Maroc Ynov Campus
M2 – Master Développement Web et Mobile

Microservices Architecture

Event-Driven Data Management

by
Driss ALLAKI

Event-Driven Data Management

The general goal

Implementing transactions and queries
in a microservice architecture
using asynchronous messaging

Event-Driven Data Management

- Problem Statement : **Distributed Data Management** (*Transactions* and *Queries*)
- One solution of the **Transactions** challenge:
  - The **Saga** pattern
  - Coordination sagas
    - Choreography-based saga
    - Orchestration-based saga
- Two Solutions of the **Queries** challenge :
  - Solution 1 : The **API Composition** pattern
  - Solution 2 : The **CQRS** pattern

Problem Statement : The problem of Distributed Data Management

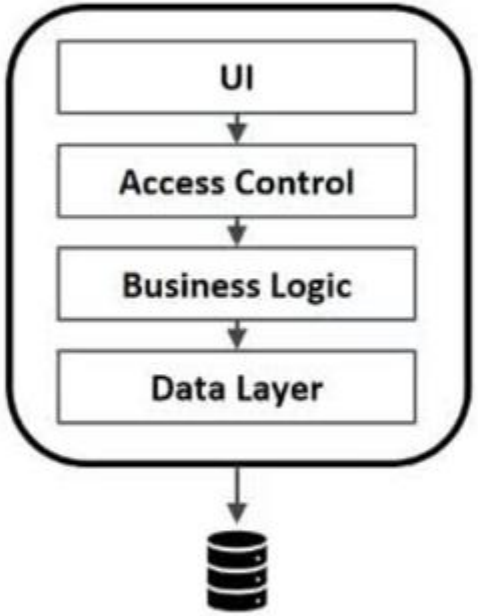
Data Management in Monolith Applications

- A **monolithic application** typically has a **single relational database**. A key benefit of using a relational database is that your application can use **ACID transactions**, which provide some important guarantees
    - ***Atomicity*** : *Changes are made atomically (all or nothing)*
    - ***Consistency :*** *The state of the database is always consistent*
    - ***Isolation :*** *Even though transactions are executed concurrently it appears they are executed serially*
    - ***Durability :*** *Once a transaction has committed it is not undone*

- As a result, the application can simply :
    - begin a transaction
    - change (insert, update, and delete) multiple rows
    - and commit the transaction.

- Another great benefit of using a relational database is that it provides **SQL**.

- You can easily write a query that **combines data from multiple tables**.



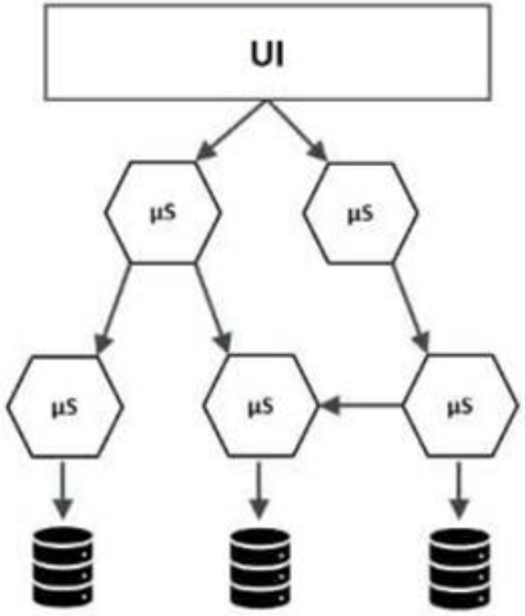
Problem Statement : The problem of Distributed Data Management

Data Management in a Distributed Microservices Architecture

- Data access becomes much more complex when we move to a **microservices architecture**.
  - The **data** owned by each microservice is **private** to that microservice and can only be accessed via its API *(the **Database per service** pattern)*
    (If multiple services access the same data, schema updates require time-consuming, coordinated updates to all the services => Tight coupling and dependency between services)
  - Different microservices often use **different kinds of databases** *(the **polyglot persistence** approach)*

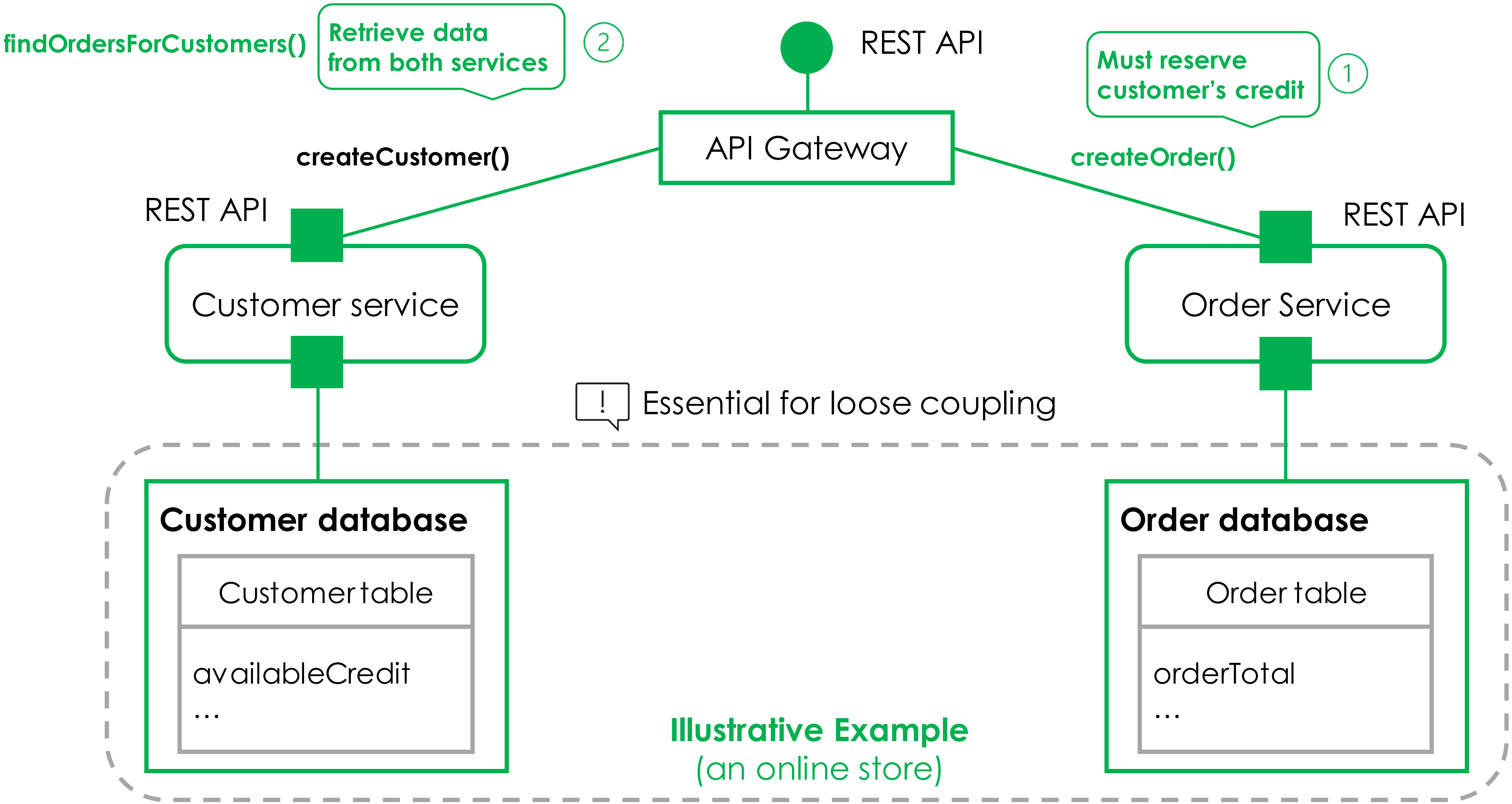
⊕ **A partitioned, polyglot-persistent architecture for data storage** has many benefits, including :
- **loosely coupled services**
- better **performance**
- better **scalability**

⊖ It introduces some **distributed data management challenges**.



