Event Sourcing Pattern

Last Updated : 23 Jul, 2025

-
-
-

The Event Sourcing Pattern is a way to store data by recording every change as a sequence of events, instead of just saving the latest state. This approach is particularly useful for applications needing a complete history of operations, easy data recovery, or complex auditing. By replaying these events, the system can rebuild the current state or investigate past states, making Event Sourcing ideal for handling complex workflows.

Event Sourcing Pattern

Table of Content

What is Event Sourcing?

The Event Sourcing Pattern is like keeping a detailed diary for your software. Instead of just updating the current state of your data, you record every change as a separate event. These events form a complete history of what happened to your data over time. Therefore, you may rebuild your data by replaying these events to find out how it got to where it is now.

- These events are stored sequentially, forming a log or journal of actions that have occurred.
- The status of the program can be restored at any moment by replaying these events.
- It is frequently utilized in fields like finance and e-commerce where precise historical data are essential.

Core Concepts and Components of Event Sourcing

Event Sourcing in system design revolves around several core concepts and components:

1. **Events:** These are permanent records of system state changes. In addition to representing a particular action or occurrence, each event includes all the relevant information required to rebuild the system's state at the moment of the event.

2. **Event Store:** The stream of events produced by the system is sustained by the event store, a robust data store. It guarantees that the sequence of events is maintained by storing them in the order in which they were received.
3. **Aggregate:** For the purposes of processing commands and producing events, an aggregate is a logical collection of linked domain objects that are handled as a single unit. The system's state changes and business logic are contained in aggregates. In the event store, each aggregate is linked to a distinct stream of events.
4. **Command:** Clients or other system components can issue commands, which are requests or instructions to carry out particular tasks. Aggregates process commands by validating them, applying business logic, and, if the command is approved, generating the appropriate events.
5. **Projection:** Projections are read models created from the stream of events kept in the event store that shows the system's current state. Projections are used to give effective access to data for reporting and querying.
6. **Event Bus:** An event bus is a messaging infrastructure that makes it easier for various system components to communicate about events. It enables components to respond asynchronously to particular event types and subscribe to them.

How Event Sourcing Pattern works?

Below are the steps to understand how event sourcing pattern works:

- **Capture Events Instead of State**: Every system modification is documented as an event (e.g., “order created,” “item added,” “order completed”) rather than just preserving the final state (e.g., “order is completed”). Every event denotes a distinct action or modification.
- **Store Events in Sequence**: All events are stored in a sequence (often in a database or event store) in the exact order they occurred. This sequence of events acts as the "source of truth" for the system.
- **Reconstruct State by Replaying Events:** When you need to know the current state, the system “replays” or processes all past events to build up the state from scratch.
- **Handle New Events:** As new changes happen, new events are created and added to the sequence. For instance, if an order is updated, a new event (e.g., “order updated”) is added to the sequence without changing or removing past events.
- **Replay Events for Debugging:** If you need to see the history or investigate an issue, you can replay events to see how the state evolved over time. This replay ability makes it easy to understand the sequence of actions taken and trace any errors or issues.

Example of Event Sourcing Pattern

Lets understand event sourcing pattern with the help of an example of registration system:

1. Registration System Before Event Sourcing

In a traditional event registration system, user registrations and cancellations are handled using a direct approach where the state is updated immediately in the database. This can lead to challenges such as difficulty in tracking changes, loss of historical data, and issues in notifying users when their registration status changes.

Below is the traditional registration class in which there are several issues:

- When users register or cancel their registrations, the system updates the database immediately without maintaining a history of actions.
- The system does not record the reasons for registration changes, making it hard to audit past actions.
- Notifications to users must be handled separately, leading to potential delays and errors in communication.

```
public class Registration {
    private RegistrationState state;
    private String userId;
    private String eventId;

    public void register(String userId, String eventId) {
        this.state = RegistrationState.REGISTERED;
        this.userId = userId;
        this.eventId = eventId;
        // Update the database directly
        Database.update(this);
    }

    public void cancel() {
        this.state = RegistrationState.CANCELLED;
        // Update the database directly
```

```
        Database.update(this);
    }
}
```

In the above code, When a user registers or cancels, the state is directly updated in the database, which may cause inconsistencies if issues arise during processing. Notifications to users are not integrated with the registration process, which may lead to missing or delayed alerts.

2. Registration System Using Event Sourcing

With the move to an event-sourced architecture, the system now records each significant change as an event, allowing better tracking and historical analysis.

- Instead of directly updating the database, the system processes commands to create and cancel registrations and generates events for each action.
- Each state change is represented by an event, providing a clear audit trail of user actions.
- Notifications are handled through event subscribers, ensuring users are promptly informed about their registration status changes.

