# Event Sourcing

Think Aggregate Storage in terms of Event Store

# Agenda

- **Part 1**
  - Context & Problem
  - What? and How? of Event Sourcing
  - Handling Concurrent Updates
  - persists Aggregates using Events
  - Publishing Events
- **Part 2**
  - Snapshotting to improve Performance
  - Idempotency
  - Evolving domain events
  - Benefits and Drawbacks of ES
  - Code Example Walkthrough

# Context

# Context

- Database per service pattern
- Distributed Transaction Management
  - 2PC is not a good fit - RabbitMQ, Kafka etc and MongoDB and Cassandra etc do not support it
  - 2PC may reduces service availability
  - Saga Pattern
    - Reliably publishing message/event after DB update is crucial
    - Also need to preserve event ordering
      - T1 -> E1, T2 -> E2 ... if T1 processed before T2 then E1  must appear before E2

# Problem: Issues in Traditional Persistence

- Object Relational Impedance Mismatch
- Lack of aggregate history
- Implementing audit logging is tedious and error prone
- Event publishing is bolted onto the business logic

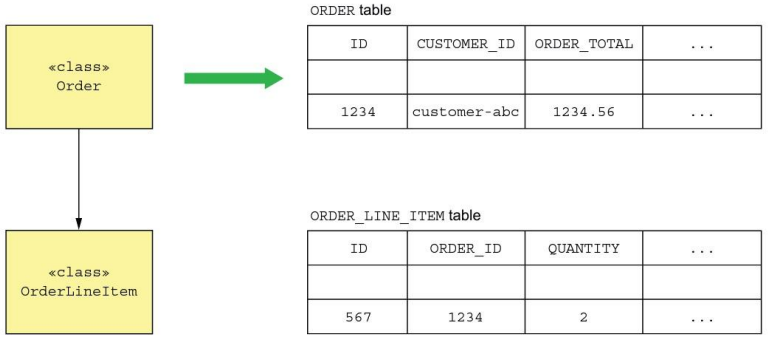



**Figure 6.1 The traditional approach to persistence maps classes to tables and objects to rows in those tables.**

# Problem: Object Relational Impedance Mismatch

- Granularity
  - Object model which has more classes than the number of corresponding tables in the database
- Subtypes (inheritance)
  - RDBMS has no standardised approach to support subtyping / inheritance
- Identity
  - RDBMS defines exactly one notion of 'sameness': the primary key.
  - OOP defines both object identity: a==b and object equality: a.equals(b)
- Associations
  - Unidirectional references in Object Oriented languages whereas RDBMSs use the notion of foreign key
  - For bidirectional relationships in Java, you must define the association twice.
  - Likewise, you cannot determine the multiplicity of a relationship by looking at the object domain model.
- Data navigation
  - To access data in OOP language, e.g., Java, you navigate from one association to another by walking the object network
  - In RDBMS, load several entities via JOINs and select the targeted entities before you start walking the object network

# What and How of Event Sourcing

# Event Sourcing

- Persisting an aggregate as a sequence of domain events that represent state changes:
  - Each Event represent a state change of the aggregate.
  - An Application recreates the state of an aggregate by replaying the events
  - An aggregate’s business logic is structured around the requirement to produce and consume these events.

# Event Sourcing persists Aggregates using Events

- An aggregate persisted as sequence of events in DB, known as an event store.
- E.g. Order Aggregate, instead of storing the order as a row in ORDER table, event sourcing persists each order aggregate as one or more row in EVENTS table. Each row is a domain event, such as Order Created, Order Approved, Order Shipped etc.
  - When an application creates or updates an aggregate it inserts the events emitted by the aggregate into EVENTS table.
  - An application loads an aggregate from the event store by retrieving its events and replaying them.
  - Loading an aggregate consists of following steps:
    - Load the events for the aggregate
    - Create an aggregate instance by using its default constructor
    - Iterate through the events, calling apply()

