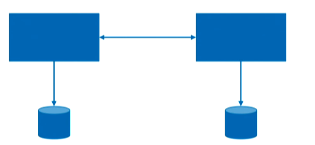
# Data Consistency in Microservice Architecture

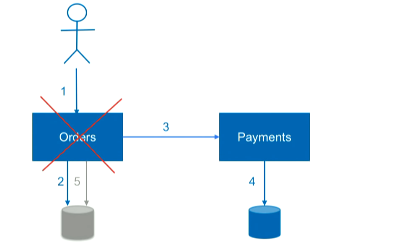
Whenever we design systems as independent services data consistency problem will arise because they have different technologies, different bounded contexts even owned by different teams.

When we save the data in one service then try to save in another one many things can go wrong Network, Service got restarted and data is gone.



So those are the scenarios we are trying to solve.

## In a start where things can go wrong



In above case **Order** service got the order saves it calls **Payment** which happened success and after that something happened we restarted the service or something and order status was **NOT CNF.**

So this order will never go to **Delivery.**

## Solution

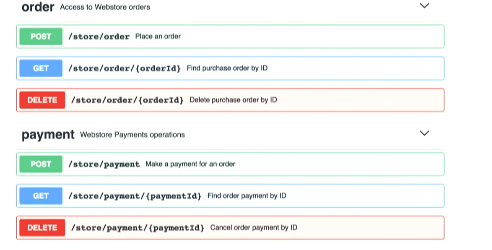
### SAGA pattern

SAGA is a popular pattern to address this consistency problem. Let’s look more.

How we can present that there would be two operations

* Compensating operation to our Order and Payment
* Reconciliation

In **Compensation** we will have API to remove the change the status

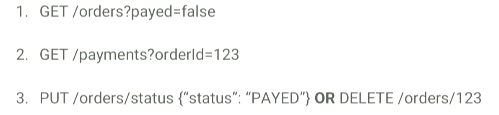


In **Reconciliation** we do check the summary of the different entities to ensure they match or not. This can check if all the services are in sync or any single message lost. This can be done as end-of-day procedure (Scheduled implementation) or Proactive even-based (more frequently)

**How we can use the same in previous failure case**

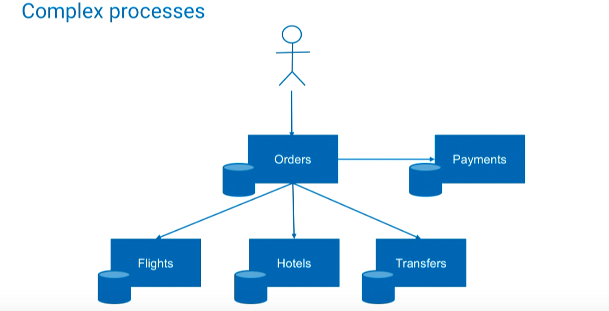
Once customer’s order status failed and received the order and customer reloads the cart or status to see. From back end we can check all the order status of the customer and if one of those already paid but status is Failed. Then we got to Payment service with specific order id to get more status and if we see “status”:”PAYED” then we can make the Order status in order service CONFIRMED.

Or if payment is not successful we can start compensating operation. That is deleting that particular Order



This process is bit delayed but we are syncing our service and making customer happy.

**Drawback of this solution is** when our system is more complex where many micro services are communicating with each other Ex. Complete TRIP booking (Ordering Trip which consists Flight, Hotel, Transfers)

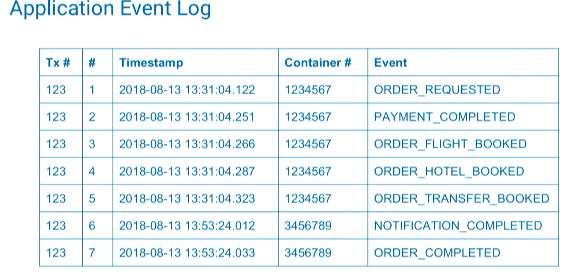


In this kind of situation we can try with **Application Event Log** technique.

### Application Event Log

This technique is similar to Logging we deal everyday but more towards Application aware logging.

We can track in one place all the steps of our distributed business process that needs to be atomic from customer point of view.



Here if any of process got failed. We can go through the tables get the status and resume our process and apply the same technique either confirm and finish the business operation or compensate deleting the order.

### Challenges

* Orchestration or Choreography
* Periods of inconsistency
* Implementation complexity

This may looks like Orchestration but not necessary to be can be **Choreography** and distributed or application log will be the one single source of truth.

We need or will have intermediate state (ex. In progress) which is intermediate state of inconsistency. Which means orders are not paid should not be confirmed so the Order service needs to know the intermediate state.

It is complex because we need to design lot of compensation logics. Also going to a particular state and resume operation also needs technical design.

### BASE and ACID

2-phase commit tries to apply **ACID** guarantee through transaction coordinator. Which means either all are saved/committed or roll-back.

In microservice we have multiple technologies, network communication which could create latency.

