Not Just Events: Developing Asynchronous Microservices

Chris Richardson

Founder of Eventuate.io

Founder of the original CloudFoundry.com

Author of POJOs in Action and Microservices Patterns

@crichardson

chris@chrisrichardson.net

http://learn.microservices.io

GOTO CHICAGO 2019

Presentation goal

Implementing transactions and queries in a microservice architecture using asynchronous messaging

Presentation goal

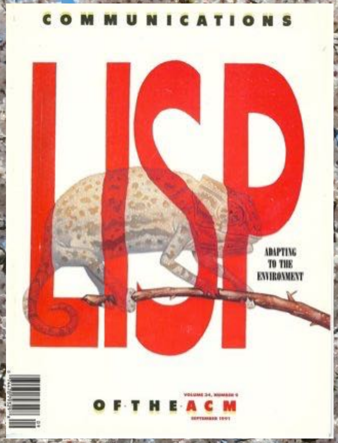
Microservices > REST

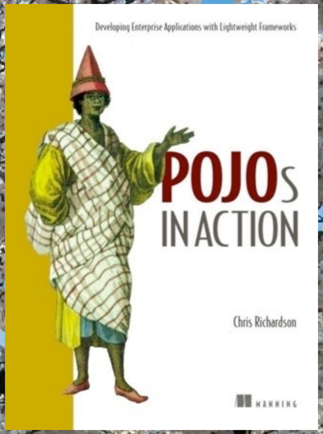
Microservices > Events

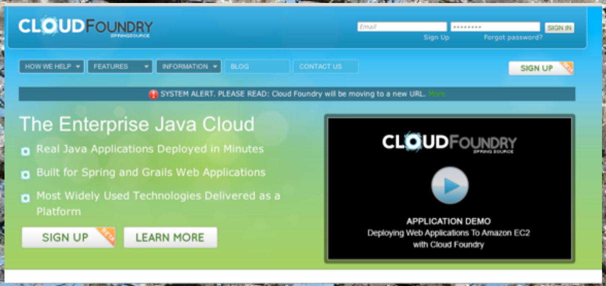
Microservices != Event Sourcing

Apache Kafka != Event Store

About Chris


COMMUNICATIONS
LISP
ADAPTING TO THE ENVIRONMENT
OF THE ACM


Developing Enterprise Applications with Lightweight Frameworks
POJOs IN ACTION
Chris Richardson
MANNING


CLOUDFOUNDRY
The Enterprise Java Cloud
Real Java Applications Deployed in Minutes
Built for Spring and Grails Web Applications
Most Widely Used Technologies Delivered as a Platform
SIGN UP
LEARN MORE
CLOUDFOUNDRY
APPLICATION DEMO
Deploying Web Applications To Amazon EC2 with Cloud Foundry


vmware
spring source

About Chris

Consultant and trainer focussed on helping organizations adopt the microservice architecture

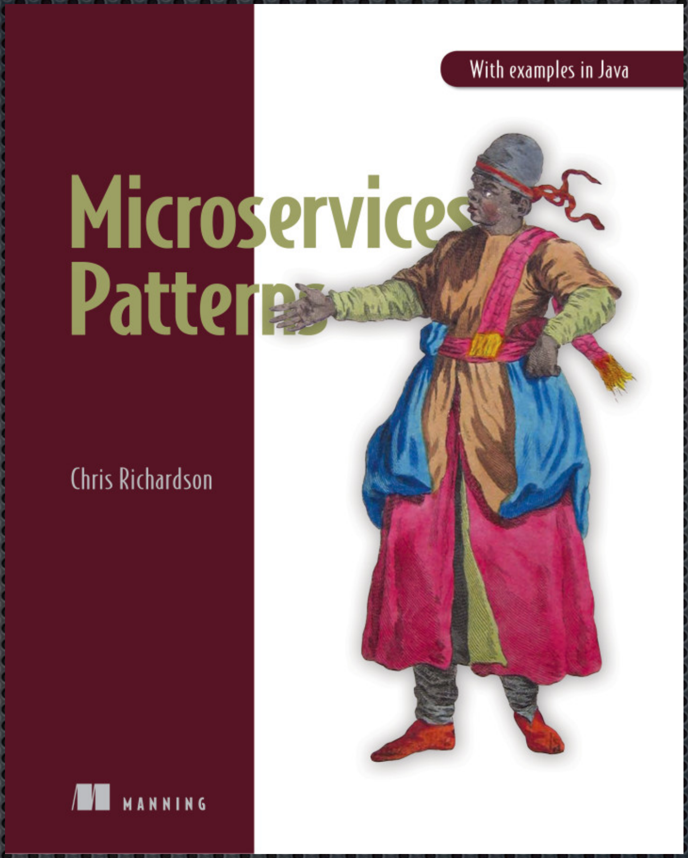
(http://www.chrisrichardson.net/)

About Chris

Founder of a startup that is creating
an open-source/SaaS platform
that simplifies the development of
transactional microservices
(http://eventuate.io)

eventuate

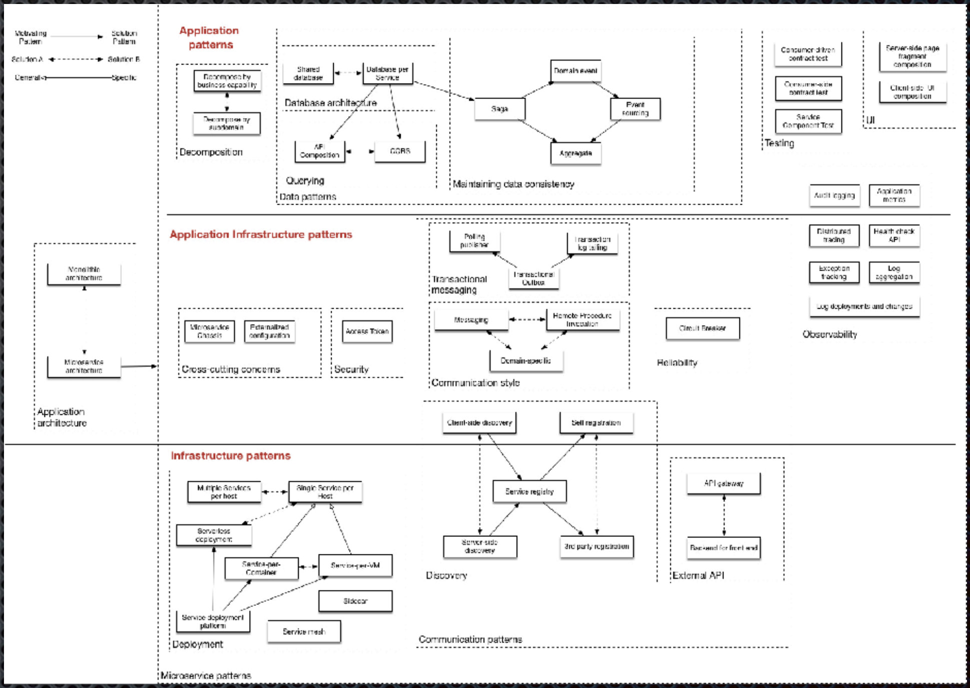
About Chris



Get a 40% discount:
http://bit.ly/gotochgo2019-microservices

About Chris: microservices.io

- Microservices pattern language
- Articles
- Code examples
- Microservices Assessment Platform - http://microservices.io/platform