# Order Aggregate

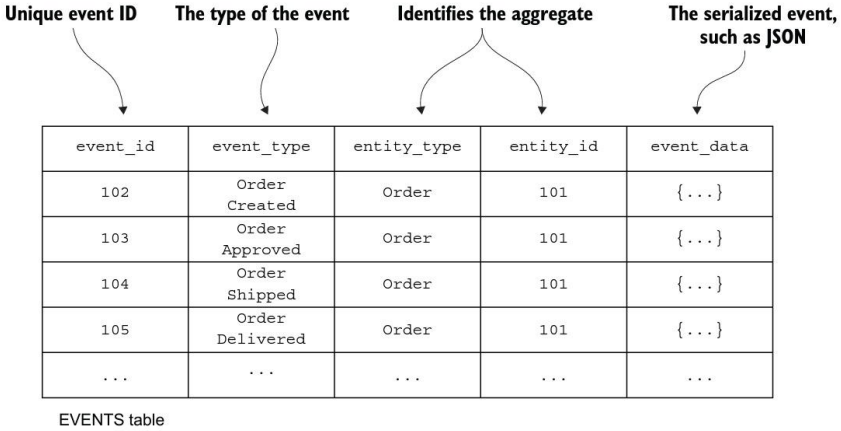



Figure 6.2 Event sourcing persists each aggregate as a sequence of events. A RDBMS-based application can, for example, store the events in an `EVENTS` table.

## Loading an Aggregate example

```
Class aggregateClass = ...;
Aggregate aggregate = aggregateClass.newInstance();
for (Event event : events) {
  aggregate = aggregate.applyEvent(event);
}
// use aggregate...
```

# Event Represent State Change

- An event must contain the data that the aggregate needs to perform the state transition
  - Events can either contain minimal data, such as just the aggregate ID, or can be enriched to contain data that's useful to a typical consumer.
- Events aren't optional when using event sourcing
  - Every state change of an aggregate, including its creation, is represented by a domain event.
  - Whenever the aggregate's state changes, it must emit an event
  - This is more stringent requirement than before, when an aggregate only emitted events that were of interest to consumers
- Because events are used to persist an aggregate, you no longer have the option of using a minimal OrderCreated event that contains the orderId.

# Event Represent State Change

- The current state of the aggregate is S and the new state is S'.
- An event E that represents the state change must contain the data such that when an Order is in state S, calling order.apply(E) will update the Order to state S'.

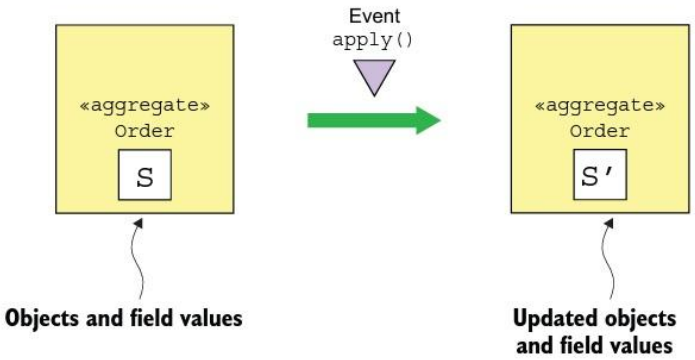



**Figure 6.3 Applying event `E` when the `Order` is in state `S` must change the `Order` state to `S'`. The event must contain the data necessary to perform the state change.**

# Aggregate Methods are all About Events

- To update an aggregate, a command method on aggregate root is called which return sequence of events that represent the state changes which must be made
- These events are persisted in the database and applied to the aggregate to update its state.
- Event sourcing refactors a command method into two or more methods:
  - **The first method takes a command object parameter,**
    - Represents the request, and determines what state changes need to be performed
    - Validates its arguments, and without changing the state of the aggregate, returns a list of events representing the state changes
    - This method typically throws an exception if the command cannot be performed
  - **Other methods take a particular event type as a parameter and update the aggregate**
  - There's one of these methods for each event
- These methods can't fail, as an event represents a state change that has happened and ach method updates the aggregate based on the event
- Some event-sourcing frameworks names these methods process() and apply()

