.

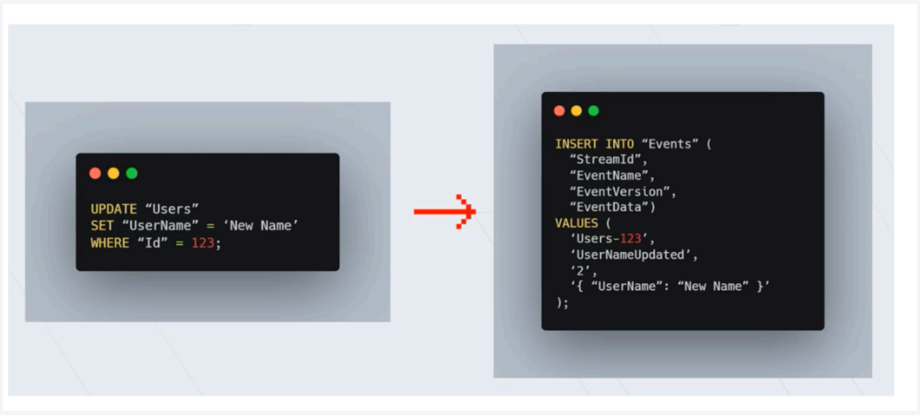


The fundamental idea of Event Sourcing is that of ensuring every change to the state of an application is captured in an event object, and that these event objects are themselves stored in the sequence they were applied for the same lifetime as the application state itself.

Martin Fowler, 2005

The main idea is that instead of storing and updating the application's state, you save the events that describe changes. Events act as the core source of information. This differs from the traditional approach, where the current state is saved and updated with each change. In event sourcing, each change is logged as a new event instead of modifying an existing record.

For example, in an app where users can edit their profiles, the traditional approach uses an "Update" command to modify the existing database record. With event sourcing, instead of "Update," we use "Insert," adding a new entry to an event log that records the change.



```
UPDATE "Users"
SET "UserName" = 'New Name'
WHERE "Id" = 123;
```

```
INSERT INTO "Events" (
  "StreamId",
  "EventName",
  "EventVersion",
  "EventData")
VALUES (
  'Users-123',
  'UserNameUpdated',
  '2',
  '{ "UserName": "New Name" }'
);
```

Events include a user identifier (StreamID), the event name, version, and new data, such as a new username. This creates an "append-only" log, an immutable record where each change is a separate event.

This method preserves the full change history, simplifying auditing, recovery, and data analysis.

Event Sourcing Pros and Cons

Below are some benefits of event sourcing:

1. **Observability**. You can track all changes, create an audit log, and restore data states at any point in time.
2. **System decomposition**. Supports asynchronous interactions between system components and microservices.
3. **Enhanced fault tolerance**. Reduces data loss risk, especially when there are multiple Event Store replicas.
4. **Easy data migration**. Data can be easily transferred between databases by replaying events.

Challenges

1. **Complex implementation and maintenance**. Event sourcing isn't always the best fit.
2. **Potential overengineering**. It's not suitable for every project.

Before implementing event sourcing, it's essential to answer these questions:

1. **Can events be considered part of my app's domain**?

Determine if you plan to use events for asynchronous tasks, like sending emails, or for breaking down the system into microservices that interact through events.

2. **Can events help me develop and maintain my app**?

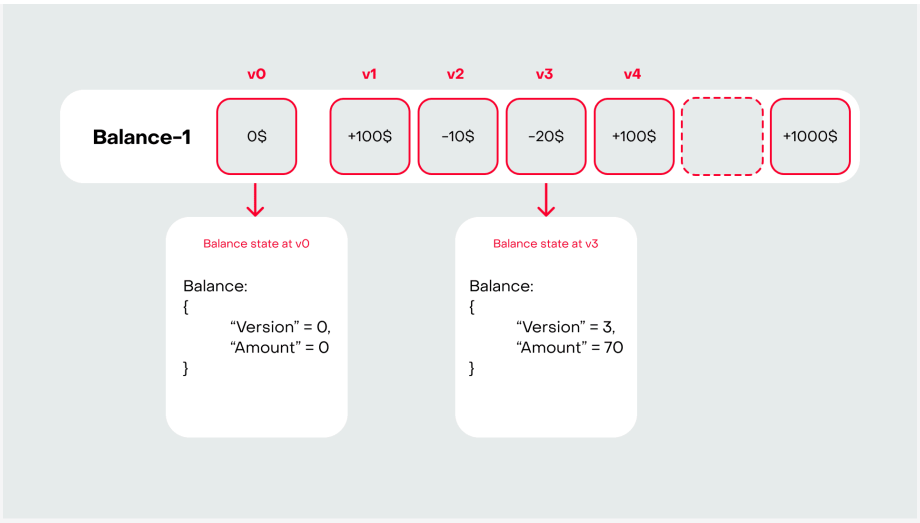
Consider whether event storage will support your development and maintenance needs.