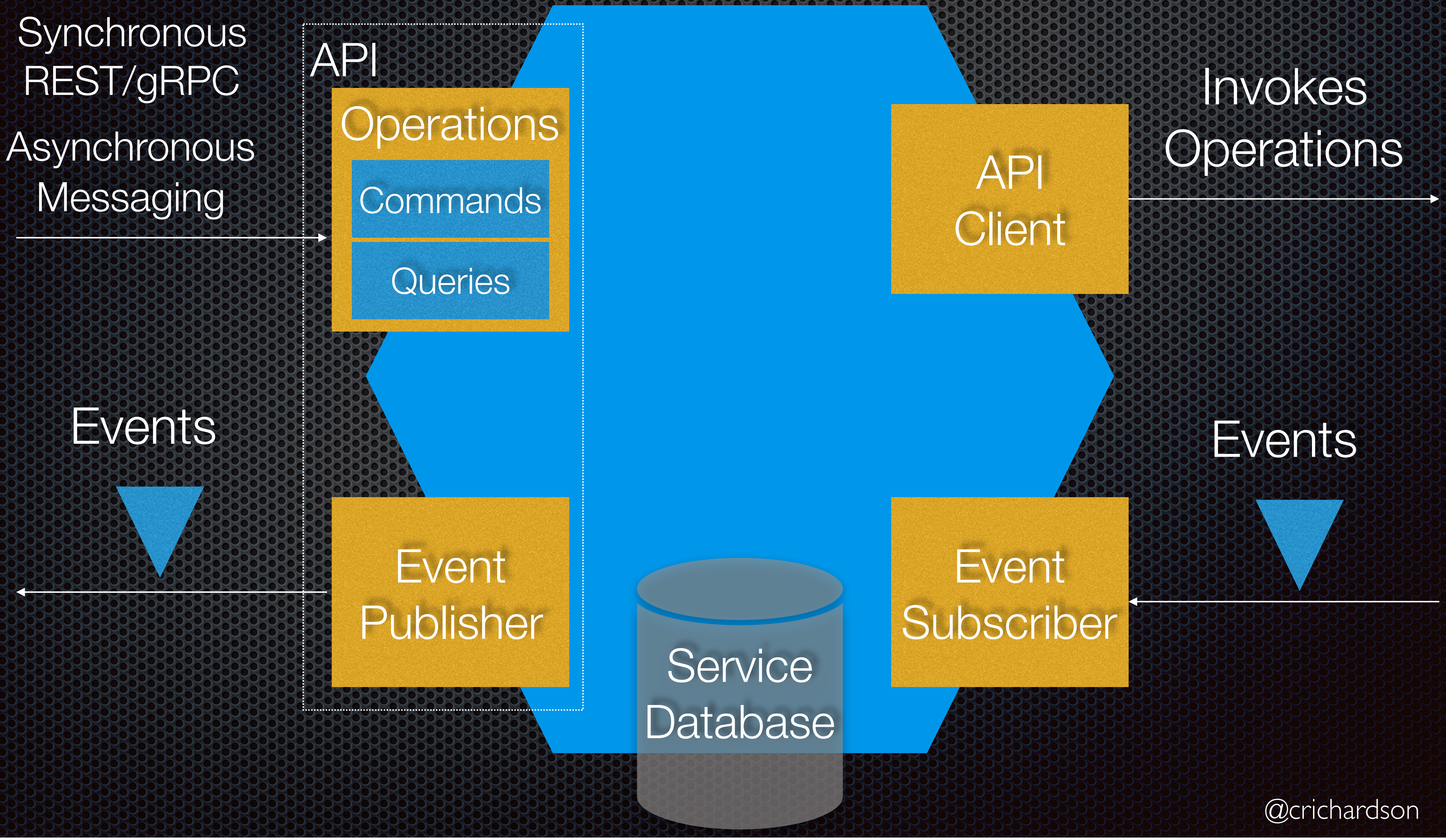


Agenda

- Transactions, queries and microservices
- Managing transactions with sagas
- Implementing queries with CQRS
- Implementing transactional messaging

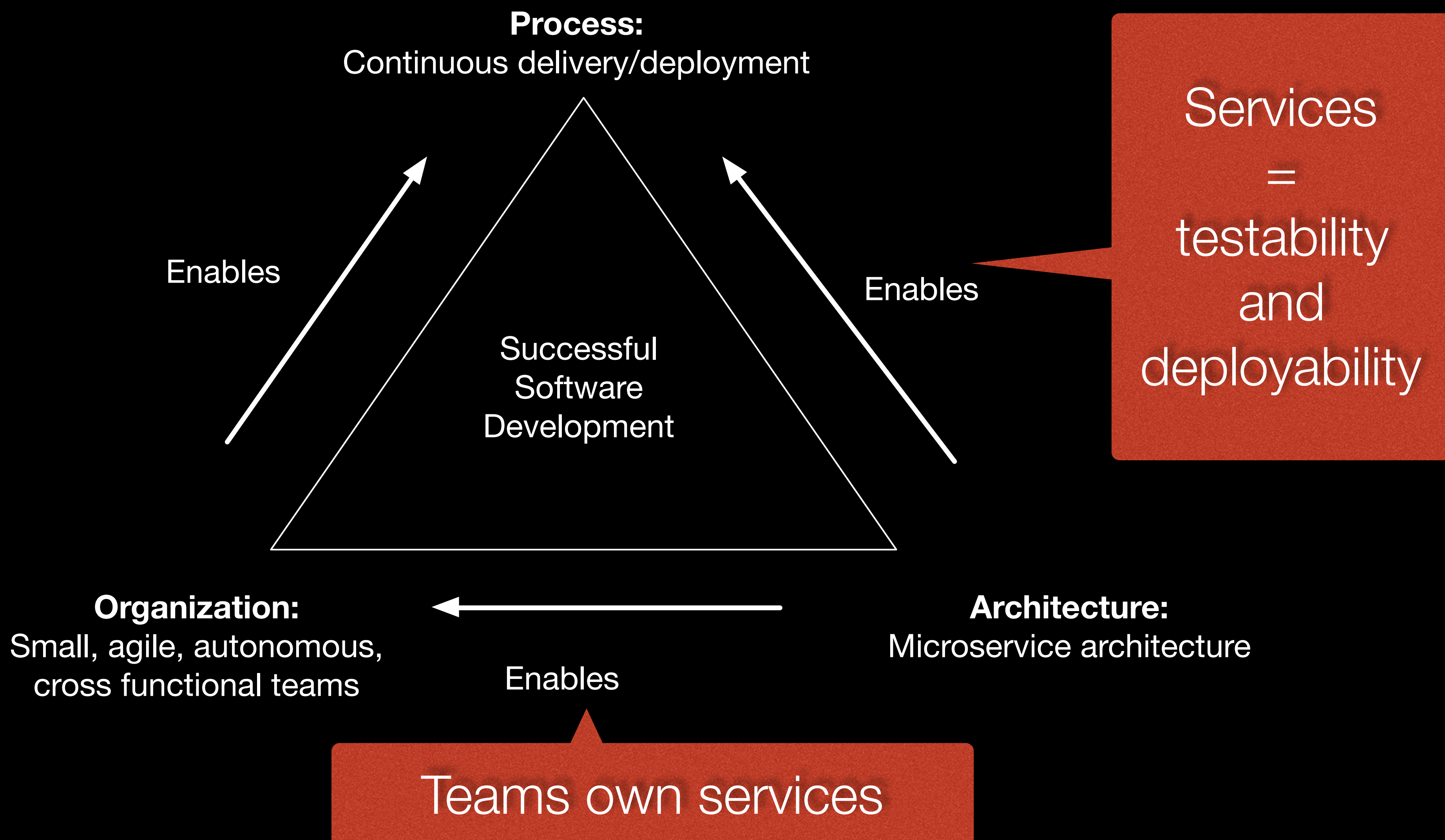
The microservice architecture **structures** an application as a **set of loosely coupled services**