Problem Statement : The problem of Distributed Data Management

How to achieve consistency across multiple microservices



Problem Statement : The problem of Distributed Data Management

How to achieve consistency across multiple microservices

Illustrative Example (an online store)

1 **createOrder()**

Must reserve customer's credit

```
BEGIN TRANSACTION
…
SELECT ORDER_TOTAL
FROM ORDER WHERE CUSTOMER_ID = ?
…
SELECT CREDIT_LIMIT
FROM CUSTOMER WHERE CUSTOMER_ID = ?
…
INSERT INTO ORDERS …
…
COMMIT TRANSACTION
```

No ACID transactions that span services

- Distributed transaction
- ORDER : private to the Order service
- CUSTOMER : private to the Customers service

2 **findOrdersForCustomers()**

Retrieve data from both services

```
SELECT *
FROM CUSTOMER c, ORDER o
WHERE c.id = o.ID
AND c.id = ?
```

Querying across services is not straightforward

- ORDER : private to the Order service
- CUSTOMER : private to the Customers service

Problem Statement : The problem of Distributed Data Management

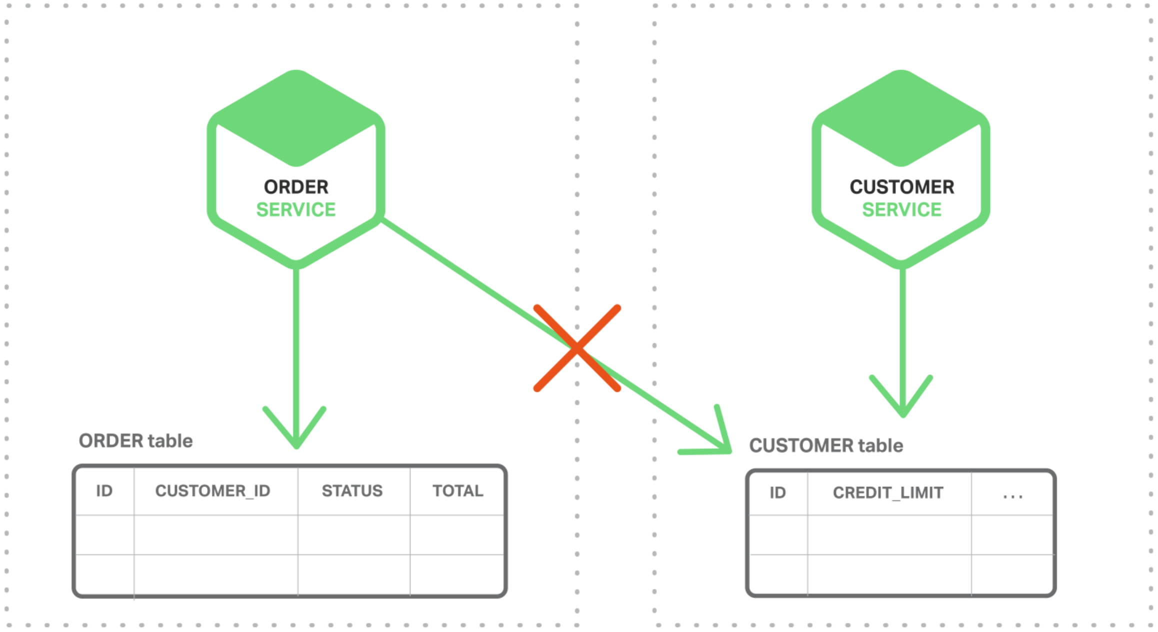
How to achieve consistency across multiple microservices

Illustrative Example : (an online store)

- *The Customer Service maintains information about customers, including their credit lines.*
- *The Order Service manages orders and must verify that a new order doesn't exceed the customer's credit limit.*

In a microservices architecture the ORDER and CUSTOMER tables are private to their respective services.

(The Order Service cannot access the CUSTOMER table directly. It can only use the API provided by the Customer Service.)



Challenges :

1. **How to implement transactions that span services?**
2. **How to implement queries that retrieve data from multiple services ?**

Maintaining data consistency across services and databases is of utmost importance

Event-Driven Data Management

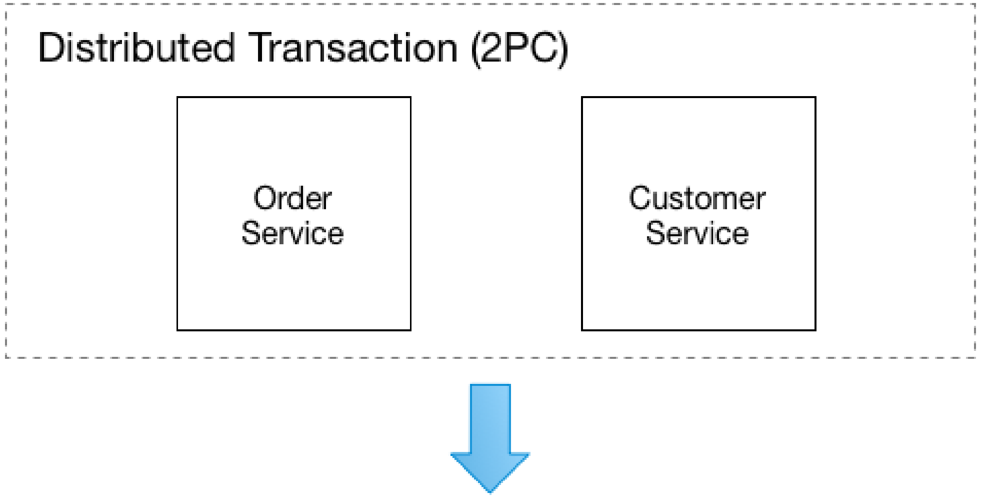
- Problem Statement : **Distributed Data Management** (*Transactions* and *Queries*)
- One solution of the **Transactions** challenge:
    - The **Saga** pattern
    - Coordination sagas
        - Choreography-based saga
        - Orchestration-based saga
- Two Solutions of the **Queries** challenge :
    - Solution 1 : The **API Composition** pattern
    - Solution 2 : The **CQRS** pattern