3. **Do I need to know about every change in my app**?

Evaluate if you need to track every change or if there are storage limitations for tracking all operations.

With these questions in mind, you can decide if event sourcing is suitable for your application.

For example, in financial apps where account balances frequently change, events can efficiently reflect these adjustments. Each balance change links to events like deposits or withdrawals. This approach enables you to recreate the account's state at any given time by applying the sequence of events, each with a specific type and logic.



Here are some examples of the approach across different domains:

FinTech

- **State example**: Account balance
- **Events**: Deposit, withdrawal, financial month start, account creation
- **Projections**: Transaction logs, account history

Supply Chain

- **State example**: Shipment status
- **Events**: Delivery requested, accepted by carrier, status updates
- **Projections**: Delivery requests, carrier pending deliveries

Healthcare

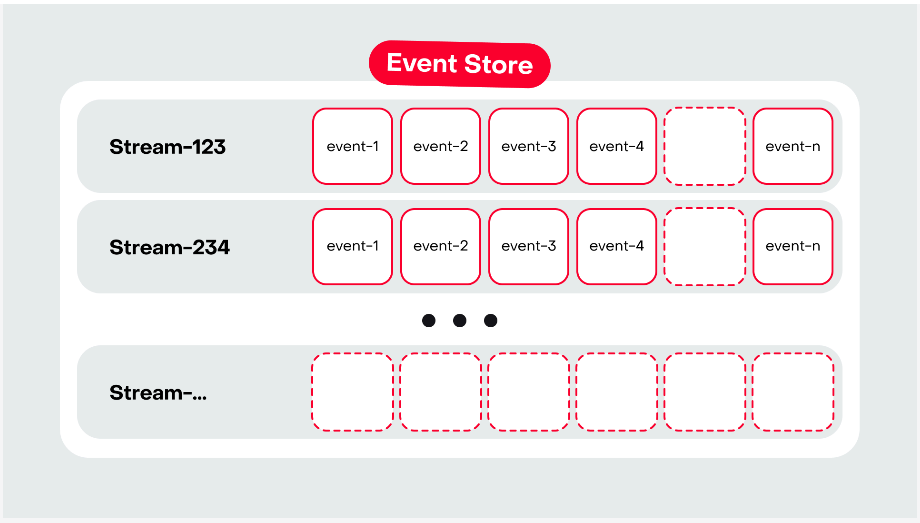
- **State example**: Patient's medical record
- **Events**: Doctor visits, medical exams
- **Projections**: Medical history

Event Sourcing Building Blocks

To build an event-sourcing system, focus on these key components:

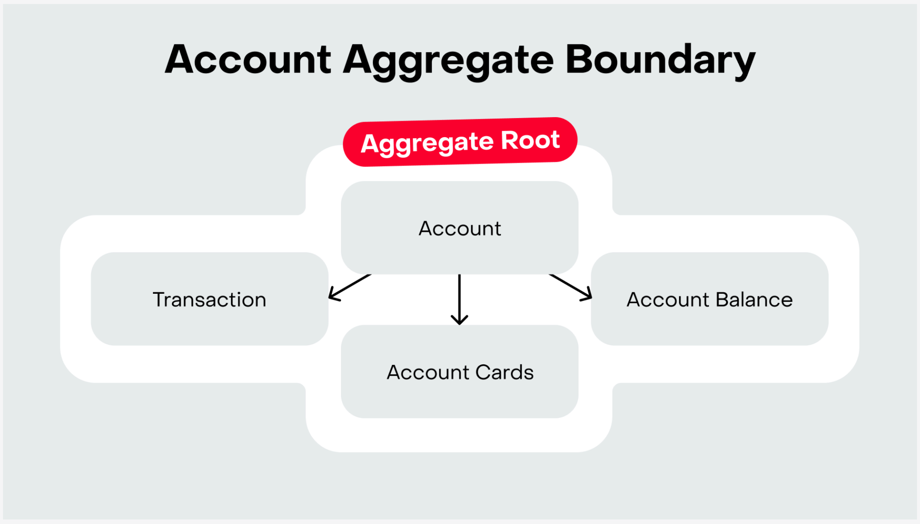
1. Event Store

A storage solution for all events in the system, grouped into streams with unique IDs and ordered versions. For example, EventStoreDB is a popular choice. The event store organizes events into streams, each with an ID and a chronological list of events for a particular entity.



2. Aggregates

A Domain-Driven Design (DDD) concept that groups multiple objects into a whole, each with a unique ID. For a financial app, an aggregate could be an account that ties together balances and cards, while transactions might form a separate aggregate. The root of an aggregate is the access point for managing changes.



3. Events

Each event has a unique ID, version, and type, and is immutable. For example, a transaction event can be represented in JSON with fields like version, name (e.g., "deposit"), amount, currency, and other details.

4. Event Stream

An ordered sequence of events, each with a version in the stream. There are different ways to organize streams:

- **Single stream per aggregate**. Each stream is tied to a specific aggregate, like a "Balance" stream for account ID 1 with events only for that balance.
- **Multiple streams for one aggregate**. You can create separate streams for different periods or categories, such as monthly balance streams.
- **Global event stream**. Contains all events for all aggregates, ordered by time.

5. Projections

Also called read models, query models, or view models, projections organize events into user-friendly formats, like transaction histories or account statements. They can aggregate or group events to simplify data access.

6. Snapshots

Optional state saves for aggregates, used to optimize state restoration. Snapshots are not required but can improve performance when replaying event histories.

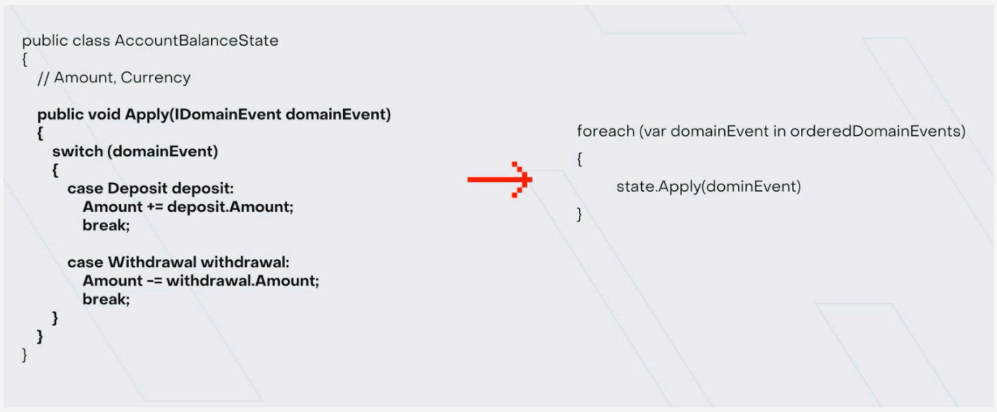
Building Aggregate States

To build an aggregate state from events in a stream, follow these steps:

1. Replaying StateFrom Events

To recreate the account's state, define a class like 'AccountBoundState' with fields such as 'Amount', 'Currency', and other properties. The class includes an 'Apply' method that takes an event and adjusts the state based on the event type:

- For a deposit event, the 'Apply' method increases 'Amount' by the deposit amount.
- For a withdrawal event, the 'Apply' method decreases 'Amount' by the corresponding amount.



```
public class AccountBalanceState
{
  // Amount, Currency

  public void Apply(IDomainEvent domainEvent)
  {
    switch (domainEvent)
    {
      case Deposit deposit:
        Amount += deposit.Amount;
        break;

      case Withdrawal withdrawal:
        Amount -= withdrawal.Amount;
        break;
    }
  }
}
```