# State Change by Applying Event

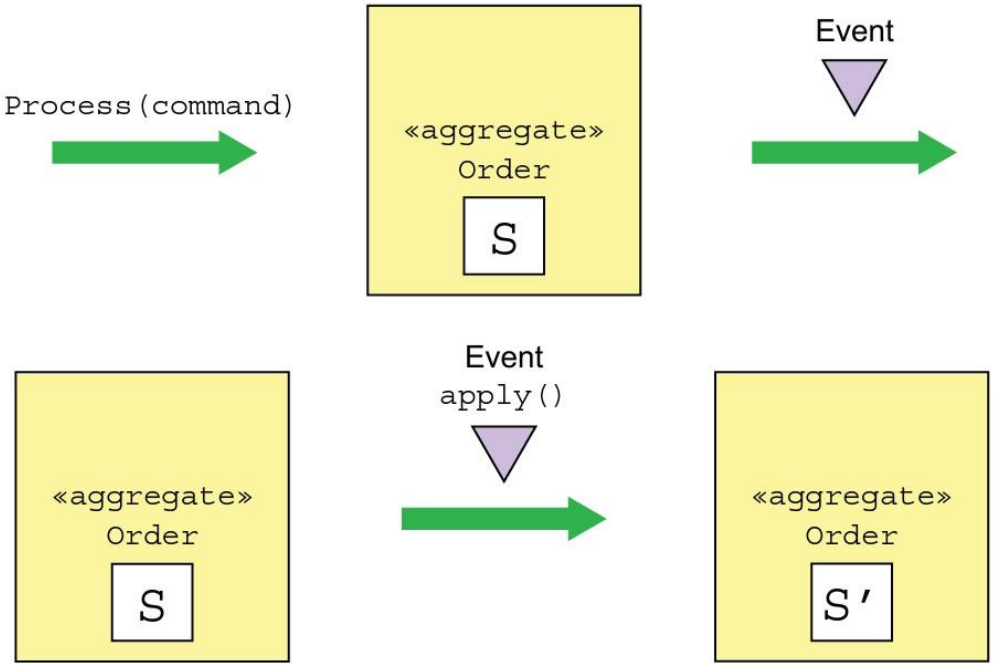



**Figure 6.4 Processing a command generates events without changing the state of the aggregate. An aggregate is updated by applying an event.**

