

Microservice Architecture (/index.html)

Supported by Kong (<https://konghq.com/>)

 pattern (/tags/pattern)

---

# Pattern: Event sourcing

## Context

A service command typically needs to update the database **and** send messages/events. For example, a service that participates in a saga (/patterns/data/saga.html) needs to atomically update the database and sends messages/events. Similarly, a service that publishes a domain event (domain-event.html) must atomically update an aggregate (aggregate.html) and publish an event.

A service must atomically update the database and send messages in order to avoid data inconsistencies and bugs. However, it is not viable to use a traditional distributed transaction (2PC) that spans the database and the message broker to atomically update the database and publish messages/events. The message broker might not support 2PC. And even if does, it's often undesirable to couple the service to both the database and the message.

But without using 2PC, sending a message in the middle of a transaction is not reliable. There's no guarantee that the transaction will commit. Similarly, if a service sends a message after committing the transaction there's no guarantee that it won't crash before sending the message.

In addition, messages must be sent to the message broker in the order they were sent by the service. They must usually be delivered to each consumer in the same order although that's outside the scope of this pattern. For example, let's suppose that an aggregate is updated by a series of transactions  $T_1$ ,  $T_2$ , etc. This transactions might be performed by the same service instance or by different service instances. Each transaction publishes a corresponding event:  $T_1 \rightarrow E_1$ ,  $T_2 \rightarrow E_2$ , etc. Since  $T_1$  precedes  $T_2$ , event  $E_1$  must be published before  $E_2$ .

## Problem

How to reliably/atomically update the database and send messages/events?

## Forces

- 2PC is not an option
- If the database transaction commits messages must be sent. Conversely, if the database rolls back, the messages must not be sent
- Messages must be sent to the message broker in the order they were sent by the service. This ordering must be preserved across multiple service instances that update the same aggregate.

## Solution

A good solution to this problem is to use event sourcing. Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic. The application reconstructs an entity's current state by replaying the events.

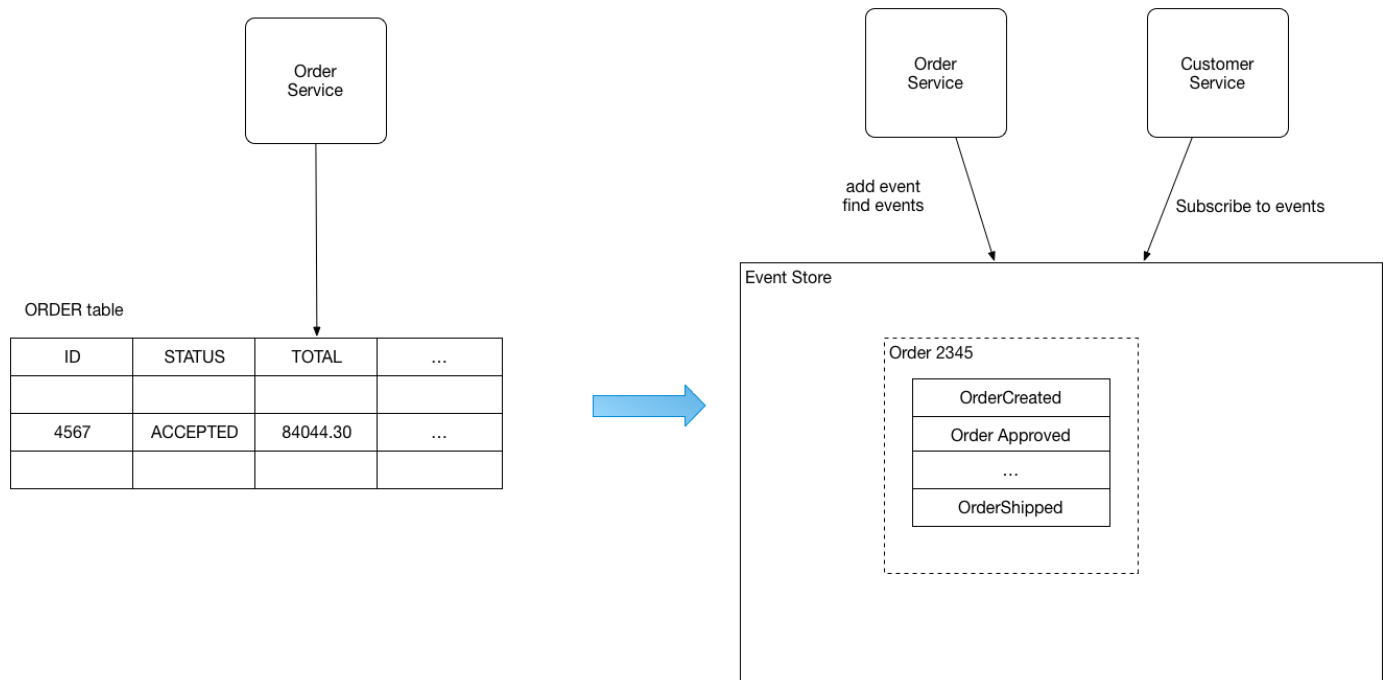
Applications persist events in an event store, which is a database of events. The store has an API for adding and retrieving an entity's events. The event store also behaves like a message broker. It provides an API that enables services to subscribe to events. When a service saves an event in the event store, it is delivered to all interested subscribers.