Service = independently deployable component



@crichardson

Microservices enable continuous delivery/deployment



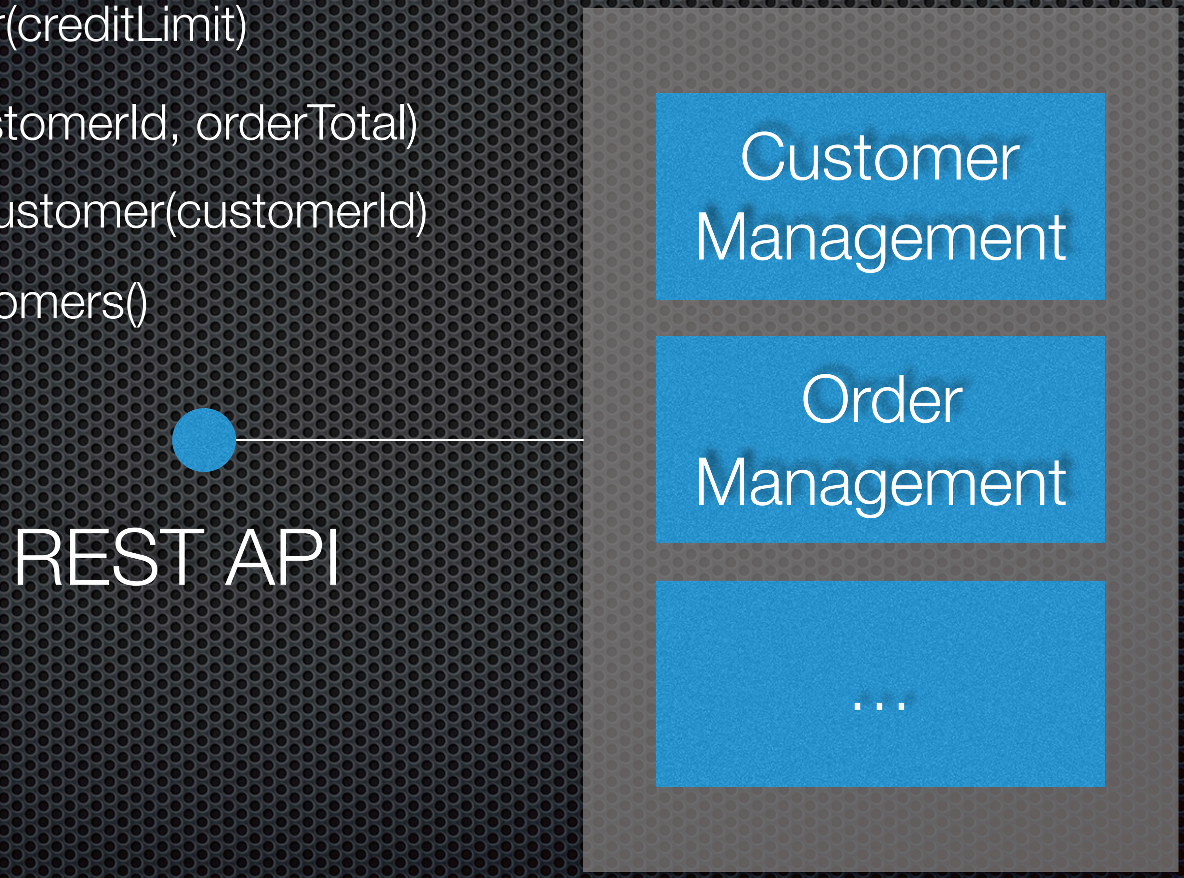
Let’s imagine that you are building an online store API