# Command Method Split

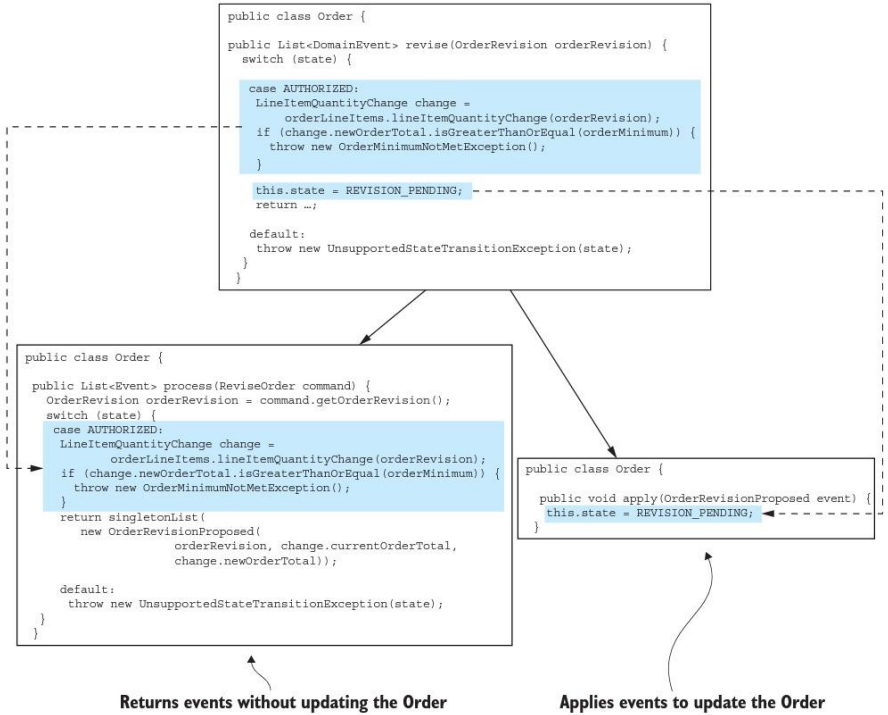



**Figure 6.5** Event sourcing splits a method that updates an aggregate into a `process()` method, which takes a command and returns events, and one or more `apply()` methods, which take an event and update the aggregate.

# Aggregate Creation Vs Updation Steps

**An aggregate creation steps:**

1. Instantiate aggregate root using its default constructor
2. Invoke process() to generate the new events
3. Update the aggregate by iterating through the new events, calling its apply()
4. Save the new events in the event store

**An aggregate updation steps:**

1. Load aggregate's events from the event store
2. Instantiate the aggregate root using its default constructor
3. Iterate through the loaded events, calling apply() on the aggregate root
4. Invoke its process() method to generate new events
5. Update the aggregate by iterating through the new events, calling apply()
6. Save the new events in the event store

# Event Sourcing based Order Aggregate Example

- Event sourcing version of Order aggregate will be somewhat similar to the JPA-based version Its fields are almost identical and it emits similar events
- The difference lies in business logic implementation, in terms of processing commands that emit events then state is updated by applying those events
    - i.e. Each method that creates or updates the JPA-based aggregate, such as createOrder() and reviseOrder()

# Event Sourcing based Order Aggregate Example

- This class's fields are similar to those of the JPA-based Order. The only difference is that the aggregate's ID isn't stored in the aggregate

**Listing 6.1 The `Order` aggregate's fields and its methods that initialize an instance**

```
public class Order {

  private OrderState state;
  private Long consumerId;
  private Long restaurantId;
  private OrderLineItems orderLineItems;
  private DeliveryInformation deliveryInformation;
  private PaymentInformation paymentInformation;
  private Money orderMinimum;

  public Order() {
  }

  public List<Event> process(CreateOrderCommand command) {
     ... validate command ...
    return events(new OrderCreatedEvent(command.getOrderDetails()));
  }

  public void apply(OrderCreatedEvent event) {
    OrderDetails orderDetails = event.getOrderDetails();
    this.orderLineItems = new OrderLineItems(orderDetails.getLineItems());
    this.orderMinimum = orderDetails.getOrderMinimum();
    this.state = APPROVAL_PENDING;
  }
```

Validates the command and returns an OrderCreatedEvent

Apply the OrderCreatedEvent by initializing the fields of the Order.

# Another Example

- Event sourcing version of reviseOrder() and confirmRevision()

**Listing 6.2 The `process()` and `apply()` methods that revise an `Order` aggregate**

```
public class Order {

public List<Event> process(ReviseOrder command) {
  OrderRevision orderRevision = command.getOrderRevision();
  switch (state) {
    case APPROVED:
      LineItemQuantityChange change =
              orderLineItems.lineItemQuantityChange(orderRevision);
      if (change.newOrderTotal.isGreaterThanOrEqual(orderMinimum)) {
        throw new OrderMinimumNotMetException();
      }
      return singletonList(new OrderRevisionProposed(orderRevision,
                           change.currentOrderTotal, change.newOrderTotal));

    default:
      throw new UnsupportedStateTransitionException(state);
  }
}

public void apply(OrderRevisionProposed event) {
   this.state = REVISION_PENDING;
}

public List<Event> process(ConfirmReviseOrder command) {
  OrderRevision orderRevision = command.getOrderRevision();
  switch (state) {
    case REVISION_PENDING:
      LineItemQuantityChange licd =
            orderLineItems.lineItemQuantityChange(orderRevision);
      return singletonList(new OrderRevised(orderRevision,
              licd.currentOrderTotal, licd.newOrderTotal));
    default:
      throw new UnsupportedStateTransitionException(state);
  }
}

public void apply(OrderRevised event) {
  OrderRevision orderRevision = event.getOrderRevision();
  if (!orderRevision.getRevisedLineItemQuantities().isEmpty()) {
    orderLineItems.updateLineItems(orderRevision);
  }
  this.state = APPROVED;
}
```

Verify that the Order can be revised and that the revised order meets the order minimum.