So in microservice we try for **BASE** over **ACID**

**B**asic **A**vailability

**S**oft state

**E**ventual consistency

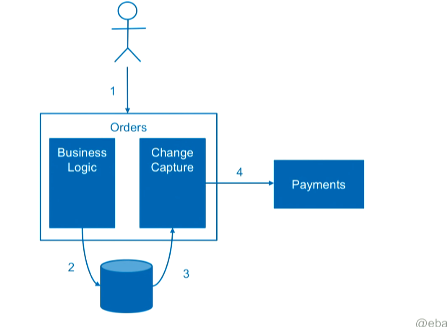
**BASE** is basically accepting the fact that in some of their services for some of the period will have intermediate state (IN PROGRESS) that we need to treat differently.

### Change Data Capture (CDC)

We need Compensating solution because the fact that we were making atomic store in two different data storage differently.

So what if we make a change in one process in only one database. And this solution can be named to **Change Data Capture (CDC).**

How does it look like?

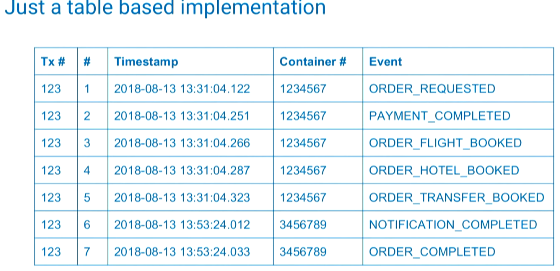


Here instead of making 2 different processes making change to Order and Payment together. We first save the Order then a separate process will pick the change data and process the Payment.

How is it different, here we are always making change in single data store. So **either it goes in one record saved or not saved** so we don’t need to manually reconcile the record as we have single record and single data store. And eventually change capture process will always pick this change.

How this can be implemented?

We can simply have a table based implementation as same Application log table.



By the way with multiple participants and one single event log is **really scalable.**

**Problem**

If we poll the table we will get **Load distribution** problem. When we have more than one service we should poll the table and table distribute the work.

**Solution**

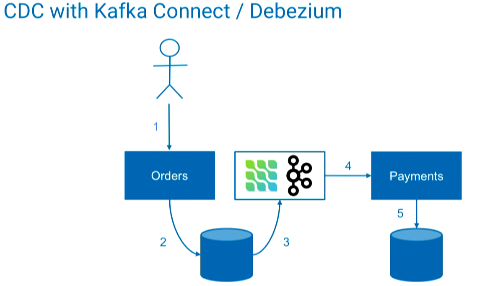
There are some technique and solutions on how to make is simpler for example many databases they offer **Operation Log** this log is helpful for distribution of changes from **Master** to **Read replicas** or can be used by our application.

We don’t have to poll the database continuously; change will be propagated through change log or operation log notifications.

* Mongo DB Oplog
* Mysql changelog
* PostgreSQL Changelog

### CDC with Kafka Connect

We can use Kafka connect to **stream** our changes from our database to Kafka topic



It saves time because it takes care of **work distribution** we can create customized mapping and it takes care of capturing change data from our database.

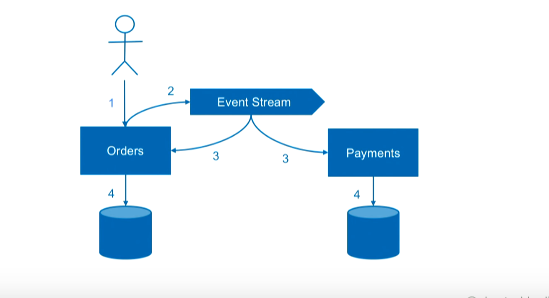
**Challenges with CDC**

* Business Logic Split
* Self-implemented vs Packaged

### Event First

This is one of the approach. This is a way of event driven approach.

We make it in reverse way.

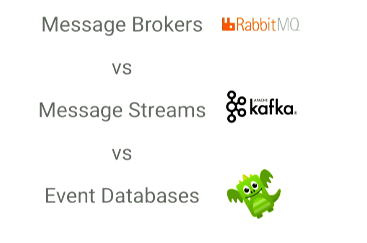


First order is receive. Then instead writing to database change we publish an event saying order was received.

This event stream is a single source of truth, could be a Kafka topic, message broker or database.

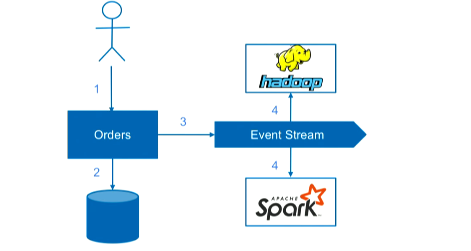
Then the other services including Order service pick up this change and create a record to its database.

We can rely on it, if we say “save” both Order and Payment receives this change and process it.



### Accepting Inconsistency

It’s always not needed full 100% consistency. There could be some part or system in our application like analytics, recommendation which doesn’t necessarily require consistency if we save a record that will be fine.



### Consistency By Design

Approaching consistency from very beginning of our designing when we are designing for Domain boundaries and Bounded contexts.

This which needs to stay consistent may stay in one microservice as well.

(Sometimes it’s ok to merge many services if it’s too challenging to make them consistent)