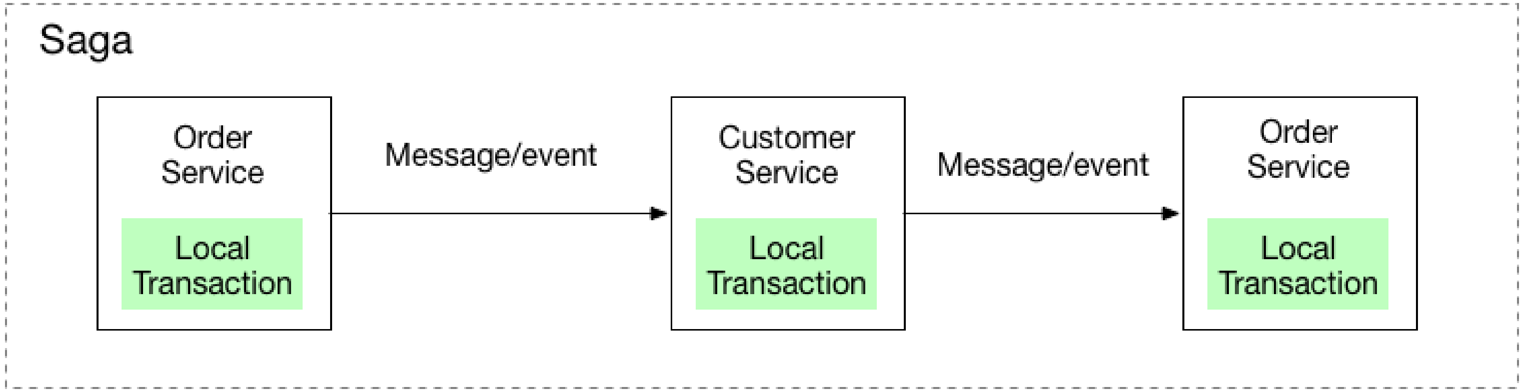
Solution : Managing transactions with SAGAs

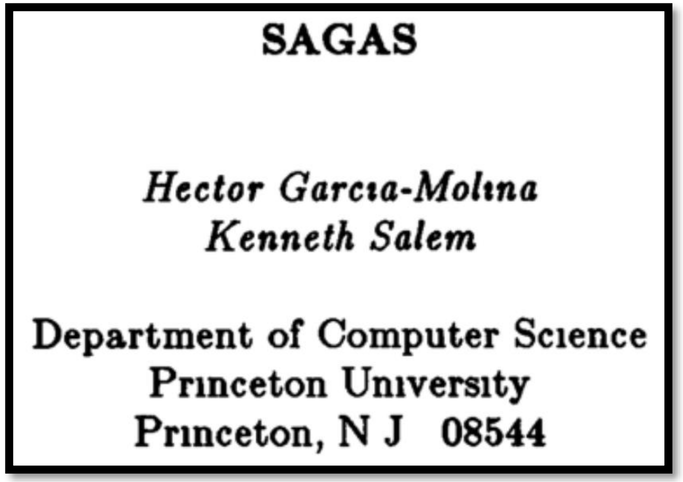
2PC is not an option for Microservices

Read more about
[2PC (Two-Phase Commit protocol)]

Use SAGAs pattern instead of 2PC







SAGAS

Hector Garcia-Molina
Kenneth Salem

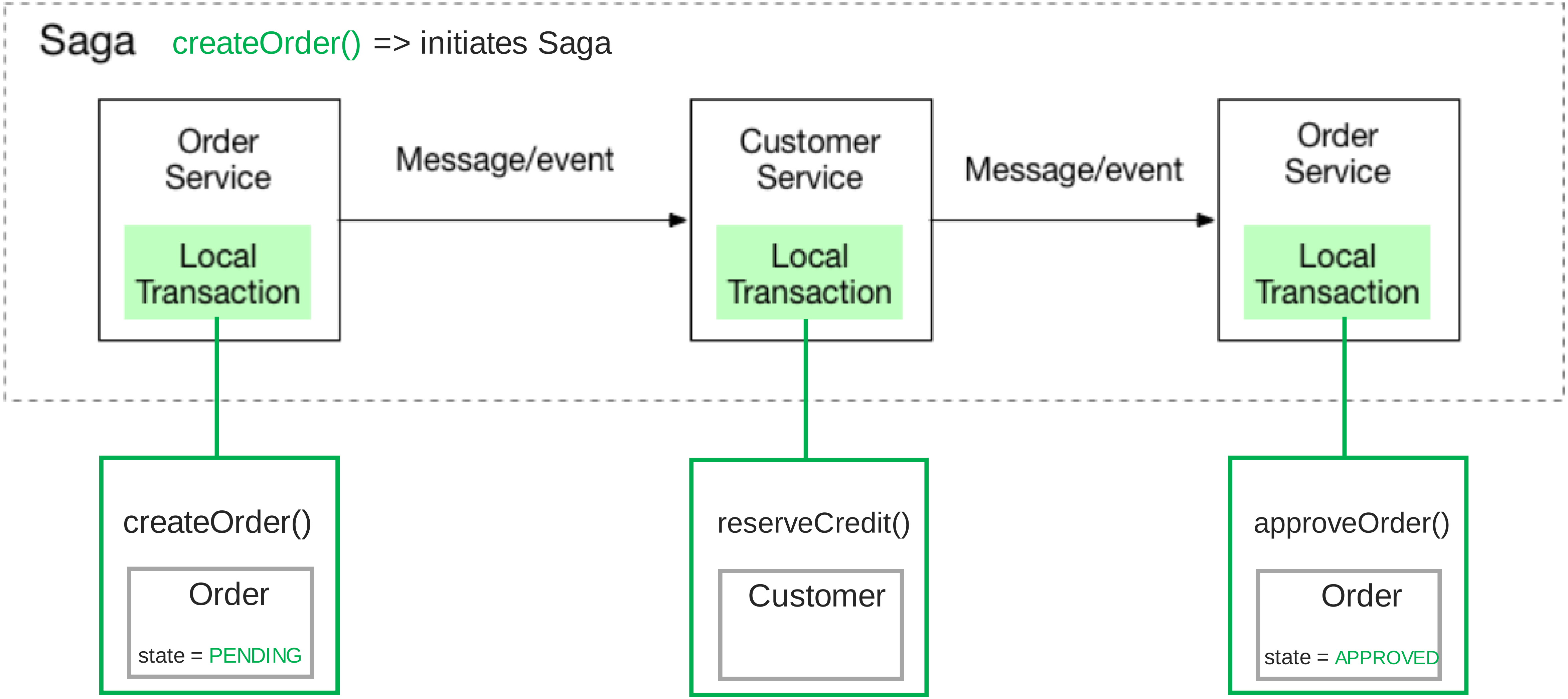
Department of Computer Science
Princeton University
Princeton, N J 08544

From a 1987 paper

- A **saga** is a **sequence of local transactions** (one saga step = a transaction local to a service)
- Each local transaction **updates the database** and **publishes a message or an event** to trigger the next local transaction in the saga.