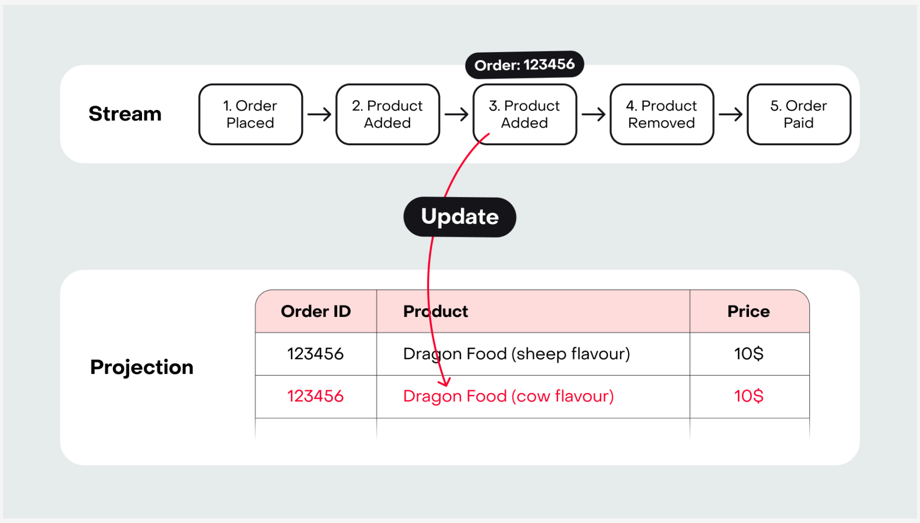
```
foreach (var domainEvent in orderedDomainEvents)
{
    state.Apply(dominEvent)
}
```

This approach allows you to restore the account's state at any given version by sequentially applying each event up to the desired version.

2. Using Projections

Projections simplify data access by creating read-only models for efficient querying. Projections can be tailored for individual events or aggregated for groups of events. Examples include:

- **Transaction projection**. Stores a list of all account transactions.
- **Balance projection**. Reflects the current account balance.



Order ID	Product	Price
123456	Dragon Food (sheep flavour)	10$
123456	Dragon Food (cow flavour)	10$

Optimizing With Snapshots

When the number of events grows very large (millions or more), replaying all events becomes inefficient. Snapshots help by saving the aggregate's state at specific versions, so you can start from the latest snapshot rather than from the beginning.

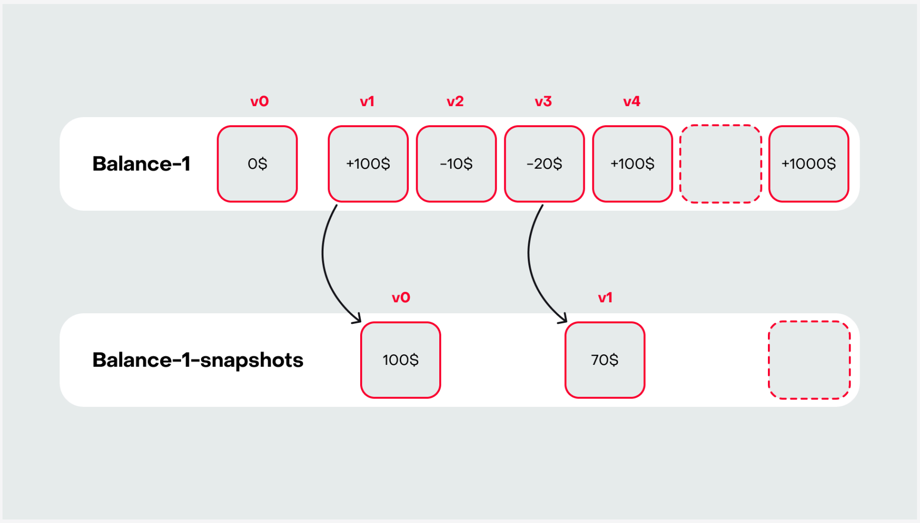
Let's load all the events
Apply them to our state
And append another event to the stream
We need to load one billion of events