Some entities, such as a Customer, can have a large number of events. In order to optimize loading, an application can periodically save a snapshot of an entity's current state. To reconstruct the current state, the application finds the most recent snapshot and the events that have occurred since that snapshot. As a result, there are fewer events to replay.

## Example

Customers and Orders (<https://github.com/eventuate-examples/eventuate-examples-java-customers-and-orders>) is an example of an application that is built using Event Sourcing and CQRS ([cqrs.html](https://cqrs.html)). The application is written in Java, and uses Spring Boot. It is built using Eventuate (<http://eventuate.io>), which is an application platform based on event sourcing and CQRS.

The following diagram shows how it persists orders.



Instead of simply storing the current state of each order as a row in an `ORDERS` table, the application persists each `Order` as a sequence of events. The `CustomerService` can subscribe to the order events and update its own state.

Here is the `Order` aggregate:

```
public class Order extends ReflectiveMutableCommandProcessingAggregate<Order, OrderCommand> {

    private OrderState state;
    private String customerId;

    public OrderState getState() {
        return state;
    }

    public List<Event> process(CreateOrderCommand cmd) {
        return EventUtil.events(new OrderCreatedEvent(cmd.getCustomerId(), cmd.getOrderTotal()));
    }

    public List<Event> process(ApproveOrderCommand cmd) {
        return EventUtil.events(new OrderApprovedEvent(customerId));
    }

    public List<Event> process(RejectOrderCommand cmd) {
        return EventUtil.events(new OrderRejectedEvent(customerId));
    }

    public void apply(OrderCreatedEvent event) {
        this.state = OrderState.CREATED;
        this.customerId = event.getCustomerId();
    }

    public void apply(OrderApprovedEvent event) {
        this.state = OrderState.APPROVED;
    }

    public void apply(OrderRejectedEvent event) {
        this.state = OrderState.REJECTED;
    }
}
```

Here is an example of an event handler in the `CustomerService` that subscribes to `Order` events:

```

@EventSubscriber(id = "customerWorkflow")
public class CustomerWorkflow {

    @EventHandlerMethod
    public CompletableFuture<EntityWithIdAndVersion<Customer>> reserveCredit(
        EventHandlerContext<OrderCreatedEvent> ctx) {
        OrderCreatedEvent event = ctx.getEvent();
        Money orderTotal = event.getOrderTotal();
        String customerId = event.getCustomerId();
        String orderId = ctx.getEntityId();

        return ctx.update(Customer.class, customerId, new ReserveCreditCommand(orderTotal, orderId));
    }
}

```

It processes an `OrderCreated` event by attempting to reserve credit for the orders customer.

