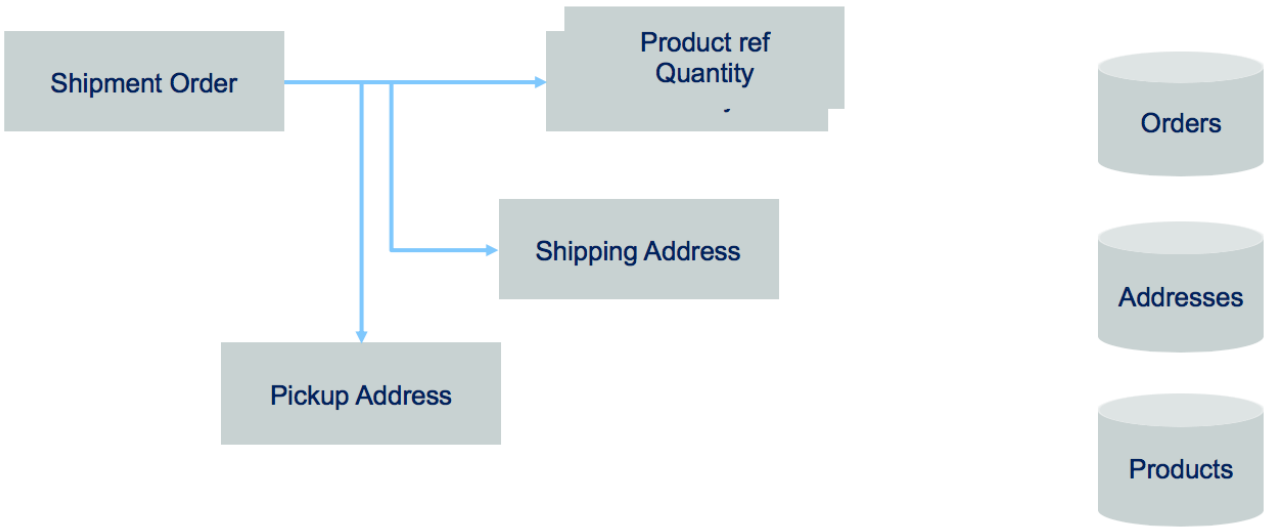
← Architecture Center (/cloud/architecture/architectures)

Event sourcing pattern

Use an event sourcing pattern to persist the state of a business entity as a sequence of state-changing events.

Most business applications are state-based persistent: any update changes the previous state of business entities. The database keeps the last committed update. But for some business applications, you need to explain how they reached their current state. For that purpose, the application must keep a history of business facts.

Traditional domain-oriented implementations build a domain data model and map it to a relational database management system (RDBMS). In the Order model example, the database record keeps the last state of the Order entity. The addresses and the last ordered items are in separate tables.

