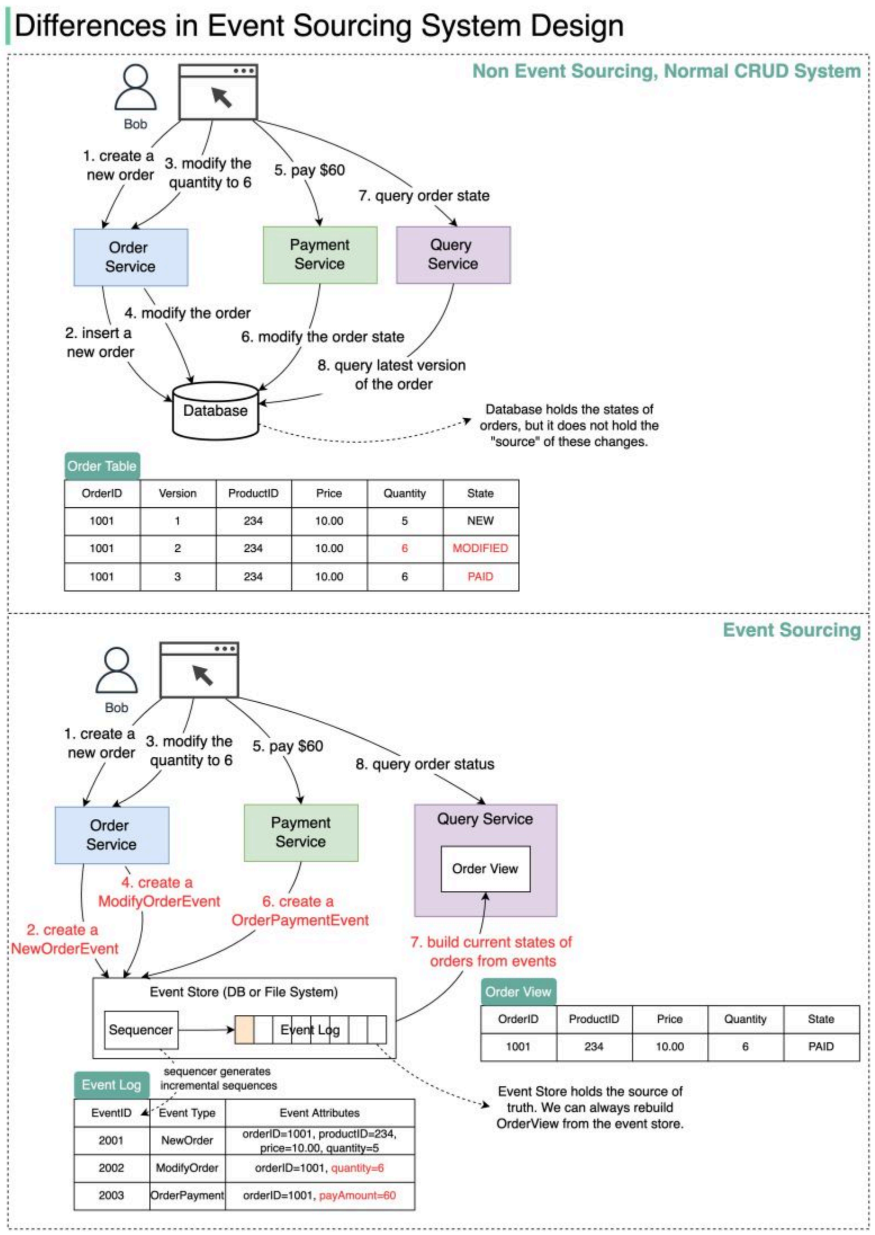
Differences in Event SOurcing System Design

How do we design a system using the **event sourcing** paradigm? How is it different from normal system design? What are the benefits? We will talk about it in this post.

The diagram below shows the comparison of a normal CRUD system design with an event sourcing system design. We use an e-commerce system that can place orders and pay for the orders to demonstrate how event sourcing works.


Differences in Event Sourcing System Design
Non Event Sourcing, Normal CRUD System
Bob
1. create a new order
3. modify the quantity to 6
5. pay $60
7. query order state
Order Service
Payment Service
Query Service
2. insert a new order
4. modify the order
6. modify the order state
8. query latest version of the order
Database
Database holds the states of orders, but it does not hold the "source" of these changes.
Order Table
Event Sourcing
Bob
1. create a new order
3. modify the quantity to 6
5. pay $60
8. query order status
Order Service
Payment Service
Query Service
Order View
2. create a NewOrderEvent
4. create a ModifyOrderEvent
6. create a OrderPaymentEvent
7. build current states of orders from events
Event Store (DB or File System)
Sequencer
Event Log
sequencer generates incremental sequences
Event Store holds the source of truth. We can always rebuild OrderView from the event store.
Order View
Event Log

Order Table

OrderID	Version	ProductID	Price	Quantity	State
1001	1	234	10.00	5	NEW
1001	2	234	10.00	6	MODIFIED
1001	3	234	10.00	6	PAID

Order View

OrderID	ProductID	Price	Quantity	State
1001	234	10.00	6	PAID

Event Log

EventID	Event Type	Event Attributes
2001	NewOrder	orderID=1001, productID=234, price=10.00, quantity=5
2002	ModifyOrder	orderID=1001, quantity=6
2003	OrderPayment	orderID=1001, payAmount=60

The event sourcing paradigm is used to design a system with determinism. This changes the philosophy of normal system designs.

How does this work? Instead of recording the order states in the database, the event sourcing design persists the events that lead to the state changes in the event store. The event store is an append-only log. The events must be sequenced with incremental numbers to guarantee their ordering. The order states are rebuilt from the events and maintained in OrderView. If the OrderView is down, we can always rely on the event store which is the source of truth to recover the order states.

Let's look at the steps in detail.

- Non-Event Sourcing

  Steps 1 and 2: Bob wants to buy a product. The order is created and inserted into the database.

  Steps 3 and 4: Bob wants to change the quantity from 5 to 6. The order is modified with a new state.

  Steps 5 and 6: Bob pays $60 for the order. The order is complete and the state is Paid.

  Steps 7 and 8: Bob queries the latest order state. Query service retrieves the state from the database.

- Event Sourcing

  Steps 1 and 2: Bob wants to buy a product. A NewOrderEvent is created, sequenced and stored in the event store with eventID=2001.

  Steps 3 and 4: Bob wants to change the quantity from 5 to 6. A ModifyOrderEvent is created, sequenced, and persisted in the event store with eventID=2002.

  Steps 5 and 6: Bob pays $60 for the order. An OrderPaymentEvent is created, sequenced, and stored in the event store with eventID=2003. Notice the different event types have different event attributes.

  Step 7: OrderView listens on the events published from the event store, and builds the latest state for the orders. Although OrderView receives 3 events, it applies the events one by one and keeps the latest state.

  Step 8: Bob queries the order state from OrderService, which then queries OrderView. OrderView can be in memory or cache and does not need to be persisted, because it can be recovered from the event store.

Over to you: Which type of system is suitable for event sourcing design? Have you used this paradigm in your work?