Microservice Architecture (/index.html)

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

Pattern: Event sourcing

pattern (/tags/pattern)

Context

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

The command 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 The database and/or the message broker might not support 2PC. And even if they do, it’s often undesirable to couple the service to both the database and the message broker.

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 `T1` , `T2` , etc. This transactions might be performed by the same service instance or by different service instances. Each transaction publishes a corresponding event: `T1 -> E1` , `T2 -> E2` , etc. Since `T1` precedes `T2` , event `E1` must be published before `E2` .

Problem

How to atomically update the database and send messages to a message broker?

Forces

- 2PC is not an option. The database and/or the message broker might not support 2PC. Also, it’s often undesirable to couple the service to both the database and the message broker.
- If the database transaction commits then the 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.

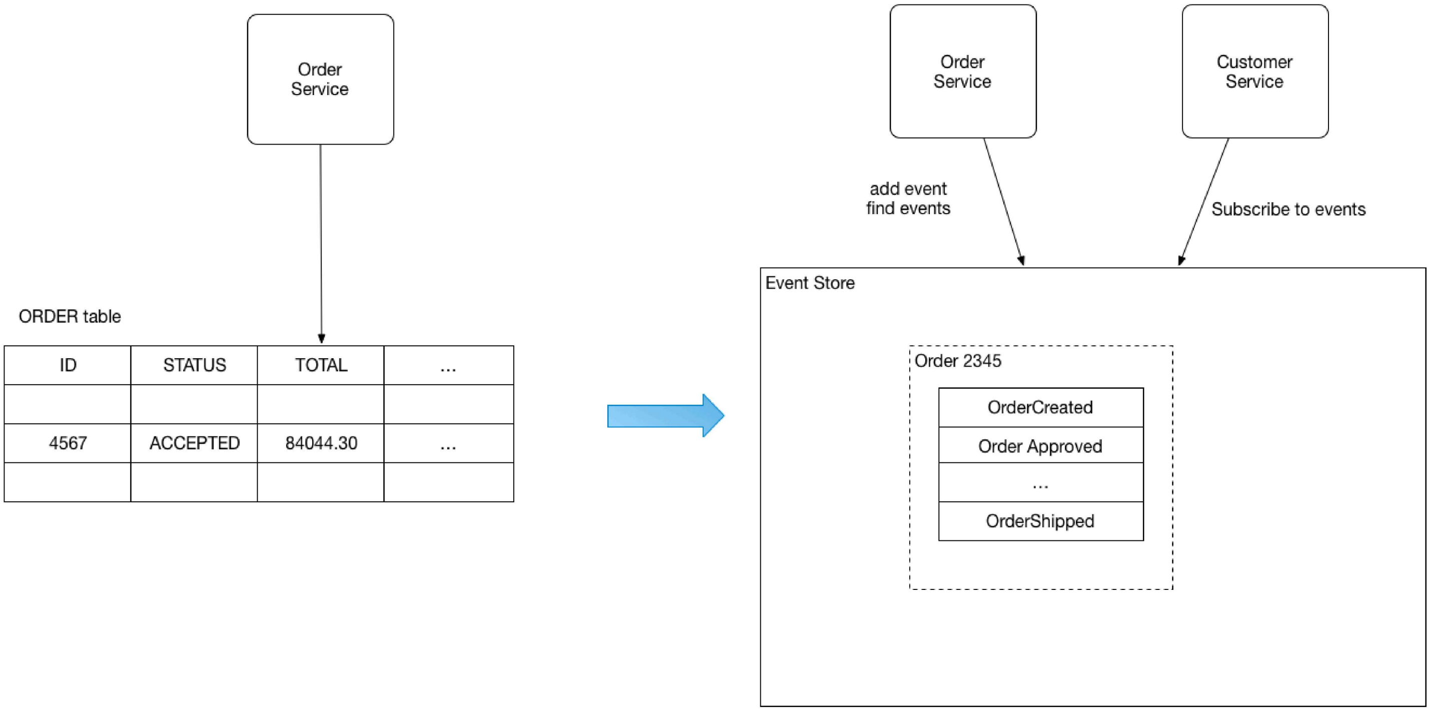
Applications persist events in an event store, which is a database of events. The store has an API for adding and retrieving an entity's events. The event store also behaves like a message broker. It provides an API that enables services to subscribe to events. When a service saves an event in the event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



ID	STATUS	TOTAL	...
4567	ACCEPTED	84044.30	...