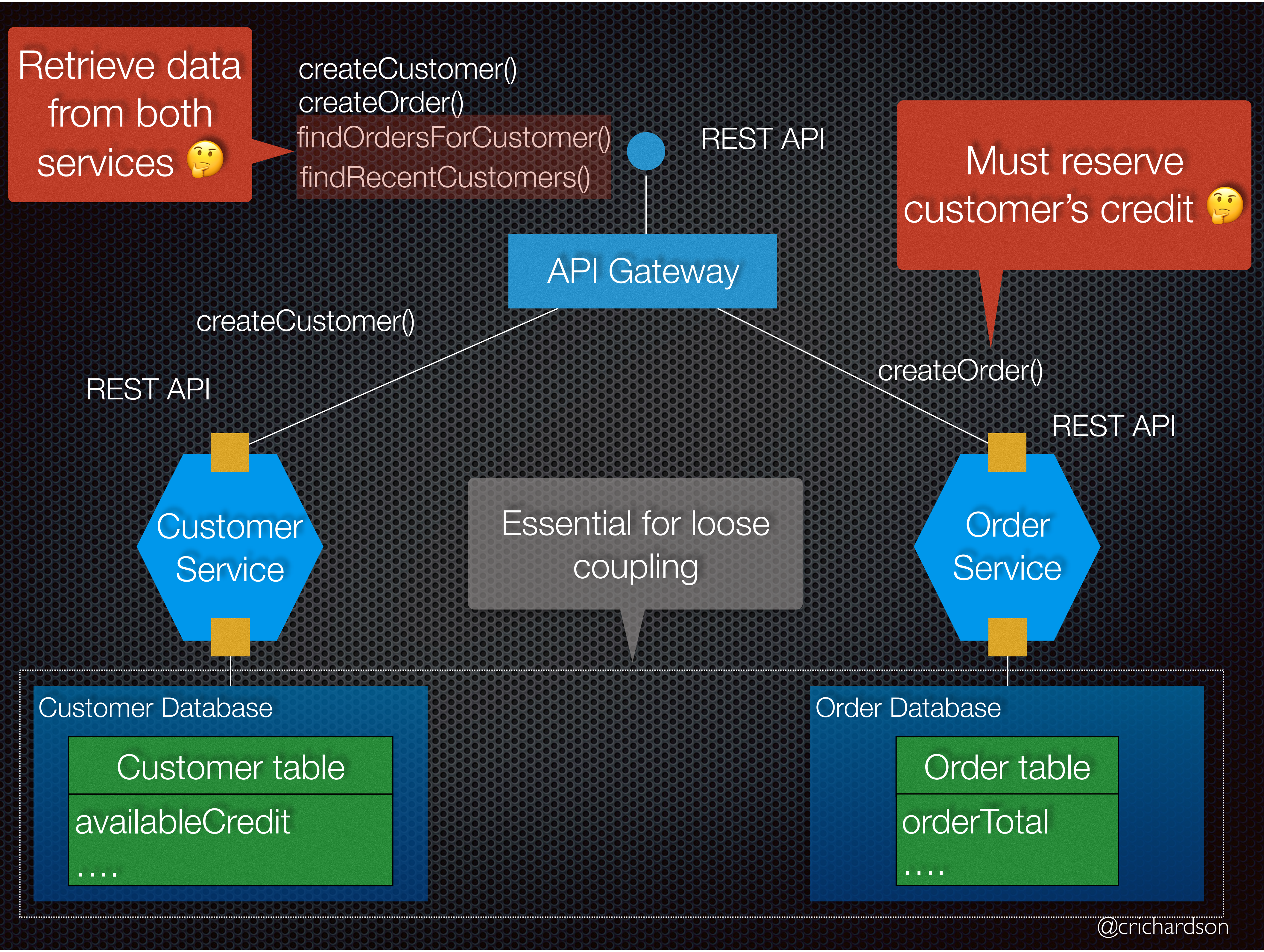
createCustomer(creditLimit)

createOrder(customerId, orderTotal)

findOrdersForCustomer(customerId)

findRecentCustomers()




Retrieve data from both services 🤔
createCustomer()
createOrder()
findOrdersForCustomer()
findRecentCustomers()
REST API
Must reserve customer's credit 🤔
API Gateway
createCustomer()
createOrder()
REST API
REST API
Customer Service
Essential for loose coupling
Order Service
Customer Database
Customer table
availableCredit
....
Order Database
Order table
orderTotal
....
@crichardson

No ACID transactions that span services

Distributed transactions

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

Private to the Order Service

Private to the Customer Service

rdson

Querying across services is not straightforward

Private to Customer Service

Private to Order Service

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

Find customer and their orders

Agenda

- Transactions, queries and microservices
- Managing transactions with sagas
- Implementing queries with CQRS
- Implementing transactional messaging