Change the state of the Order to REVISION_PENDING.

Verify that the revision can be confirmed and return an OrderRevised event.

Revise the Order.

# Let's summarize this section...

- Overview
- Event Sourcing persists Aggregates using Events
- Event Represent State Change
- Aggregate Methods are all About Events
- State Change by Applying Event
- Command Method Split
- Aggregate Creation Vs Updation Steps in Event Sourcing
- Event Sourcing based Order Aggregate Example

# Handling Concurrent Updates

# Handling Concurrent Updates using Optimistic Locking

- Two or more requests can simultaneously update the same aggregate
- Developers often use optimistic locking to prevent one transaction from overwriting another's changes.
- **Optimistic locking**
  - uses version column to detect any change in aggregate since it was read
  - Map the aggregate root to a table that has a VERSION column, which is incremented whenever the aggregate is updated
  - An aggregate can  be updated using an UPDATE statement like this:

UPDATE AGGREGATE_ROOT_TABLE SET VERSION = VERSION+1... WHERE VERSION = <original version>

# Publishing Events

# Event Sourcing and Publishing Events

- Saving an event in the event store is an inherently atomic operation
- To publishing event messages that are inserted into the database as part of a transaction, two mechanisms:
  - polling and
  - transaction log tailing
- This permanently stores events in an EVENTS table rather than temporarily saving events in an OUTBOX table and then deleting them after processing

# Polling to Publish Domain Events

- Poll the EVENTS table for new events by a SELECT statement and publish the events to a message broker, the challenge is determining which events are new.
- Superficially appealing approach, record the last eventId that it has processed.
- Then retrieve new events using a query like this:
  - SELECT * FROM EVENTS where event_id > ? ORDER BY event_id ASC
- But transactions can commit in an order that's different from the order in which they generate events.
- As a result, the event publisher can accidentally skip over an event.

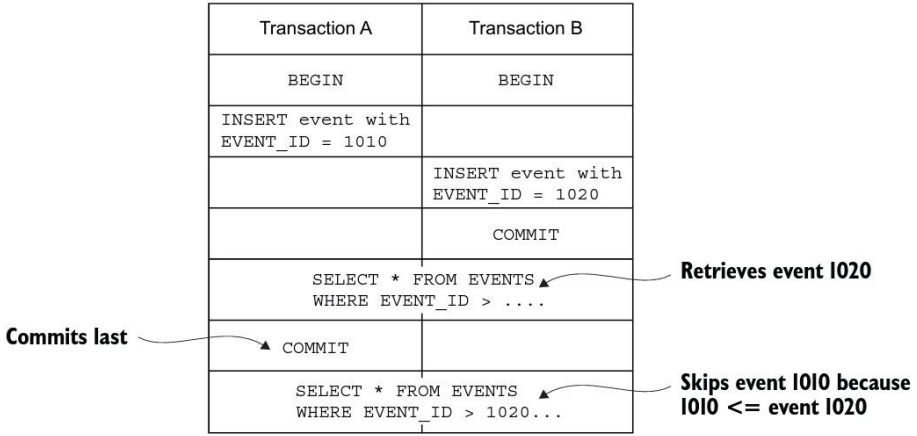



**Figure 6.6 A scenario where an event is skipped because its transaction *A* commits after transaction *B*. Polling sees `eventId=1020` and then later skips `eventId=1010`.**

# Polling to Publish Domain Events

- **Possible solution:** add an extra column to the EVENTS table that tracks whether an event has been published
- The event publisher would then use the following process:
  - Find unpublished events by executing this SELECT statement:
    - SELECT * FROM EVENTS where PUBLISHED = 0 ORDER BY event_id ASC.
  - Publish events to the message broker.
  - Mark the events as having been published:
    - UPDATE EVENTS SET PUBLISHED = 1 WHERE EVENT_ID in.
- This approach prevents the event publisher from skipping events.