Solution : Managing transactions with SAGAs

"Create Order" SAGA example



Solution : Managing transactions with SAGAs

Few questions to ask about Sagas

- **Question 1 :** How do the saga participants communicate?
- Answer 1 : Collaboration using **asynchronous, broker-based messaging**

- **Question 2 :**

After the completion of a transaction **T(i)** “something” must decide : what step to execute next ?

  - **Success:** which T(i+1) - branching
  - **Failure:** C(i - 1)

- Answer 2 : There are two ways of coordination sagas :

Choreography
(Distributed decision making)

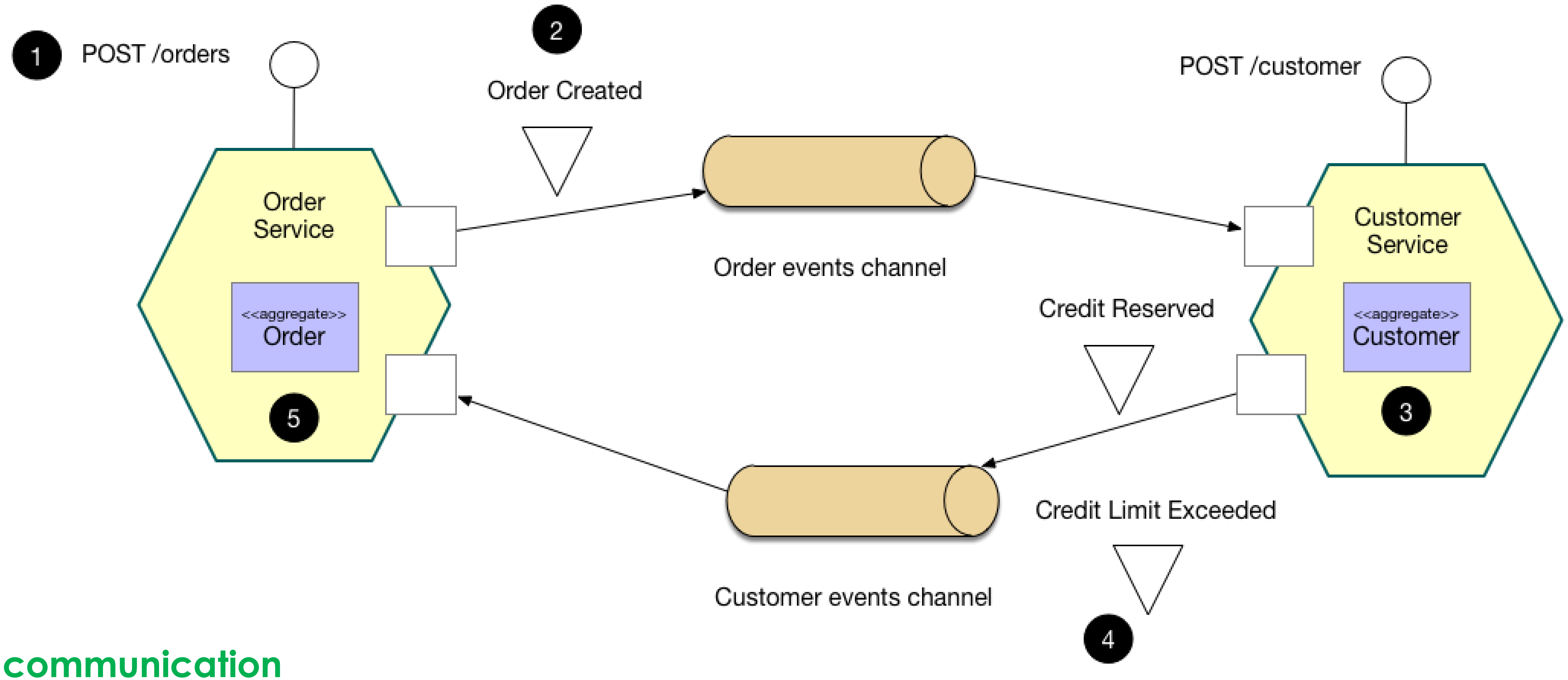
Each local transaction publishes domain events that trigger local transactions in other services

Orchestration
(Centralized decision making)

An orchestrator (object) tells the participants what local transactions to execute

Solution : Managing transactions with SAGAs

Choreography-based saga : Example



Pub/Sub communication

1. The Order Service receives the POST /orders request and creates an Order in a PENDING state
2. It then emits an Order Created event
3. The Customer Service's event handler attempts to reserve credit
4. It then emits an event indicating the outcome
5. The OrderService's event handler either approves or rejects the Order

[Implementation]

Solution : Managing transactions with SAGAs

Choreography-based saga : Benefits and Drawbacks

+

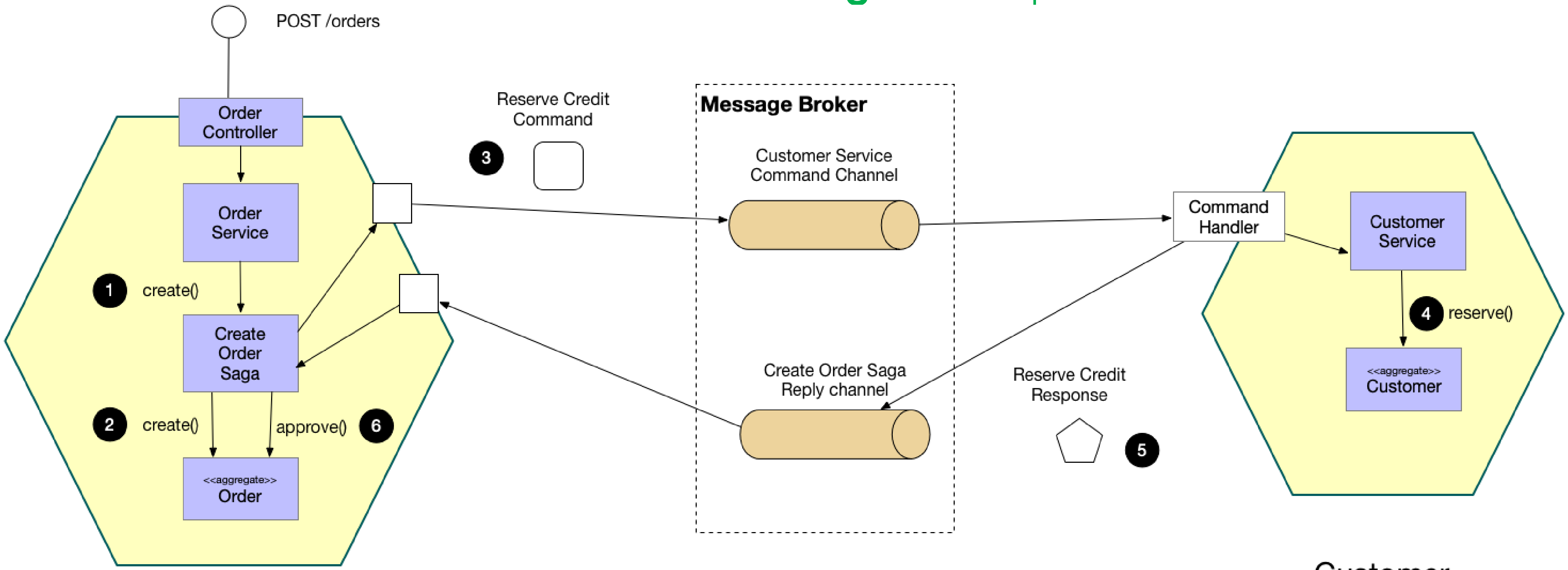
- Simple, especially when using event sourcing
- Participants are loosely coupled