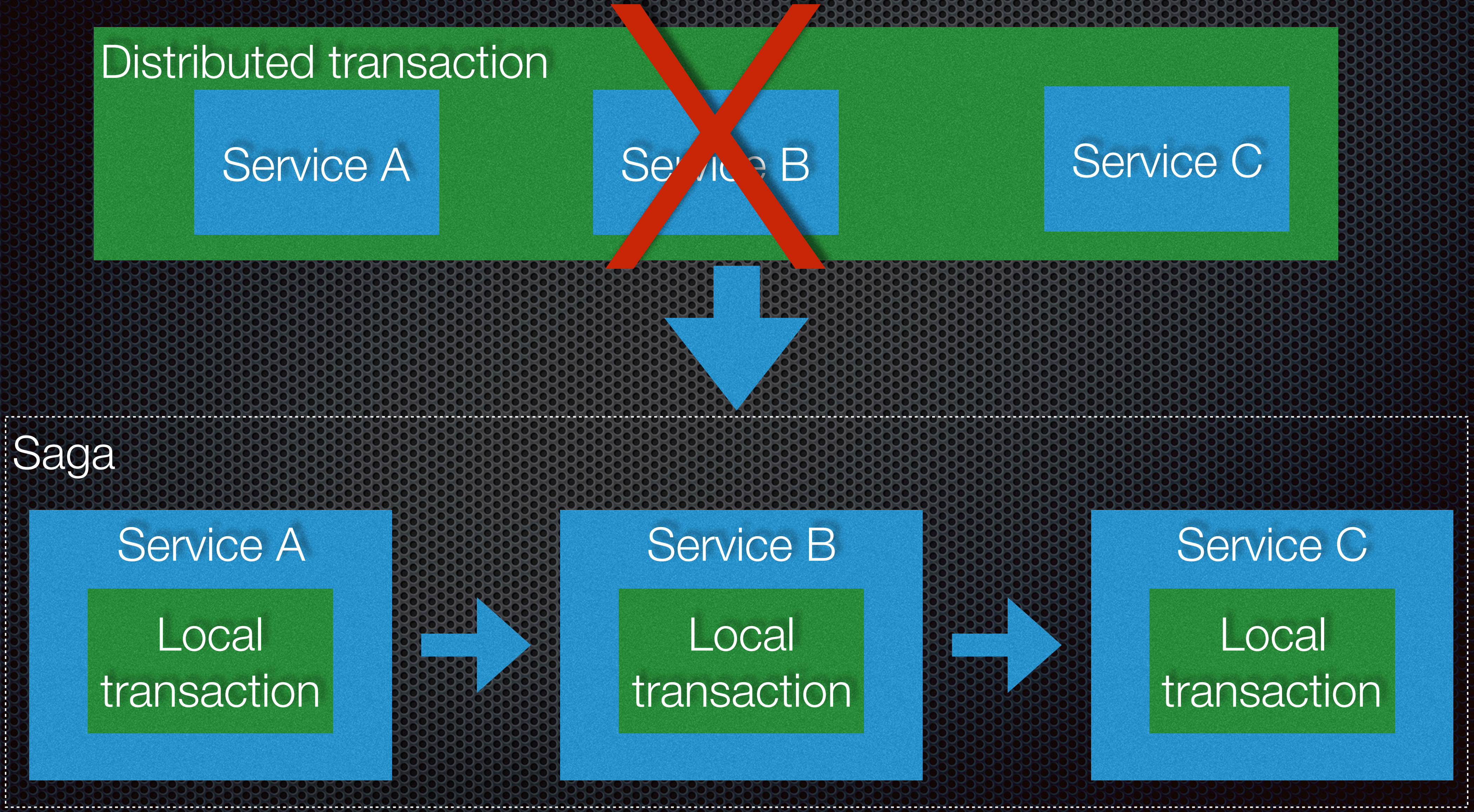
From a 1987 paper

SAGAS

Hector Garcia-Molina
Kenneth Salem

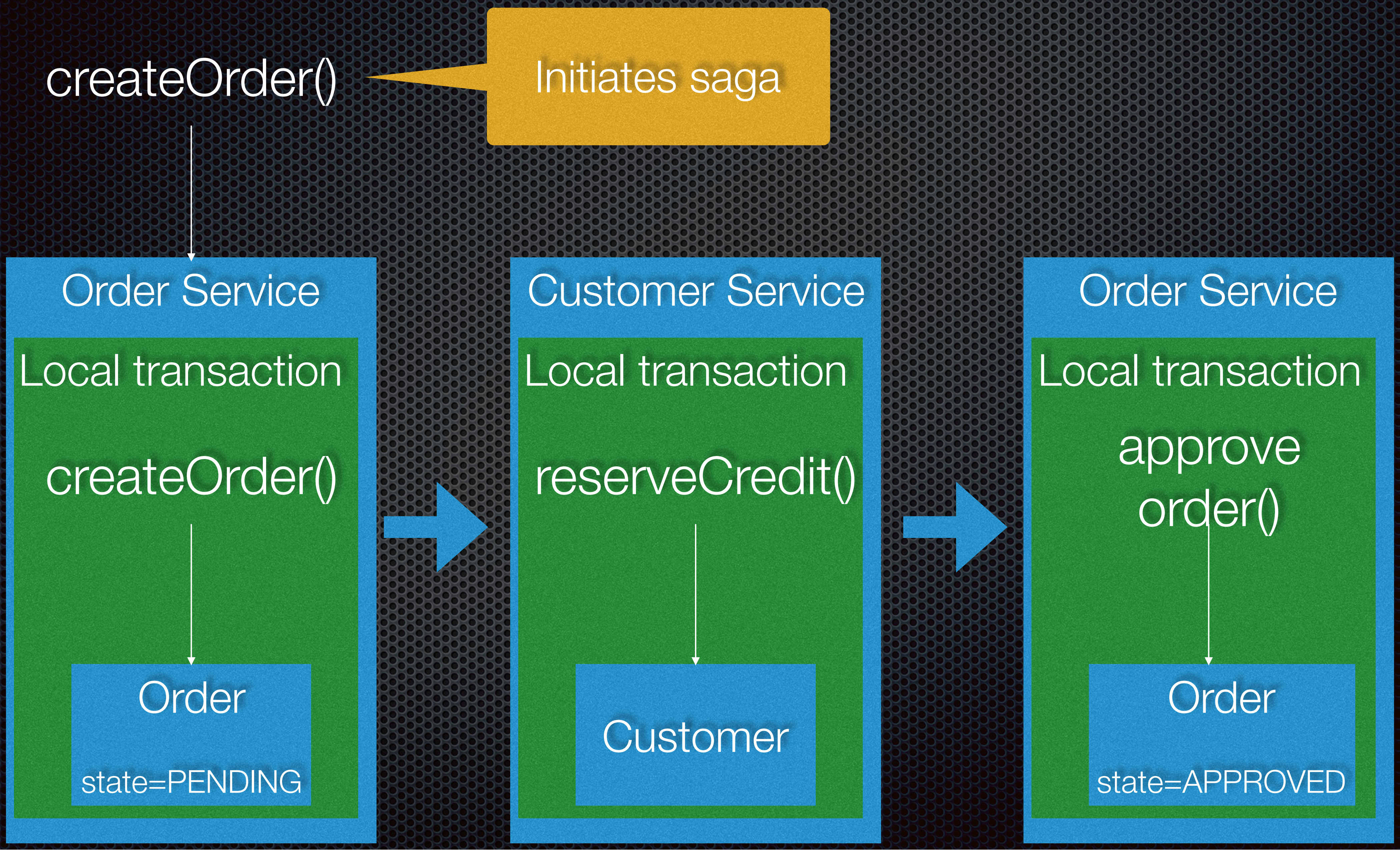
Department of Computer Science
Princeton University
Princeton, N J 08544

Use Sagas instead of 2PC



https://microservices.io/patterns/data/saga.html

Create Order Saga



Saga design challenges

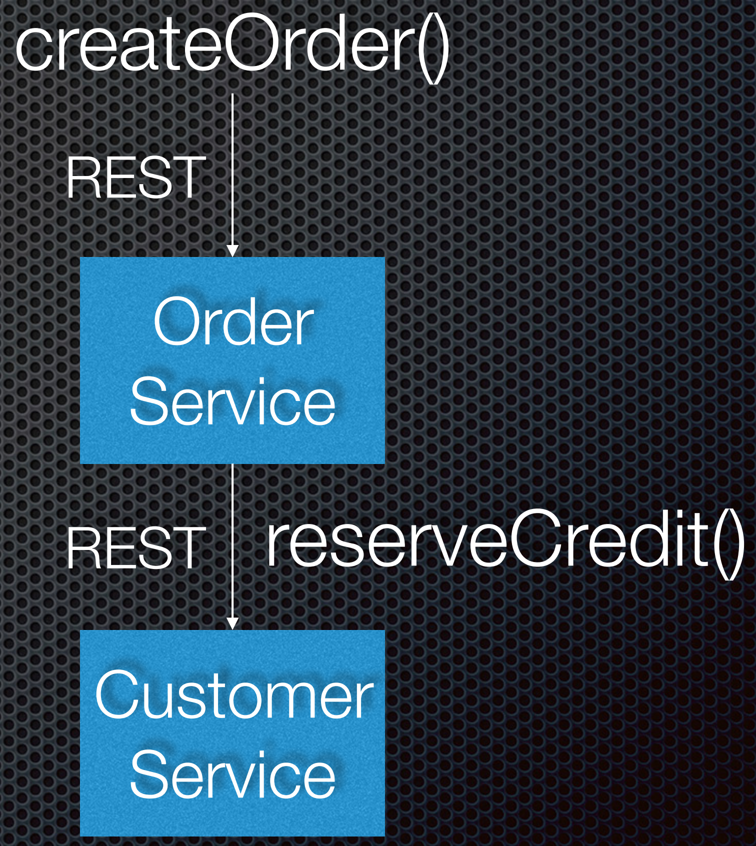
- API design
  - Synchronous REST API initiates asynchronous saga
  - When to send back a response?
- Rollback ⇒ compensating transactions
- Sagas are ACD - No I
  - Sagas are interleaved ⇒ anomalies, such as lost updates
  - Must use countermeasures

https://www.slideshare.net/chris.e.richardson/saturn-2018-managing-data-consistency-in-a-microservice-architecture-using-sagas