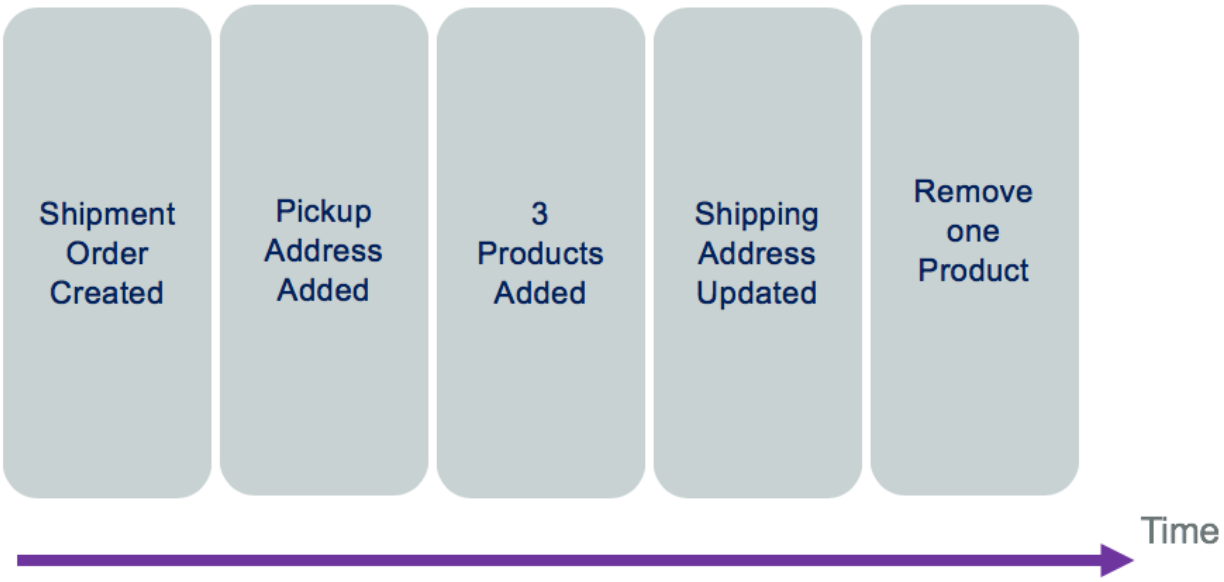


If you need to implement a query that looks at what happened to the order over time, you need to change the model and add historical records. Basically, you build a log table.

Designing a service to manage the lifecycle of this order usually adds a delete operation to remove data. For legal reasons, most businesses don't remove data. For example, a business ledger must include a new record to compensate a previous transaction. Previously logged transactions can't be erased. You can always understand what was done in the past. Most business applications need to keep this capability.

Solution and pattern

Event sourcing persists the state of a business entity, such an order, as a sequence of state-changing events or immutable "facts" that are ordered over time.



When the state of a system changes, an application issues a notification event of the state change. Any interested parties can become consumers of the event and take required actions. The state-change event is immutable and is stored in an event log or event store in time order. The event log becomes the principal source of truth. You can re-create the system state from a point in time by reprocessing the events. The history of state changes becomes an audit record for the business. It is often a useful source of data for business analysts to gain insights into the business.

You can see that the "removing an item" event in the log is a new event. With this capability, you can count how often a specific product is removed from the shopping cart.

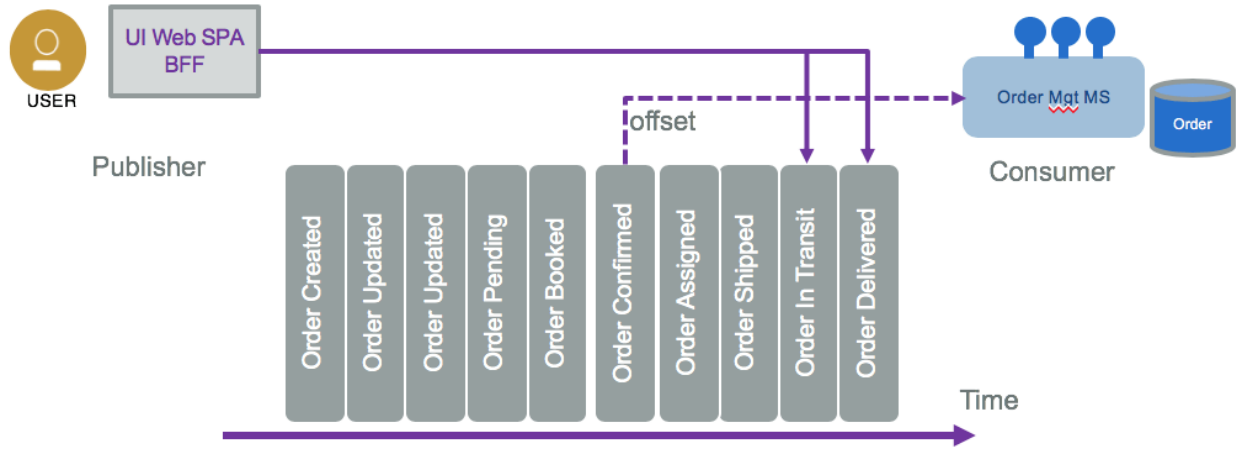
In some cases, the event sourcing pattern is implemented completely within the event backbone. Kafka topics and partitions are the building blocks for event sourcing. However, you can also consider implementing the pattern with an external event store, which provides optimizations for how the data can be accessed and used.