—

- Decentralized implementation - potentially difficult to understand
- Cyclic dependencies - services listen to each other's events,
  e.g. Customer Service must know about all Order events that affect credit
- Overloads domain objects,
  e.g. Order and Customer know too much
- Events = indirect way to make something happen

Solution : Managing transactions with SAGAs

Orchestration-based saga : Example



Asynchronous Command/Reply communication

1. The Order Service receives the POST / orders request and creates the Create Order saga orchestrator
2. The saga orchestrator creates an Order in the PENDING state
3. It then sends a Reserve Credit command to the Customer Service
4. The Customer Service attempts to reserve credit
5. It then sends back a reply message indicating the outcome
6. The saga orchestrator either approves or rejects the Order

[Implementation]

Solution : Managing transactions with SAGAs

Orchestrator-based saga : Benefits and Drawbacks

A saga “orchestrator” is a **persistent object** that implements a state machine and **invokes** the participants

+

- Centralized coordination logic is easier to understand
- Reduced coupling,
  e.g. Customer Service knows less. Simply has API for managing available credit.
- Reduces cyclic dependencies

−

- Risk of smart sagas directing dumb services

Event-Driven Data Management

- Problem Statement : **Distributed Data Management** (*Transactions* and *Queries*)
- One solution of the **Transactions** challenge:
    - The **Saga** pattern
    - Coordination sagas
        - Choreography-based saga
        - Orchestration-based saga
- Two Solutions of the **Queries** challenge :
    - Solution 1 : The **API Composition** pattern
    - Solution 2 : The **CQRS** pattern

Solution (a) of challenge 2 : API composition pattern

Challenge : How to implement **queries** in a microservice architecture ?

Solution :
Implement a query by defining an **API Composer**, which invoke the services that own the data and performs an **in-memory join** of the results.

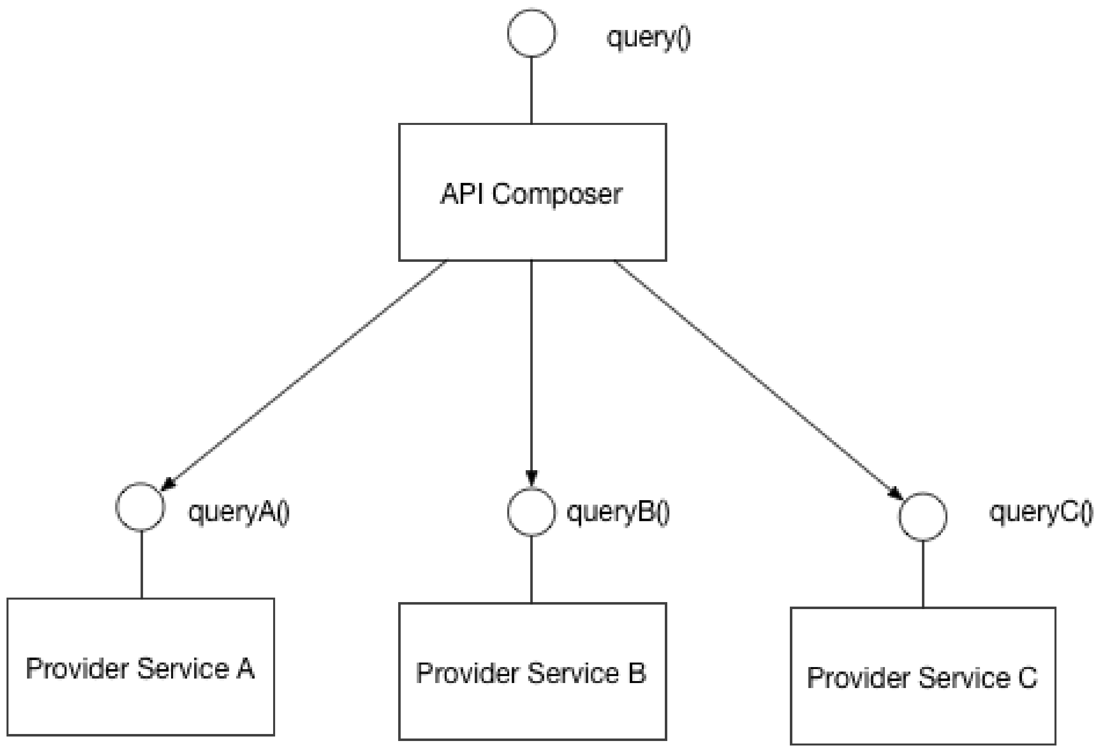
Example :
An API Gateway often does API composition.

Resulting context :
This pattern has the following benefit:

⊕ *It's a simple way to query data in a microservice architecture*

This pattern has the following drawback:

⊖ *Some queries would result inefficient (in-memory joins of large datasets)*



Solution (a) of challenge 2 : API composition pattern

Example of an API composition limitation

Find recent, valuable customers

```
SELECT *
FROM CUSTOMER c, ORDER o
WHERE c.id = o.ID
AND o.ORDER_TOTAL > 100000
AND o.STATE = 'SHIPPED'
AND c. CREATION_DATE > ?
```

Used strategies to implement the query

1 + N strategy:
- Fetch recent customers
- Iterate through customers fetching their shipped orders
- Lots of round trips => **high-latency**

Alternative strategy:
- Fetch recent customers
- Fetch recent orders
- Join
- 2 roundtrips but potentially large datasets => **inefficient**

Not efficiently implemented using API Composition !