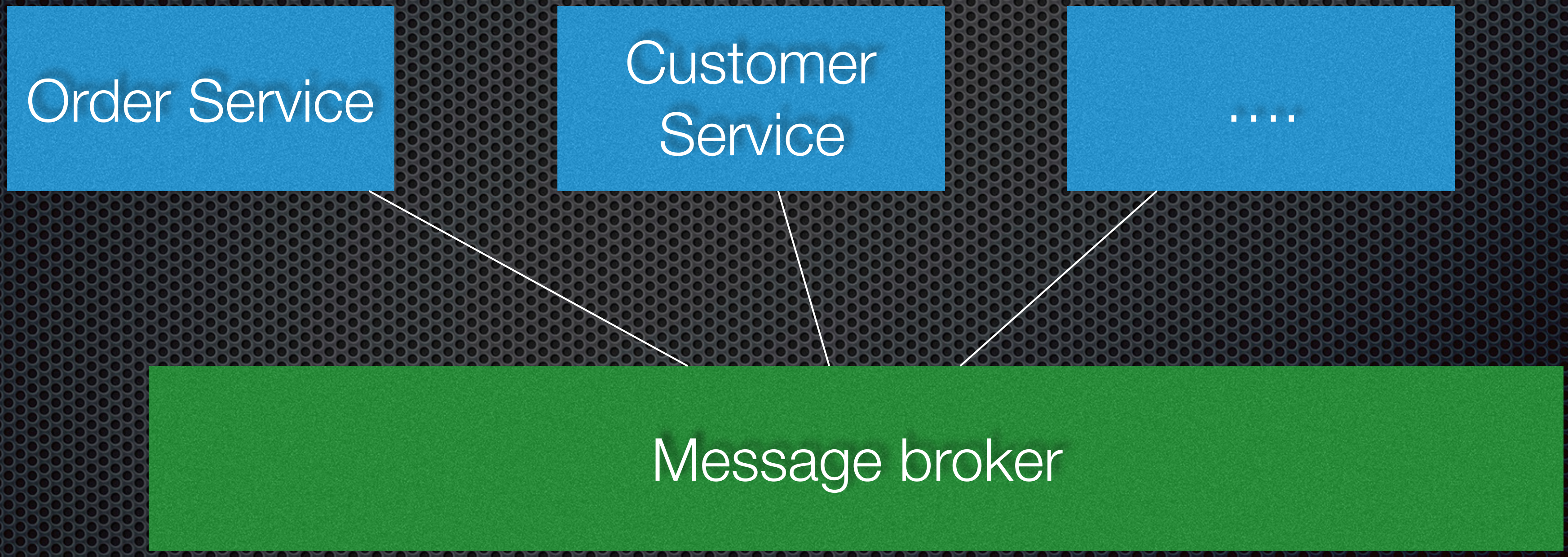
How do the saga participants communicate?

- Synchronous communication, e.g. REST = temporal coupling
- Client and server need to be both available
- Customer Service fails ⇒

  retry provided it's idempotent
- Order Service fails ⇒ Oops



Collaboration using asynchronous, broker-based messaging

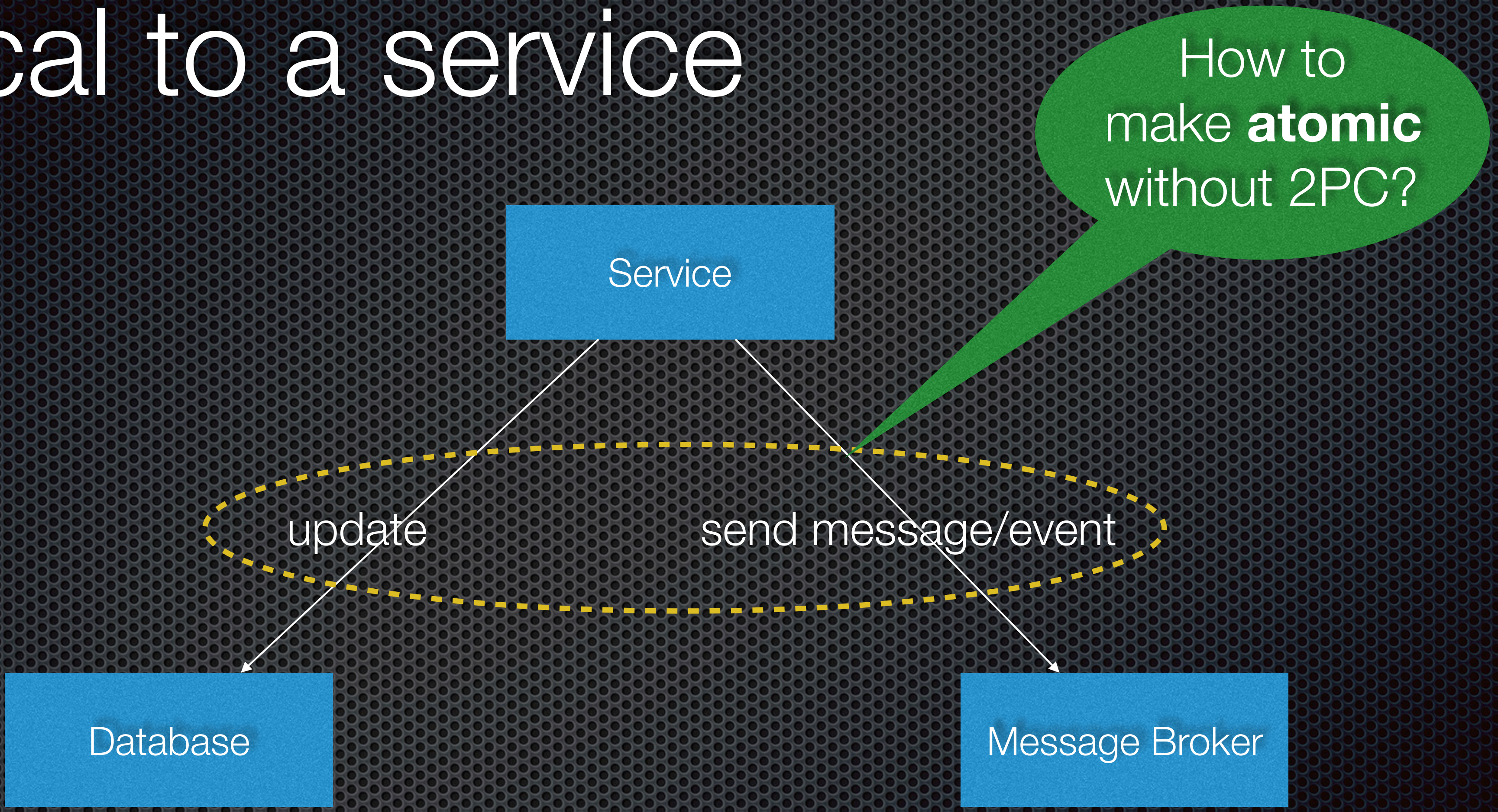


@crichardson

About the message broker

- At least once delivery
  - Ensures a saga completes when its participants are temporarily unavailable
- Ordered delivery
- Mechanism for scaling consumers that preserves ordering e.g.
  - Apache Kafka consumer group
  - ActiveMQ message group
  - …

Saga step = a transaction local to a service



@crichardson

How to sequence the saga transactions?