# Use Transaction Log Tailing to Reliably Publish Events

- Reads events inserted into an EVENTS table from the database transaction log and publishes them to the message broker
- Guarantees that events will be published
- More performant and scalable

# Snapshoting

# Using snapshots to improve performance

- An aggregate with few state transition can be efficiently loaded from event store
- A long-lived aggregate with large number of events becomes increasingly inefficient to load from the event store
- The solution to this problem is periodically snapshot the aggregate events
- Loading the most recent snapshot and only those events that occurred since the snapshot restores the state of  the aggregate

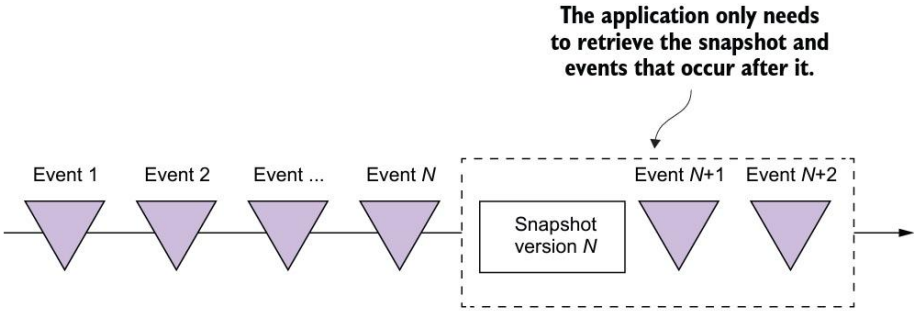


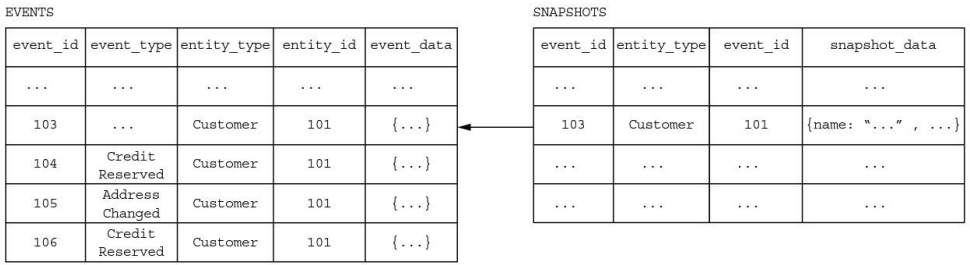

**Figure 6.7 Using a snapshot improves performance by eliminating the need to load all events. An application only needs to load the snapshot and the events that occur after it.**

# Using snapshots to improve performance

With snapshots, aggregate instance is recreated from the snapshot

```
Class aggregateClass = ...;
Snapshot snapshot = ...;
Aggregate aggregate = recreateFromSnapshot(aggregateClass, snapshot);
for (Event event : events) {
  aggregate = aggregate.applyEvent(event);
}
// use aggregate...
```

In below example, customer Service needs to load the snapshot and the events that have occurred after event #103.



EVENTS

| event_id | event_type | entity_type | entity_id | event_data |
|---|---|---|---|---|
| ... | ... | ... | ... | ... |
| 103 | ... | Customer | 101 | {...} |
| 104 | Credit Reserved | Customer | 101 | {...} |
| 105 | Address Changed | Customer | 101 | {...} |
| 106 | Credit Reserved | Customer | 101 | {...} |

SNAPSHOTS

| event_id | entity_type | event_id | snapshot_data |
|---|---|---|---|
| ... | ... | ... | ... |
| 103 | Customer | 101 | {name: "..." , ...} |
| ... | ... | ... | ... |
| ... | ... | ... | ... |