C#

```
var snapshot = snapshots.LastOrDefault();
state.LoadFromSnapshot(snapshot);
foreach (var domainEvent in orderedDomainEvents.Where(x => x.Version > snapshot.Version))
{
   state.Apply(domainEvent);
}

```

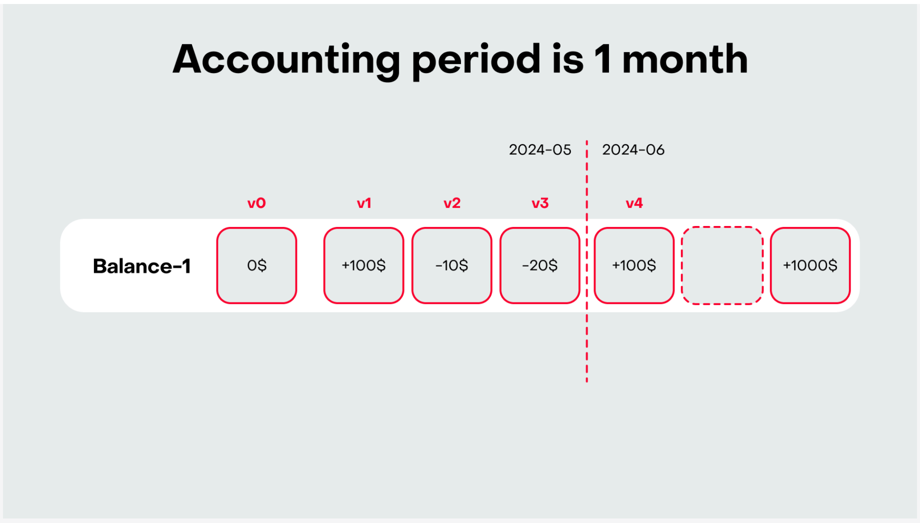
- Snapshots can be stored in a separate stream, like `balance-1-snapshot`.
- To restore the state to a certain version, retrieve the latest snapshot and apply all events with versions higher than the snapshot's version.

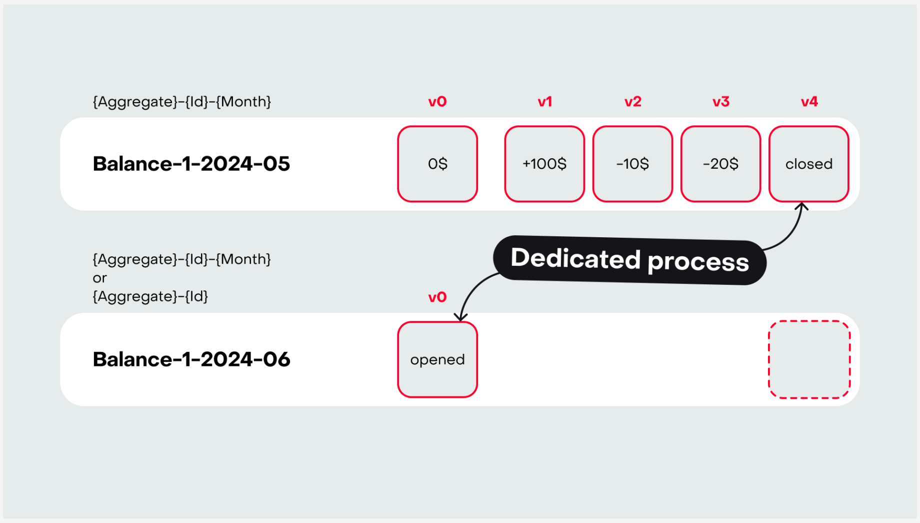


Snapshot Creation Strategy

Decide how often to create snapshots based on system usage:

- Every N event (e.g., every 100 events),
- At specific time intervals (e.g., at the start of each new month),
- A combination of both approaches.