- After the completion of transaction Ti “something” must decide what step to execute next
- Success: which T(i+1) - branching
- Failure: C(i - 1)

Choreography: **distributed** decision making

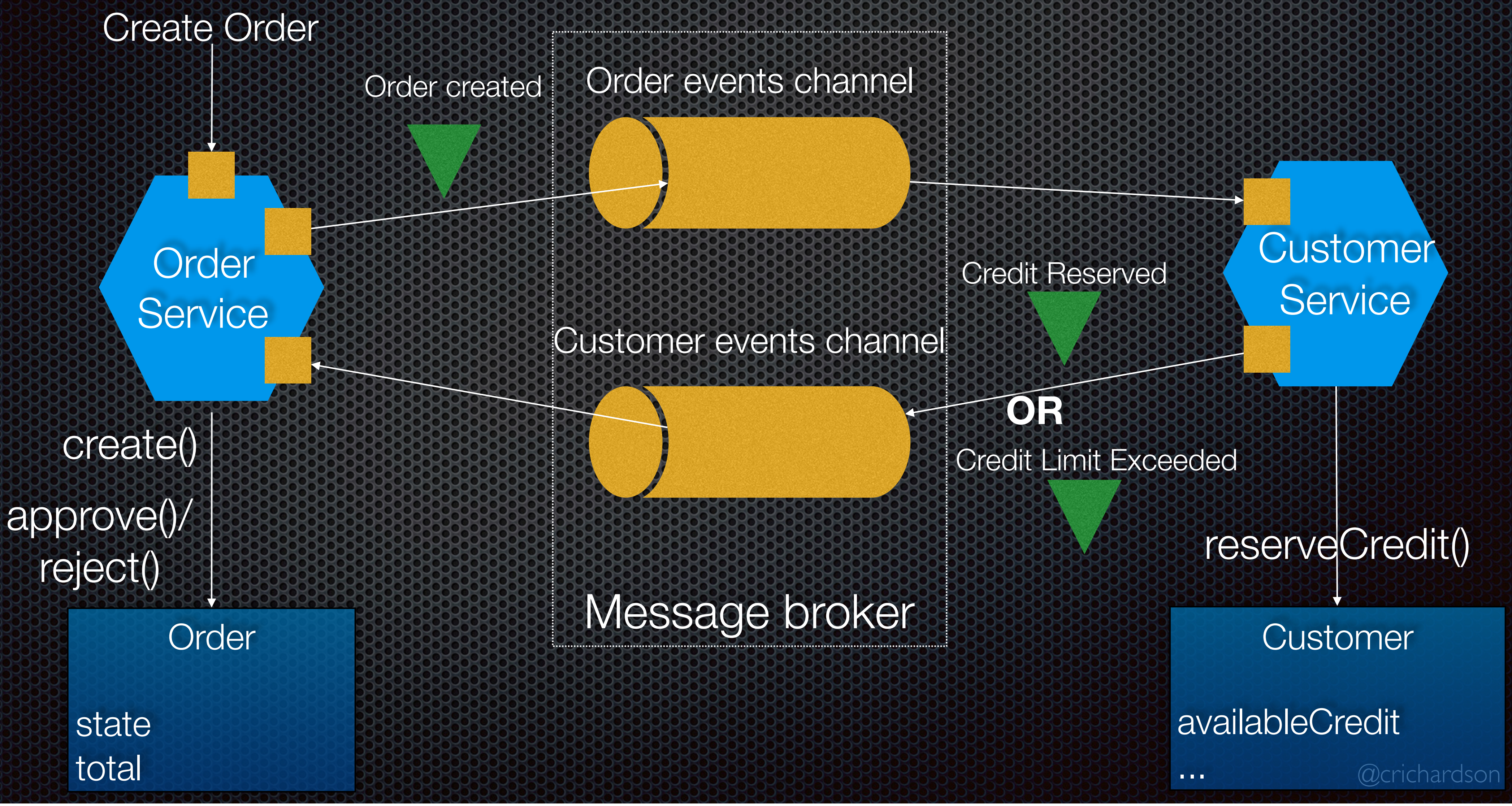
vs.

Orchestration: **centralized** decision making

Agenda

- Transactions, queries and microservices
- Managing transactions with sagas
  - Overview
  - Choreography
  - Orchestration
- Implementing queries with CQRS
- Implementing transactional messaging

Choreography-based Create Order Saga



Benefits and drawbacks of choreography

Benefits

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

Drawbacks

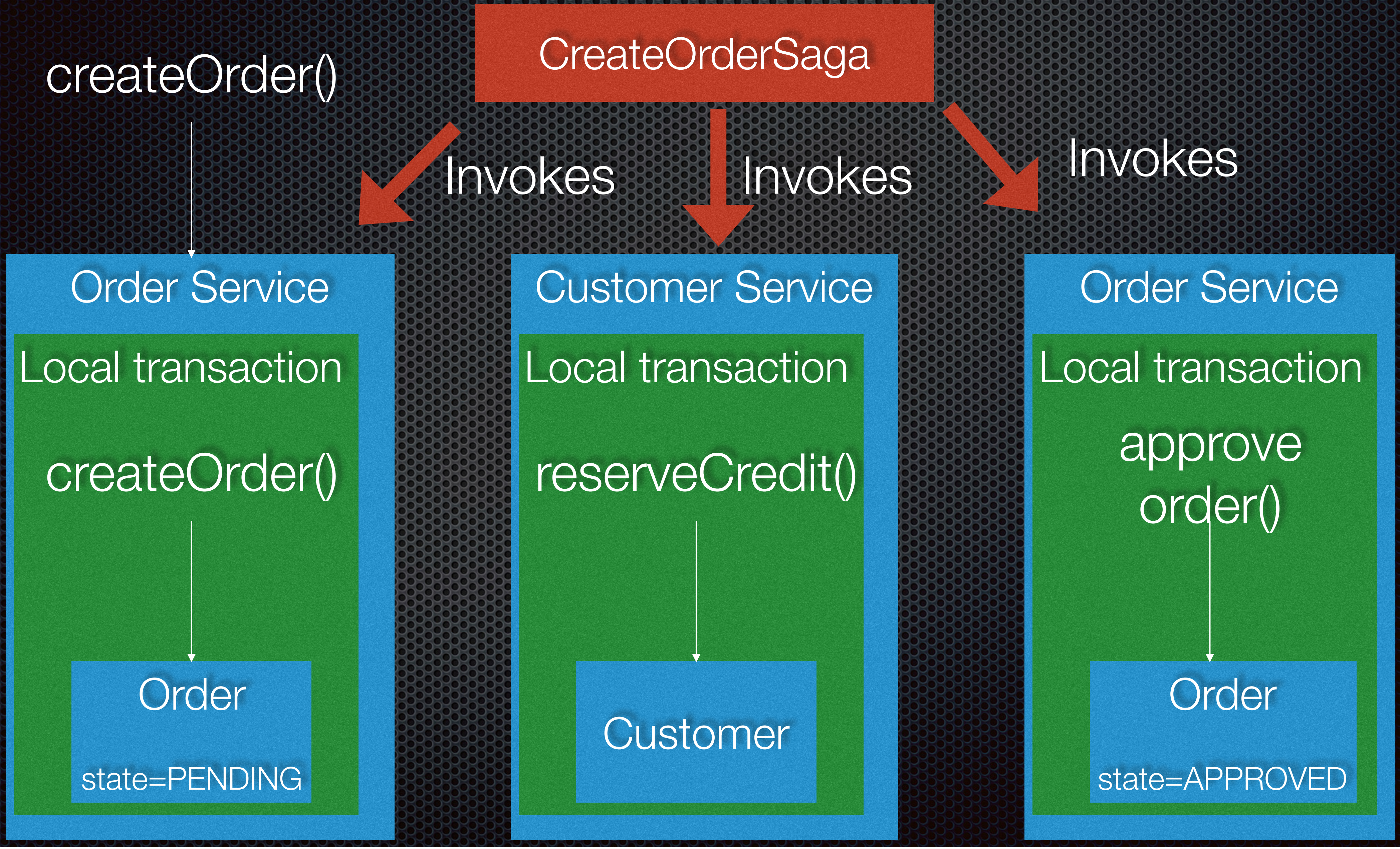
- Decentralized implementation - potentially difficult to understand
- Cyclic dependencies - services listen to each other's events, e.g. Customer Service **must know** about all Order events that affect credit
- Overloads domain objects, e.g. Order and Customer **know** too much
- Events = indirect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