An event store needs to store only three pieces of information:

- The type of event or aggregate
- The sequence number of the event
- The data as a serialized entity

More data can be added to help with diagnosis and audit, but the core function requires only a narrow set of fields. The result a simple data design that you can heavily optimize to append and retrieve sequences of records.

With a central event log, as provided by Kafka, producers append events to the log and consumers read them from an offset (a sequence number).



To get the final state of an entity, the consumer needs to replay all the events. The consumer replays the changes to the state from the last committed offset or from the last snapshot or the origin of "time".

IBM® Db2® Event store (https://www.ibm.com/products/db2-event-store) can provide the handlers and event store that are connected to the backbone and can provide optimization for downstream analytical processing of data.

Advantages

The main goal of event sourcing to understand what happens to a business entity over time. You can do a few interesting things:

- Rebuild the data view within a microservice after it fails by reloading the event log
- Because events are ordered with time, you can apply complex event processing with temporal queries, time window operations, and looking at non-events

- Reverse the state and correct data with new events

Considerations

When you replay the events, avoid generating side effects. A common side effect is to send a notification on state change to other consumers. The consumer of events must be adapted to the query and business requirements. For example, if the code needs to answer the question: "What happened to the order ID 75 over time?", no side effect exists. Only a report can be created each time that the consumer runs.

It might take too long to replay hundreds of events. In that case, you can use a snapshot to capture the current state of an entity and then replay events from the most recent snapshot. This optimization technique isn't needed for all event-sourcing implementations. When state change events are in low volume, you don't need snapshots.

Event sourcing is an important pattern to support eventual data consistency between microservices and for data synchronization between systems as the event store becomes the source of truth. Another common use case that event sourcing helps is when developers push a new code version that corrupts the data. Being able to see what was done on the data and being able to reload from a previous state helps them to fix problems.

Command sourcing

Command sourcing is a pattern that is like event sourcing, but the commands that modify the states are persisted instead of the events. Commands can be processed asynchronously, which is relevant when the command execution takes a long time.

One derived challenge is that the command can be run many times, especially when failures occur. That's why the command must be idempotent, where making multiple identical requests has the same effect as making a single request. You must also validate the command to avoid keeping wrong commands in queue. For example, the `AddItem` command is becoming `AddItemValidated`. After the command is persisted to a database, it becomes an event as `ItemAdded`. Mixing command and event sourcing is a common practice.

Business transactions aren't ACID (atomic, consistent, isolated, and durable) and span many services. They're more like a series of steps, where each step is supported by a microservice that is responsible to update its own entity.

The event backbone must guarantee that events are delivered at least once and that the microservices are responsible to manage their offset from the stream source and handle inconsistency by detecting duplicate events.

At the microservice level, updating data and emitting events must be an atomic operation to avoid inconsistency if the service fails after the update to the data source and before it emits the event. One solution is to use an event table that is added to the microservice data source and an event publisher that reads that table regularly and changes the state of the event after it is published. Another solution is to have a database transaction log reader or miner that is responsible to publish events on a new row that is added to the log.

One other approach to avoid the two-phase commit and inconsistency is to use an event store or event sourcing pattern to track what is done on the business entity with enough information to rebuild the data state. Events are becoming facts that describe state changes that are done on the business entity.

What's next

- Review the strangler pattern (/cloud/architecture/architectures/event-driven-strangler-pattern).
- Read these resources:
  - Martin Fowler's article on event sourcing (https://martinfowler.com/eaaDev/EventSourcing.html)
  - Chris Richardson's article on the event sourcing pattern (https://microservices.io/patterns/data/event-sourcing.html)
- Watch Greg Young on event sourcing (https://www.youtube.com/watch?v=8JKjvY4etTY) at the goto; conference.
- Review an implementation of event sourcing. All the microservices in the Reefer container shipment solution (https://ibm-cloud-architecture.github.io/refarch-kc/) are using event sourcing. The solution uses Kafka with long persistence. The order management service (https://github.com/ibm-cloud-architecture/refarch-kc-order-ms) uses CQRS with event sourcing. An integration test (https://ibm-cloud-architecture.github.io/refarch-kc/integration-tests/happy-path/) validates the pattern.