There are several example applications (<http://eventuate.io/exampleapps.html>) that illustrate how to use event sourcing.

## Resulting context

Event sourcing has several benefits:

- It solves one of the key problems in implementing an event-driven architecture and makes it possible to reliably publish events whenever state changes.
- Because it persists events rather than domain objects, it mostly avoids the object-relational impedance mismatch problem.
- It provides a 100% reliable audit log ([../observability/audit-logging](#)) of the changes made to a business entity
- It makes it possible to implement temporal queries that determine the state of an entity at any point in time.
- Event sourcing-based business logic consists of loosely coupled business entities that exchange events. This makes it a lot easier to migrate from a monolithic application to a microservice architecture.

Event sourcing also has several drawbacks:


- It is a different and unfamiliar style of programming and so there is a learning curve.
- The event store is difficult to query since it requires typical queries to reconstruct the state of the business entities. That is likely to be complex and inefficient. As a result, the application must use Command Query Responsibility Segregation (CQRS) ([cqrs.html](#)) to implement queries. This in turn means that applications must handle eventually consistent data.

## Related patterns

- The Saga ([saga.html](#)) and Domain event ([domain-event.html](#)) patterns create the need for this pattern.
- The CQRS ([cqrs.html](#)) must often be used with event sourcing.
- Event sourcing implements the Audit logging ([../observability/audit-logging](#)) pattern.

# See also

- Eventuate (<http://eventuate.io>), which is a platform for developing applications with Event Sourcing and CQRS
- Articles about event sourcing and CQRS (<http://eventuate.io/articles.html>)
- How Eventuate implements snapshots (<https://blog.eventuate.io/2017/03/07/eventuate-local-now-supports-snapshots/>)

 pattern (/tags/pattern)

Tweet

Follow @MicroSvcArch

Copyright © 2023 Chris Richardson • All rights reserved • Supported by Kong (<https://konghq.com/>).

## ALSO ON MICROSERVICES

4 years ago · 1 comment

Anti-corruption layer

5 years ago · 2 comments

Polling publisher

2

[

c

G

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

2 Share

Best Newest Oldest



This comment was deleted.

P

**pam1970** ➔ Guest

🕒 4 years ago

milanrawal (:

0 0 • Reply • Share ›

PT

**Prabhakar Thopa**

🕒 4 years ago

what is the difference between this pattern and the "Application Events" pattern?

8 0 • Reply • Share ›



**Chris Richardson** Mod ➔ Prabhakar Thopa

🕒 4 years ago

Event Sourcing persists domain objects as a sequences of events.

Application events (aka transactional outbox) uses a database table as a way to publish events as part of a database transaction.

1 0 • Reply • Share ›

R

**Ricardo** ➔ Chris Richardson

🕒 4 years ago

Hi, and what is the difference between Event Sourcing and Saga pattern?

Thank you Chris.

0 0 • Reply • Share ›



**Juanjo Martín** ➔ Ricardo

🕒 3 years ago

I think ES and SAGA are completely different, in fact. While ES is a way to persist domain objects as a sequence of event (an approach completely different to the classical

ORM), SAGA is a way to implement processes in a distributed systems (for example, a business process) as a sequence of steps. SAGA is more related to BPM in that sense, though as it's distributed and you cannot propagate transactions among the steps, you need to implement also compensating actions (compensating steps) in case your processes suffers some kind of unexpected behaviour. Related to microservices, and depending on the case, you can "concentrate" the SAGA in one service, or the SAGA process (its steps) can be distributed in different services that react to the events that happen in the system. A little hard to explain in a few words... :-)

1 0 • Reply • Share ›



**Muzero**

🕒 3 years ago

Not sure if using reflection here is the most elegant solution. Wouldn't have been better to use a 'visitor' pattern?

1 0 • Reply • Share ›



**Antonio Bruno**

🕒 4 years ago

Wouldn't AWS DynamoDB Streams fit for purpose?