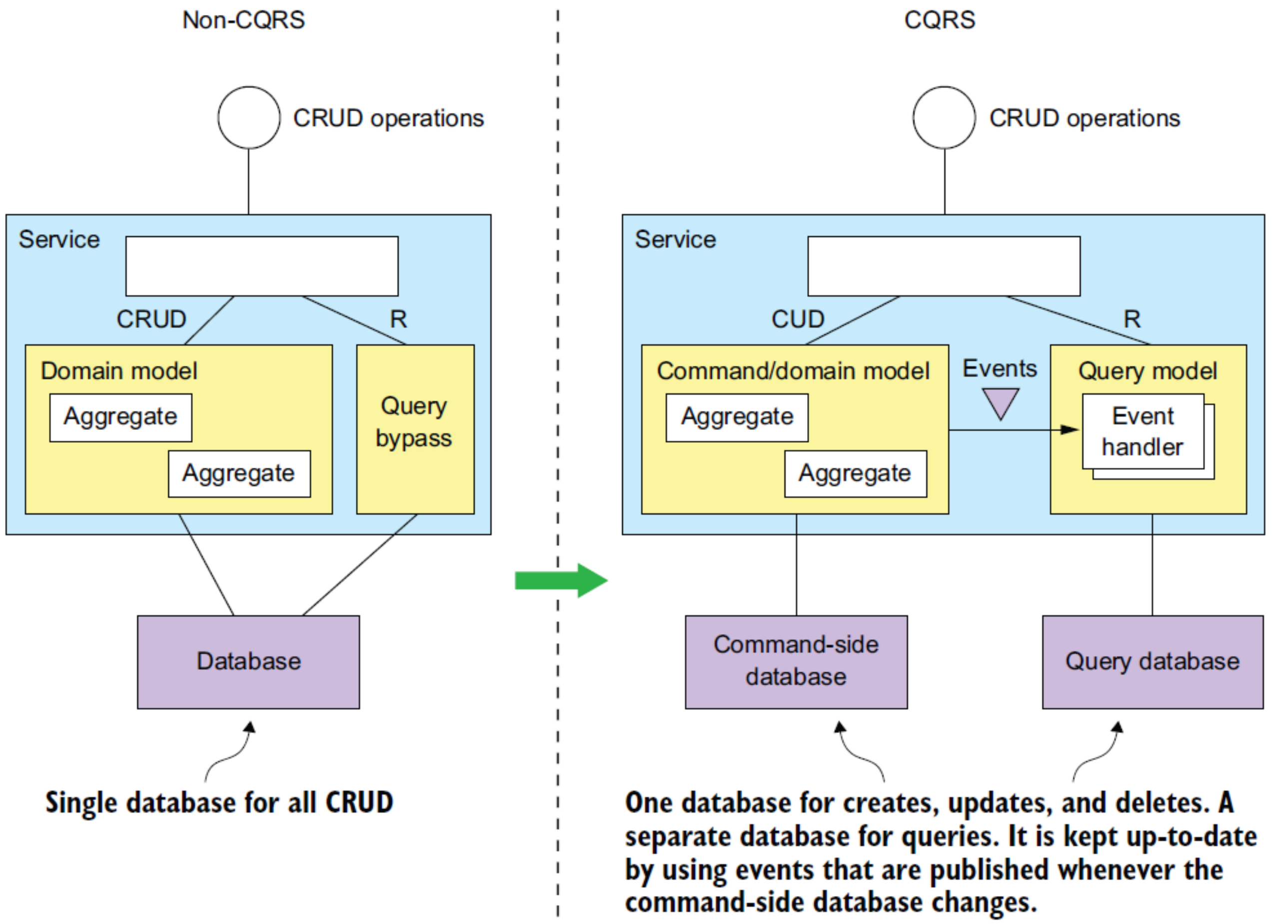
Solution (b) of challenge 2 : Implementing queries with CQRS

CQRS (Command Query Responsibility Segregation**)**

CQRS = Using events to update a queryable replica

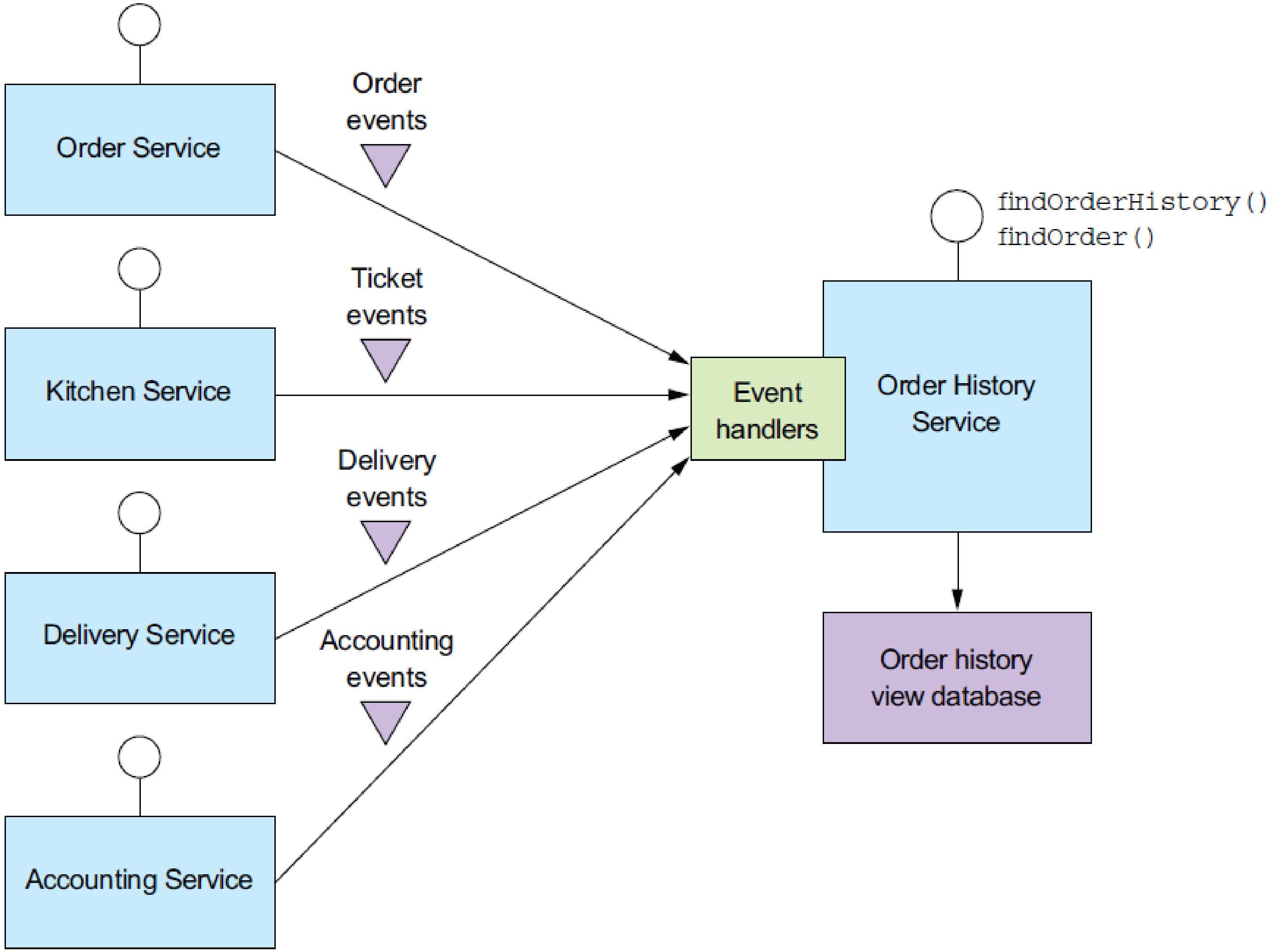
The concept :
We define a **view database**, which is a **read-only replica** that is designed to support complex queries.