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, orderI
d));
  }

}
```

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)

Post

Follow @crichardson

ALSO ON **MICROSERVICES**

Local vs. cloud-based development

a year ago • 1 comment

Local vs. cloud-based development I prefer to develop using IDE that's …

STOP hurting yourself by doing big bang …

3 months ago • 3 comments

STOP hurting yourself by doing big bang modernizations! New …

DDD, necessary but insufficient: physical …

a year ago • 3 comments

DDD, necessary but insufficient: physical design principles for …

In Augus microser

2 years ago

August 29 designing architectu

25 Comments

1 Login ▾

G

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

♡ 3 Share

Best Newest Oldest

P

Prabhakar Thopa

6 years ago

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

8 0 Reply

Chris Richardson Mod → Prabhakar Thopa

6 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

R

Ricardo → Chris Richardson

6 years ago

Hi, and what is the difference between Event Sourcing and Saga pattern? Thank you Chris.

0 0 Reply

Juanjo Martín → Ricardo

5 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

Tim Abell

4 years ago

I assume 2PC is short for "Two-phase Commit" https://en.wikipedia.org/wi...

2 0 Reply

Shubham Jain

a year ago

I didn't understand the part, how event sourcing helps in solving atomicity problem. Isn't this possible that after changing order state in order db, application fails and thus event sourcing fails. How does the architecture solves this problem.

1 0 Reply

César Augusto ➦ Shubham Jain

a year ago edited

I was about the ask the same question. It was not clear how the event sourcing solved the atomicity problem. I'd guess it solves the problem because it make it easier for the application to recover when something unexpected happens, for example, if the app stores the event into the database and it doesn't produce the message, the whole operation could be retried and it would store the event again and produce the message (correctly this time) without affecting the final result - as the event can be stored several time, different than an object (that would be wrongly duplicated). For me this is more related to immutability than event sourcing, so I am not sure if my assumption is right. Curious to see what other people understood.

1 0 Reply

Orkan Bakis ➦ César Augusto

4 months ago

It ensures that every state change in a system is captured as a distinct, immutable event.

0 0 Reply

Phillip Henry

2 years 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?

1 0 Reply

Muzero

5 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

Antonio Bruno

6 years ago

Wouldn't AWS DynamoDB Streams fit for purpose?

1 0 Reply

Sudhish Madhavan

6 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 somthing

1 0 Reply

Stuart Wakefield → Sudhish Madhavan

6 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

Ben → Sudhish Madhavan

6 years 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/wat.... It also goes into more detail on they 'why' and 'how', as well as how you'd build a system like this from the ground up.

0 1 Reply

Pavel Levchuk

9 months ago

9 months ago

It contains an overview of what EventSourcing is, but has some **completely wrong** Problem/Forces/Resulting statements.

Whoever reads about EventSourcing from this source first - will have so complicated and incorrect picture, that he/she won't be able to use it.

Event Sourcing is not about Event-Driven Architecture at all.

Event Driven Architecture is about how services communicate with each other. It's a communication concern.
Event Sourcing lies in Persistence layer and it means that Events are **the source of truth** for a particular Aggregate. It's a persistence concern.

> It has nothing to do with "2PC is not an option.

You use Transactional Outbox for this one.

> "It solves one of the key problems in implementing an event-driven architecture and makes it possible to reliably publish events whenever state changes."

see more

0 0 Reply

César Augusto

a year ago

How event sourcing is solving the atomicity problem?

0 0 Reply

Asaf H

3 years 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

Gerard Braat

5 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 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

Magnus Nilsson Gerard Braat

5 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

Gerard Braat Magnus Nilsson

5 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

Alan Boshier

6 years ago

I'm not clear about where entity id's come from in an event sourcing architecture.

In the event handler implementation `EventHandlerContext.getEntityId()` is called to get the order id. Where was that order id initialized (presumably as part of the `OrderCreatedEvent apply()` but not explicitly?).

0 0 Reply

عبدالإله التميمي Alan Boshier

5 years ago

using AggregateRepository(https://github.com/eventuat... saving the object the type of returned object is EntityWithIdAndVersion(https://github.com/eventuat... which persisting the id and the version.

0 0 Reply

Kishore

6 years ago

When you say Event Sourcing persists domain objects as a sequences of events. In this case are we saying order service will insert a new row for every event, 1. order received with order id 1234 => insert in order table and generate an Order created event 2. order updated with order id 1234 => insert in order table but generate Order updated event ? One more question when you mention, service need to atomically update the database and publish messages/events, how can you update atomically without having a distributed transaction between a database and broker. 1. you insert into the table and you publish => if publish fails then rollback the database 2. If insert fails then obviously don't publish. When you say atomically update, are you saying sending an event to own service in which order service will consume order created event which then inserts into the order table ? Also is event store separate database instance then where the order table resides? Thanks

0 0 Reply

Ben

6 years ago