1 0 • Reply • Share ›

SM

**Sudhish Madhavan**

🕒 4 years ago

The event store also behaves like a message broker <-- What does this mean in technical terms? Does it require all subscribers to provide a call back? I admit I am missing something

1 0 • Reply • Share ›



**Stuart Wakefield**

➔ Sudhish Madhavan

🕒 4 years ago edited

At the network level: subscribers connect to the event store and events are streamed from the event store to the subscriber as they come in. At the code level: depending on the API provided the subscribers, the subscriber either polls the events periodically (i.e. Kafka), stream the results (i.e. Kafka Streams Client) or register callbacks to receive the events (i.e. Eventuate). The subscriber will typically use these events to build up their own database read-optimized for the kinds of queries that they expect to do. When a query request comes in to the subscriber application, this application can use their view of the events, that they have stored in their database, to respond. An example implementation of Eventuate with Spring JPA: <https://github.com/eventuat...>

0 0 • Reply • Share ›

R

**Ben**

➔ Sudhish Madhavan

🕒 4 years ago

1 year ago

I agree with Stuart. If you want to see how this works, as well as the other principles of event sourcing, you might want to check out this talk: <https://www.youtube.com/watch?v=...> It also goes into more detail on the 'why' and 'how', as well as how you'd build a system like this from the ground up.

0 1 • Reply • Share ›

PH

**Phillip Henry**

🕒 7 months ago

Looking at the diagram, I don't understand how this will "reliably/atomically update the database and send messages/events". You write "Order 2345" to the Event Store but still need to update the Customer Service who has "subscribe(d) to events". What if the process dies after writing to the DB but before sending a message to the subscribers?

0 0 • Reply • Share ›

AH

**Asaf H**

🕒 10 months ago edited

Doesn't the event sourcing db render the db owned by the service redundant then? If it does, and many services now look at a single db for their events reconstruction, doesn't that negate the pattern of a database per service?

0 0 • Reply • Share ›

**Tim Abell**

🕒 3 years ago

I assume 2PC is short for "Two-phase Commit" [https://en.wikipedia.org/wiki/Two-phase\\_commit](https://en.wikipedia.org/wiki/Two-phase_commit)

0 0 • Reply • Share ›

**Gerard Braat**

🕒 4 years ago edited

Hi@Chris Richardson

- thank you for this great article and content on microservices and event sourcing. I am currently involved in setting up a strategic architecture to integrate SAP (master for business accounts and contacts) and Salesforce. We are using Kafka on AWS as our distributed event log. We are facing challenges when new contacts and with a new parent account record are created in SAP. How do we ensure that Salesforce can process these events in the right order as new contact events may arrive earlier at the Kafka bus than the new business account record. Events should be atomic and independent. I am keen to understand what is the best practice and where does the responsibility lie in the end-to-end architecture to process the events in the right order. Thank you!

0 0 • Reply • Share ›

MN

**Magnus Nilsson**

➔ Gerard Braat

🕒 4 years ago

@Gerard Braat Seems you have two problems. If your source system doesn't guarantee ordering (ie if SAP doesn't guarantee in which order it publishes events, ie "arrives earlier at the Kafka bus") you have to set a time horizon for your



"correctness" assumptions (ie we always wait x minutes, hours, days before processing the event log) or implement logic to update/backfill for late events (you have shorter triggers for event processing but you make sure you handle the logic of late events). You could use Flink or Spark on the consuming side using time- and session windows to implement the correct logic. On the pure Kafka side you don't have that many options if you need ordering. You have to partition on a key common to all events (if there is no such key you are out of luck) you need ordering guarantees on. Still, if SAP publishes events out of order they will be ordered out of order in Kafka.

0 0 • Reply • Share ›



**Gerard Braat**

➔ Magnus Nilsson



🕒 4 years ago

Thanks [@Magnus Nilsson](#) - I agree. I have read also that event streaming should not be used for database synchronisation. In stead published events should be business events. I think in my example the business event should be: "New contact created with a new account" The event published by the source should contain just enough information for the consumer(s) to be able to process the event.

0 0 • Reply • Share ›