The application keeps the replica up to date by subscribing to events published by the services that own the data.



Solution (b) of challenge 2 : Implementing queries with CQRS

CQRS and Query-only services



Solution (b) of challenge 2 : Implementing queries with CQRS

View database : Persisting a customer and order history in MongoDB

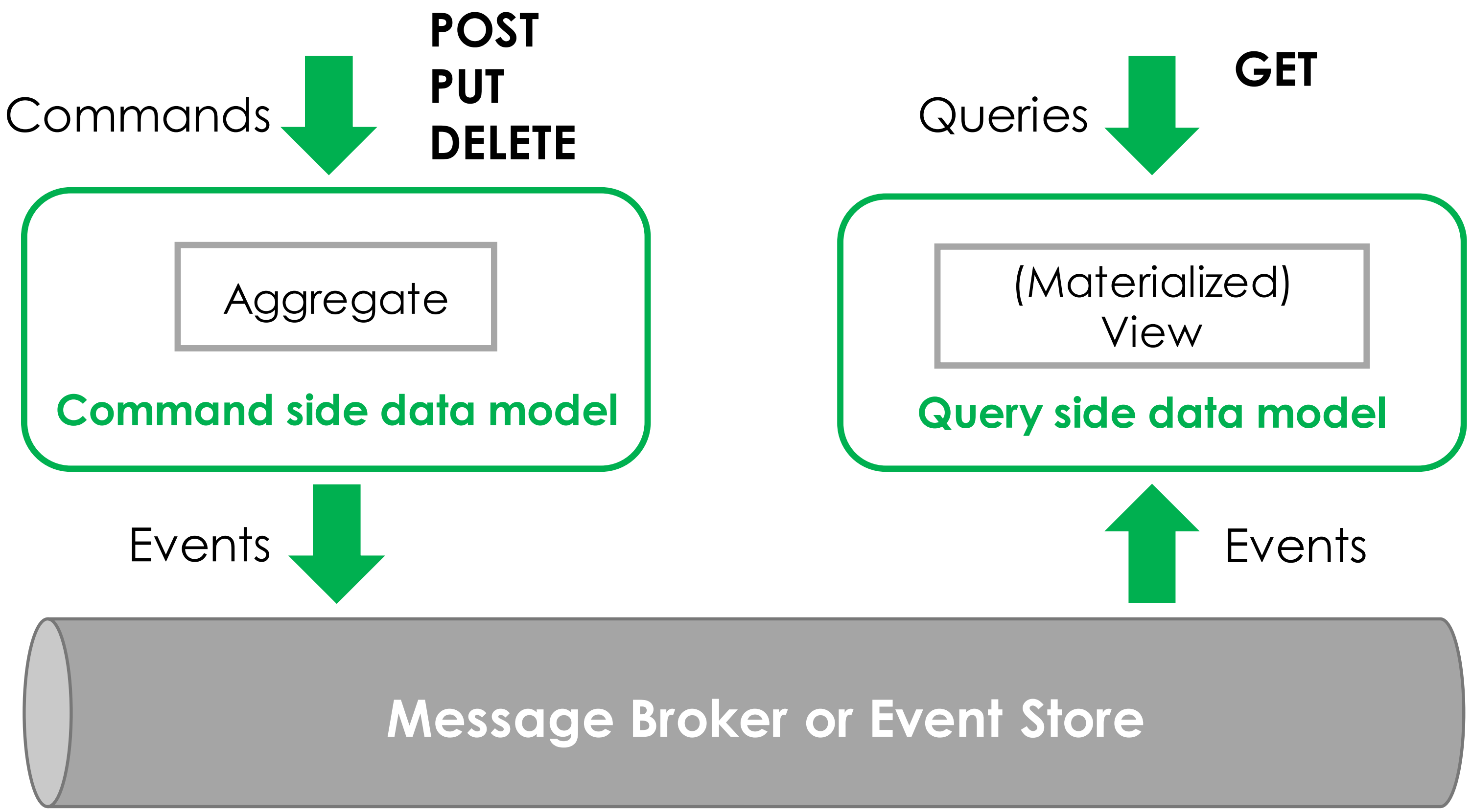
```
{
        "_id" : "0000014f9a45004b 0a00270000000000",
        "name" : "Fred",
        "creditLimit" : {
                "amount" : "2000"
        },
        "orders" : {
                "0000014f9a450063 0a00270000000000" : {
                                "state" : "APPROVED",
                                "orderId" : "0000014f9a450063 0a00270000000000",
                                "orderTotal" : {
                                                "amount" : "1234"
                                }
                },
                "0000014f9a450063 0a00270000000001" : {
                                "state" : "REJECTED",
                                "orderId" : "0000014f9a450063 0a00270000000001",
                                "orderTotal" : {
                                                "amount" : "3000"
                                }
                }
        }
}
```

Customer information

Order information

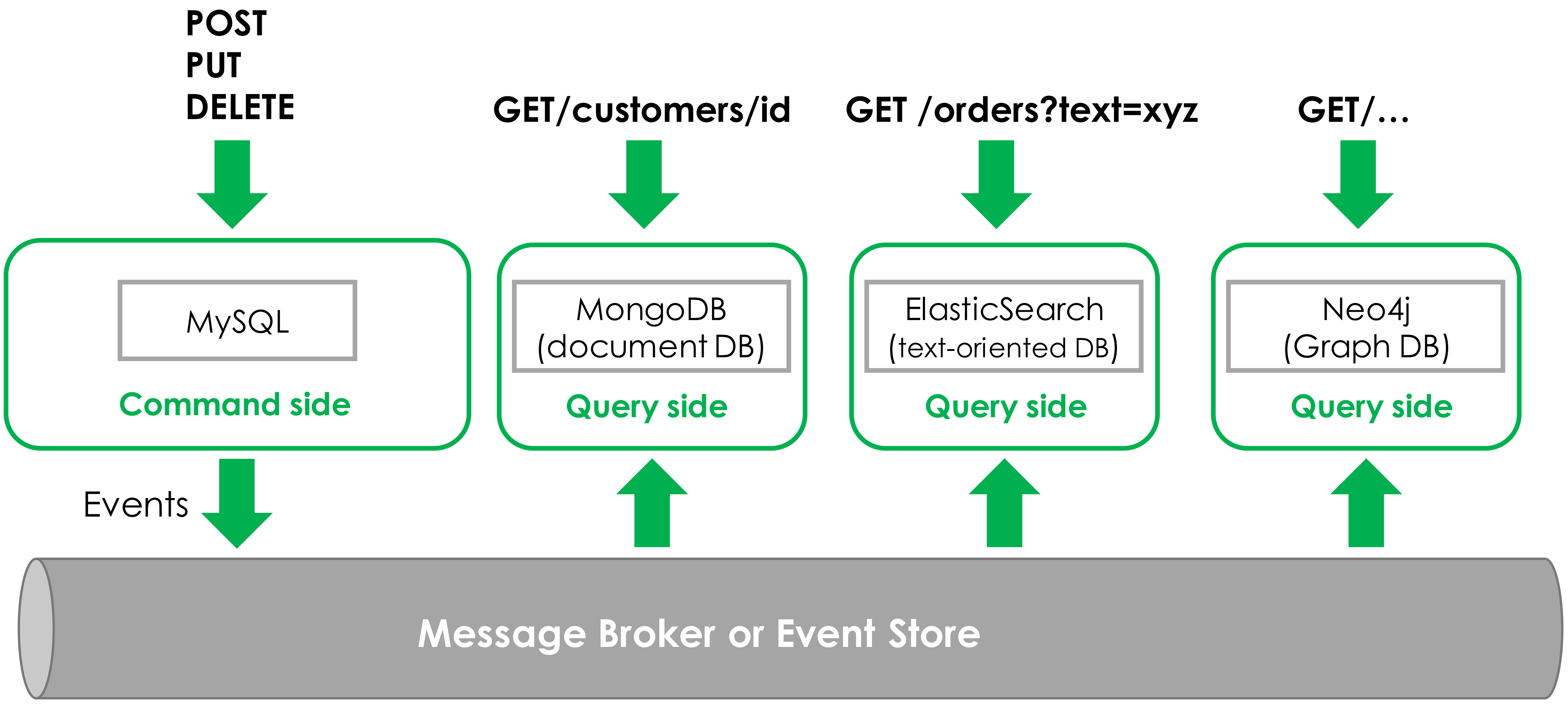
Solution (b) of challenge 2 : Implementing queries with CQRS