**Figure 6.8 The `Customer Service` recreates the `Customer` by deserializing the snapshot's JSON and then loading and applying events #104 through #106.**

# Idempotency

# Idempotent Message Processing

- A message broker might deliver the same message multiple times
- A message consumer is idempotent if it can safely be invoked with the same message multiple times
- For type of event store:
  - Idempotent message processing in SQL/RDBMS based Event-Store
  - Idempotent message processing in NoSQL based Event-Store

# Idempotent message processing in SQL/RDBMS based Event-Store

- Record ids of processed messages in PROCESSED_MESSAGES table as part of the local ACID transaction
- If ID of a message exists in PROCESSED_MESSAGES table, it’s a duplicate and can be discarded

# Idempotent message processing in NoSQL based Event-Store

Limited transaction model, must use a different mechanism

A message consumer must somehow atomically persist events and record the message ID

A message consumer stores the message's ID in the events that are generated while processing it

It detects duplicates by verifying that none of an aggregate's events contains the message ID

One challenge is that processing a message might not generate any events

The lack of events means there's no record of a message having been processed. A subsequent redelivery and reprocessing of the same message might result in incorrect behavior

# Idempotent message processing in NoSQL based Event-Store

- For example, consider the following scenario:
  1. Message A is processed but doesn't update an aggregate
  2. Message B is processed, and the message consumer updates the aggregate
  3. Message A is redelivered, and because there's no record of it having been processed, the message consumer updates the aggregate.
  4. Message B is processed again
- As a result, redelivery of events results in a different and possibly erroneous outcome
- **One way to avoid this problem:** always publish an event.
- If an aggregate doesn't emit an event, an application **saves a pseudo event** solely to record the message ID
- Event consumers must ignore these pseudo events

# Evolving Domain Events

# Evolving Domain Events

- Event sourcing stores events forever
- On one hand, it provides the application with an audit log of changes that's guaranteed to be accurate
- Aso enables an application to reconstruct the historical state of an aggregate
- On the other hand, it creates a challenge, because the structure of events often changes over time
- An application must potentially deal with multiple versions of events

# Event Schema Evolution

- Event sourcing schema is organized into three levels:
  - Consists of one or more aggregates
  - Defines the events that each aggregate emits
  - Defines the structure of the events

**Table 6.1 The different ways that an application's events can evolve**

| Level | Change | Backward compatible |
|---|---|---|
| Schema | Define a new aggregate type | Yes |
| Remove aggregate | Remove an existing aggregate | No |
| Rename aggregate | Change the name of an aggregate type | No |
| Aggregate | Add a new event type | Yes |
| Remove event | Remove an event type | No |
| Rename event | Change the name of an event type | No |
| Event | Add a new field | Yes |
| Delete field | Delete a field | No |
| Rename field | Rename a field | No |
| Change type of field | Change the type of a field | No |

# Event Schema Evolution

These changes occur naturally as a service's domain model evolves over time:

when a service's requirements change or as its developers gain deeper insight into a domain and improve the domain model.

At the aggregate level, the types of events  emitted by a particular aggregate can change.

Developers can change the structure of an event type by adding, removing, and changing the name or type of a field.

Many of these types of changes are backward-compatible changes.

# Managing Schema Changes through Upcasting

Ordinarily, changes to a database schema are handled using schema migrations using tools such as Flyway

An event sourcing application can use a similar approach to handle non-backward compatible changes

Instead of migrating events to the new schema version, event sourcing frameworks transform events when they’re loaded from the event store

A component commonly called an **upcaster** updates individual events from an old version to a newer version

As a result, the application code only ever deals with the current event schema

# Benefits and Drawbacks of ES

# Benefits of Event Sourcing

- Reliably publishes domain events
- Preserves the history of aggregates
- Mostly avoids the O/R impedance mismatch problem
- Provides developers with a time machine

# Drawbacks of Event Sourcing

- It has a different programming model that has a learning curve.
- It has the complexity of a messaging-based application.
- Evolving events can be tricky.
- Deleting data is tricky.
- Querying the event store is challenging

# Thank You!