Pattern: Event sourcing

Context
A service typically need to atomically update the database and publish messages/events. For example, perhaps it uses the Saga pattern. In order to be reliable, each step of a saga must atomically update the database and publish messages/events. Alternatively, it might use the Domain event pattern, perhaps to implement CQRS. In either case, it is not viable to use a distributed transaction that spans the database and the message broker to atomically update the database and publish messages/events.

Problem
How to reliably/atomically update the database and publish messages/events.

Forces
2PC is not an option

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.

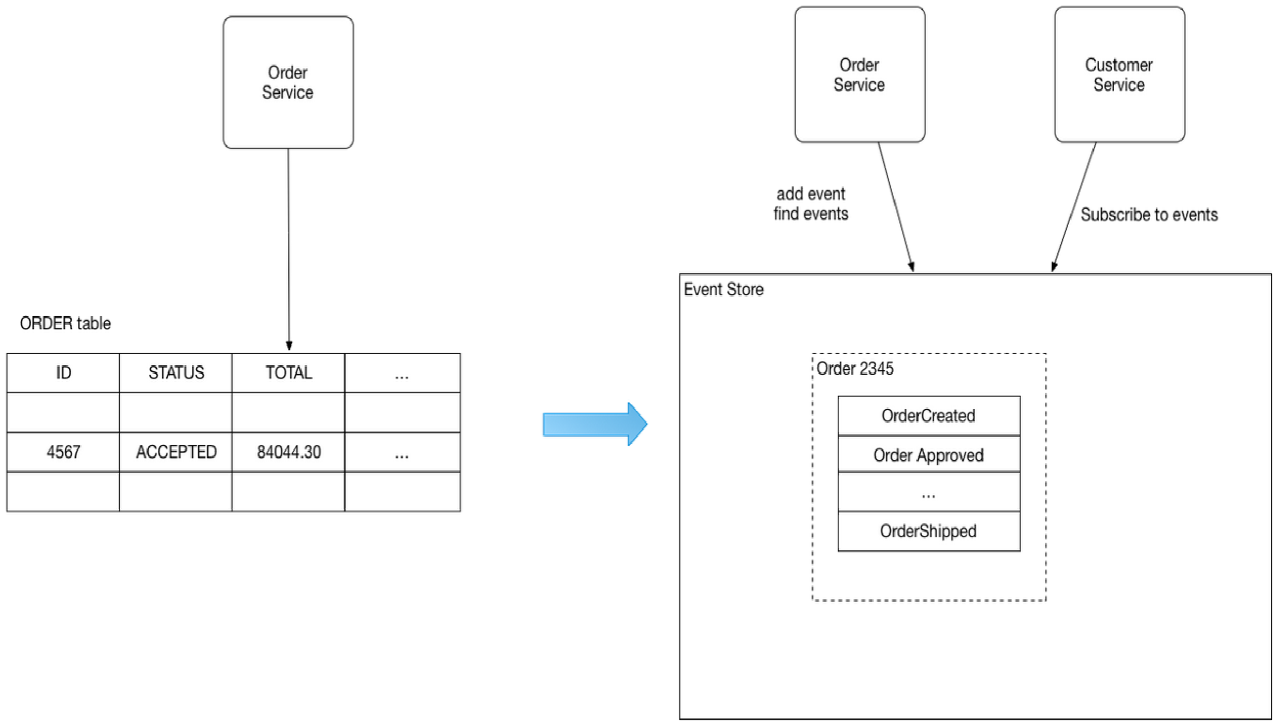
Example
Customers and Orders is an example of an application that is built using Event Sourcing and CQRS. The application is written in Java, and uses Spring Boot. It is built using Eventuate, which is an application platform based on event sourcing and CQRS.

The following diagram shows how it persist 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;
}
```



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

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 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 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) to implement queries. This in turn means that applications must handle eventually consistent data.