Command Query Responsibility Segregation (CQRS)



Solution (b) of challenge 2 : Implementing queries with CQRS

Queries with different database's types

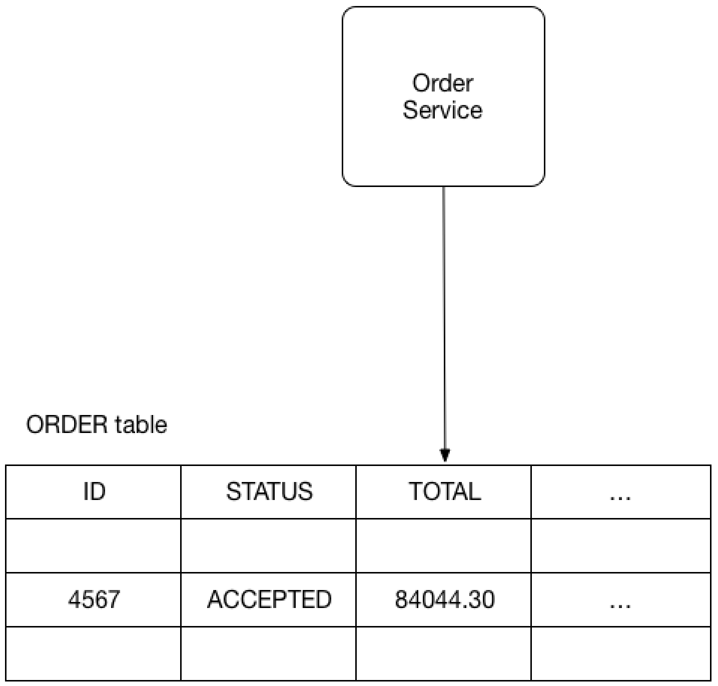


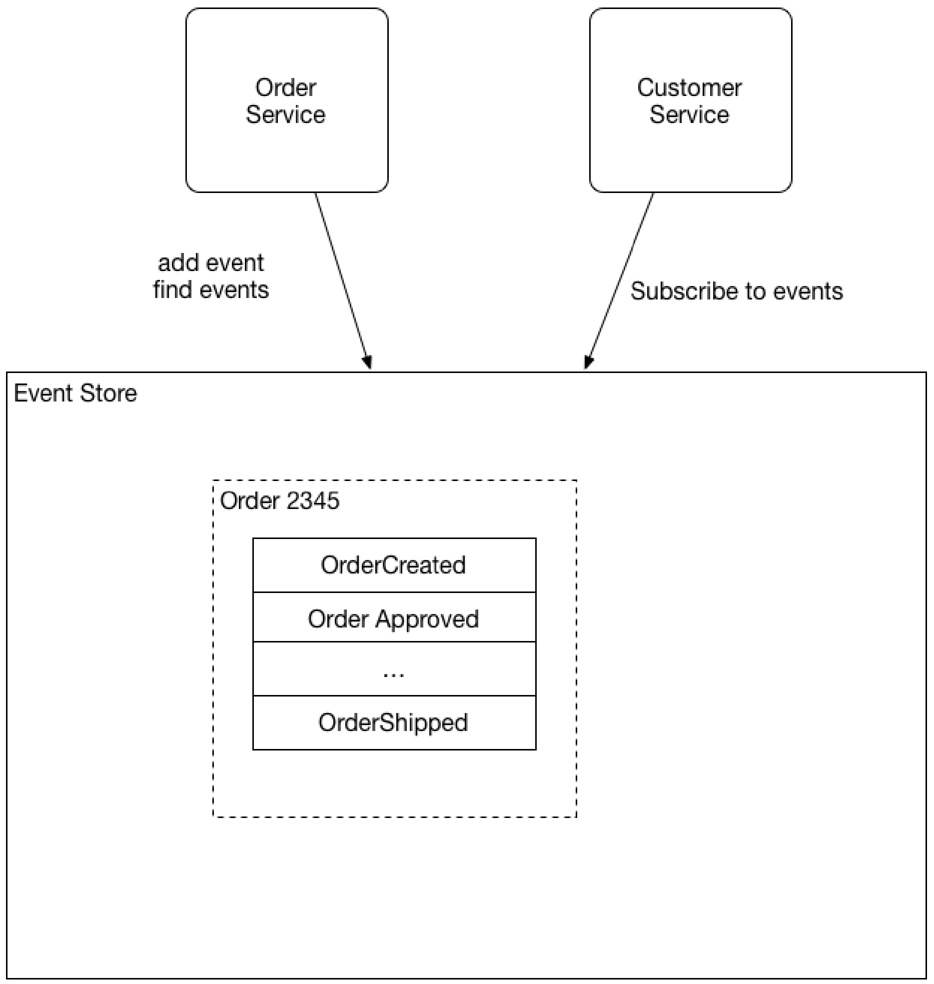
Event-Driven Data Management

- Problem Statement : **Distributed Data Management** (*Transactions* and *Queries*)
- One solution of the **Transactions** challenge:
  - The **Saga** pattern
  - Coordination sagas
    - Choreography-based saga
    - Orchestration-based saga
- Two Solutions of the **Queries** challenge :
  - Solution 1 : The **API Composition** pattern
  - Solution 2 : The **CQRS** pattern
- BONUS: The **Event Sourcing** pattern

Event sourcing

a.k.a. Persisting an object as a sequence of events





[Choreography and event sourcing-based saga]

Event Sourcing
(things to know)

- 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.

Event-Driven Data Management

</END>