Main Challenges in Event Sourcing and Their Solutions

1. Concurrency Updates

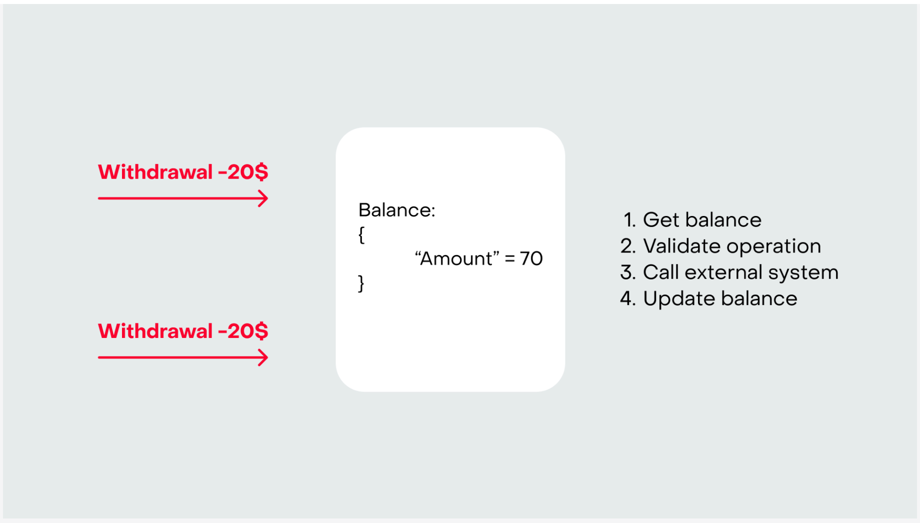
In distributed systems with multiple microservices and parallel access to data, concurrency updates and distributed transaction issues arise. The primary approaches to handling concurrency updates include:

Pessimistic Concurrency Control

This approach locks the resource (e.g., a database record) during an operation, so other processes have to wait until the lock is released. It can be implemented using distributed locks, such as with Redis or PostgreSQL.

Optimistic Concurrency Control

Instead of locking the resource, this approach relies on version checking. The object state holds a version, and during an update, it verifies that the version in the database matches the expected version. If not, the operation retries: the state is read again, validated, and updated.



In event sourcing, rather than a mutable state, there's an event stream (append-only log). This avoids concurrency update issues since new events are only appended to the end of logs. If each new event needs to depend on the previous one, optimistic version control is used, naturally supported by many event-sourcing implementations. If the dependency between events is unimportant (e.g., adding items to an order in an online store), events can be added without version checking.

2. Distributed Transactions

In distributed systems, it's crucial to ensure that changes across various data stores or services are consistent. Solutions include:

2-Phase Commit

Suitable for transactions across multiple systems, though not supported by all data stores. This process involves two stages: preparation (all participants confirm readiness) and commit, provided all are ready.

Outbox Pattern

The idea here is to write events to an "outbox" (a special messages table) in parallel with the database write. Then, a dedicated service asynchronously sends these messages to a queue, ensuring reliable delivery.

Transactional Saga Pattern

Manages distributed transactions through orchestration (centralized sequence of steps) and choreography (each step decides the next based on the current transaction state).

3. Projection Synchronization

When a system uses projections for data reads, it's essential to ensure their consistency and synchronization with the main event store. This can be achieved with:

Asynchronous Updates

Projections are updated asynchronously as new events are added. A message queue can be used so that each projection receives updates automatically.

Eventual Consistency

The system allows a small delay between event recording and projection updates. Ensuring all required events are processed is key.

These approaches allow for flexibility in meeting specific system requirements, maintaining consistency, and high performance in distributed environments.

4. Optimizing Read Process and State Computation

Event sourcing involves replaying the state from all events, which can be resource-intensive for large streams. Key optimization techniques include:

Snapshotting

Creating saved snapshots of an object's state at specific versions. For example, a snapshot of an account's state can be created every 100 events. This way, when replaying, the system starts from the latest snapshot and only applies new events, reducing the number of operations required to restore the current state.

Projections

Representing events in a convenient format for queries, such as a transaction history or current account balance. Projections can be updated in real time or periodically, depending on the specific use case.

CQRS (Command Query Responsibility Segregation)

This approach separates write operations (event creation) from read operations (queries) into different data models. Writing is done via event sourcing, while reading is done through pre-built projections, which optimizes system performance.

Conclusion

So, while event sourcing presents challenges such as handling distributed transactions and concurrency control, these can be effectively managed through established patterns and best practices. By thoughtfully implementing this concept, teams can build applications that are more resilient, maintainable, and scalable, making it a valuable asset in modern software development.