Below is the event sourced registration class:

```
public class Registration extends
ReflectiveMutableCommandProcessingAggregate<Registration, RegistrationCommand> {
    private RegistrationState state;
    private String userId;
    private String eventId;

    public RegistrationState getState() {
        return state;
    }

    public List<Event> process(RegisterForEventCommand cmd) {
        return EventUtil.events(new EventRegisteredEvent(cmd.getUserId(), cmd.getEventId()));
    }
```

```
    public List<Event> process(CancelRegistrationCommand cmd) {
        return EventUtil.events(new RegistrationCancelledEvent(userId, eventId));
    }

    public void apply(EventRegisteredEvent event) {
        this.state = RegistrationState.REGISTERED;
        this.userId = event.getUserId();
        this.eventId = event.getEventId();
    }

    public void apply(RegistrationCancelledEvent event) {
        this.state = RegistrationState.CANCELLED;
    }
}
```

By transitioning to an event-sourced architecture, the event registration system gains a clear advantage in tracking registration states, maintaining a complete history of actions, and ensuring timely notifications to users.

Benefits of Event Sourcing Pattern

Event Sourcing offers several benefits in system design:

- By recording all system modifications, Event Sourcing builds an unalterable record of all activities.
- With event sourcing, you can replay events up to a certain point in time to query the system's current state.
- Event sourcing encourages adaptability and system development over time. New event types can be created to address changes in business requirements without affecting current components.
- Event Sourcing is a good fit for distributed systems and horizontal scaling as events can be individually consumed and processed by several components, enhancing scalability and speed.

Challenges with Event Sourcing Pattern

While Event Sourcing offers various benefits, it also presents several challenges:

- Events may take on a different structure as the system develops. It becomes essential to manage event versioning and backward compatibility to guarantee seamless migrations and upgrades without erasing previous data.
- It can be difficult to properly store and retrieve huge amounts of events, particularly in situations with high throughput or lengthy event histories. It becomes crucial to put in place efficient querying and event storing systems.
- Various components of the system may see state changes at various times due to eventual consistency, which is usually the outcome of event sourcing. Handling stale data, resolving conflicts, and data synchronization all need careful thought when dealing with eventual consistency.

Use Cases and Applications of Event Sourcing

Event Sourcing finds applications in diverse domains and use cases where tracking and analyzing historical data is crucial. Below are some notable examples:

- **Financial Systems:** To keep an unchangeable record of transactions, account activity, and compliance events, banking and financial institutions use event sourcing.
- **Healthcare Records:** Electronic health record (EHR) systems use event sourcing to document patient contacts, treatment histories, and medical procedures.
- **Supply Chain Management:** Event sourcing is used in supply chain and logistics management to track inventory movements, shipping updates, and supply chain interruptions.
- **E-commerce Platforms:** Event sourcing is a useful tool for e-commerce applications to record customer interactions, order fulfillment processes, and inventory changes. This makes it feasible to track orders, offer customized recommendations, and examine customer behavior in order to boost sales and customer satisfaction.
- **Gaming and Virtual Environments:** Event Sourcing is used in virtual worlds and multiplayer online games to record player movements, game state changes, and in-game purchases.

Real-World Example of Event Sourcing Pattern

A real-world example of Event Sourcing can be seen in a modern banking system.

Consider a scenario where a bank wants to implement Event Sourcing for its account management system.

- **Account Operations:** The system creates related events whenever a consumer does an action that impacts their account, like making a deposit, taking money out of their account, moving money to another account, or changing personal data.
- **Event Logging:** elevant details including the kind of operation, the amount involved, timestamps, and any additional metadata required to recreate the account's status are contained in each event.
- **Event Processing:** These events are consumed by event handlers or processors, who then alter the status of the impacted accounts appropriately. For instance, the account balance is increased if a "Deposit" event is noted. The balance is reduced if a "Withdrawal" event is recorded.
- **Scalability and Resilience:** Event Sourcing distributes event processing over several nodes, enabling the bank's system to scale horizontally. The system can recover by replaying events from the event storage to restore the status of accounts in the event of malfunctions or outages.

When to Use Event Sourcing Pattern

Use Event Sourcing Pattern when:

- It's good for apps with complicated rules that need to keep track of how things change over time.
- It helps when you need a complete history of changes for legal reasons or audits.
- Useful for systems that need to save old data, like financial information.
- Helps your app bounce back quickly from problems by replaying past events.
- Works well in systems with separate parts (like microservices) that need to communicate without being tightly linked.

When not to Use Event Sourcing Pattern

Event Sourcing may not be the best choice in these situations:

- **Simple Applications**: If an application just needs to track the most recent state and doesn't require a thorough history or audit trail, Event Sourcing may introduce needless complexity.
- **High Storage Costs**: Event Sourcing stores every change as an event, which can lead to high storage costs, especially if there are frequent updates or changes.
- **Complex Data Consistency Needs**: Managing consistency across multiple events can become challenging, especially when coordinating data between different services or systems.

- **Performance Sensitivity**: Replaying events to rebuild the state can slow down performance in applications where quick data retrieval is essential.

Conclusion

In conclusion, Event Sourcing is like keeping a detailed diary for your software. By recording every change as a separate event, it provides an accurate and immutable record of all actions taken within a system. This pattern offers benefits such as improved auditability, scalability, and resilience. It enables accurate historical analysis, real-time tracking of system state, and efficient troubleshooting.