https://github.com/eventuate-examples/eventuate-examples-java-customers-and-orders

Agenda

- Transactions, queries and microservices
- Managing transactions with sagas
    - Overview
    - Choreography
    - Orchestration
- Implementing queries with CQRS
- Implementing transactional messaging

Orchestration-based coordination using command messages

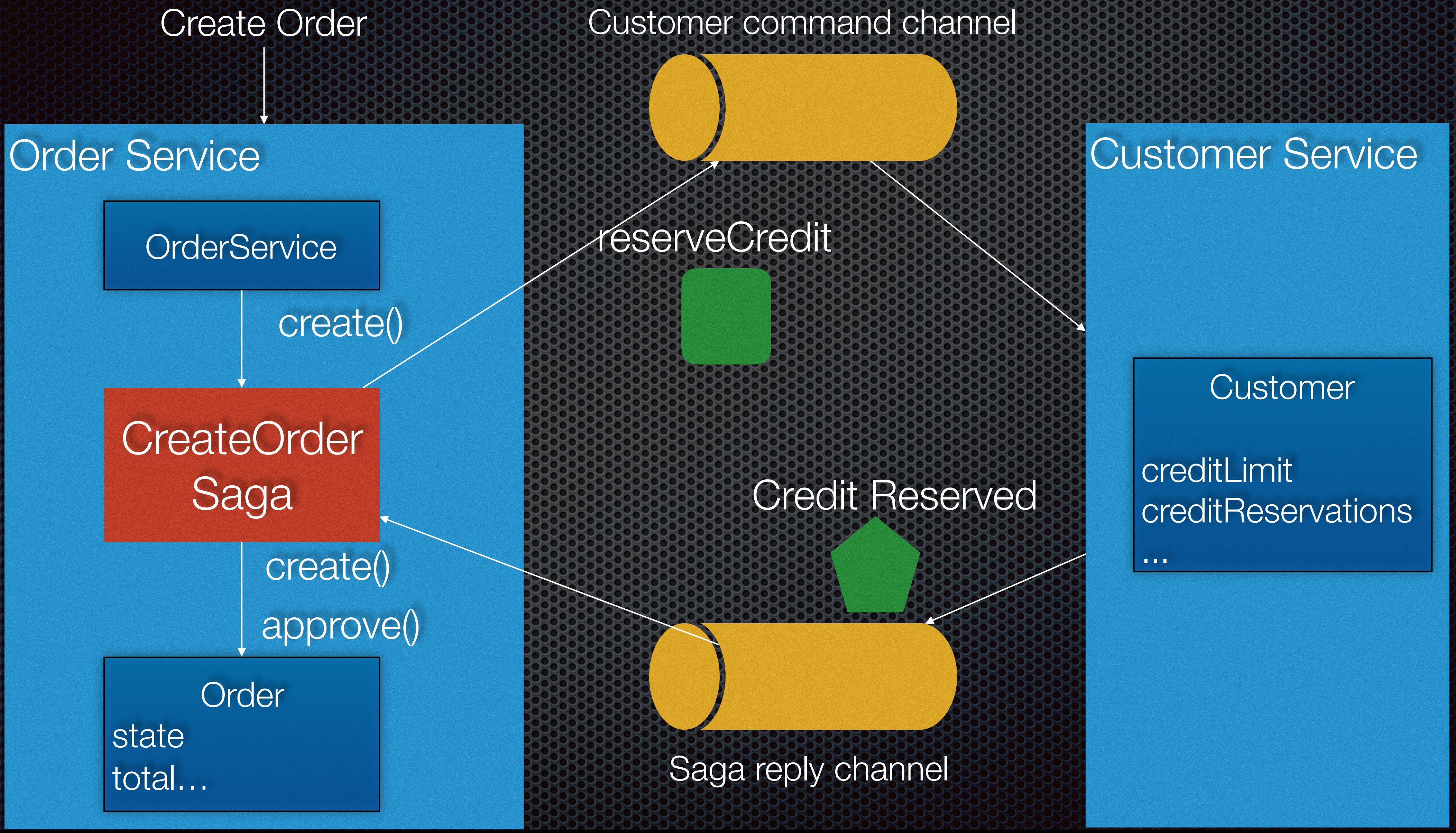


A saga (orchestrator)
is a **persistent object**
that
implements a state machine
and
invokes the participants

Saga orchestrator behavior

- On create:
  - Invokes a saga participant
  - Persists state in database
  - Wait for a reply
- On reply:
  - Load state from database
  - Determine which saga participant to invoke next
  - Invokes saga participant
  - Updates its state
  - Persists updated state
  - Wait for a reply
  - …

CreateOrderSaga orchestrator



- https://github.com/eventuate-tram/eventuate-tram-sagas-examples-customers-and-orders

Benefits and drawbacks of orchestration

Benefits

- 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

Drawbacks

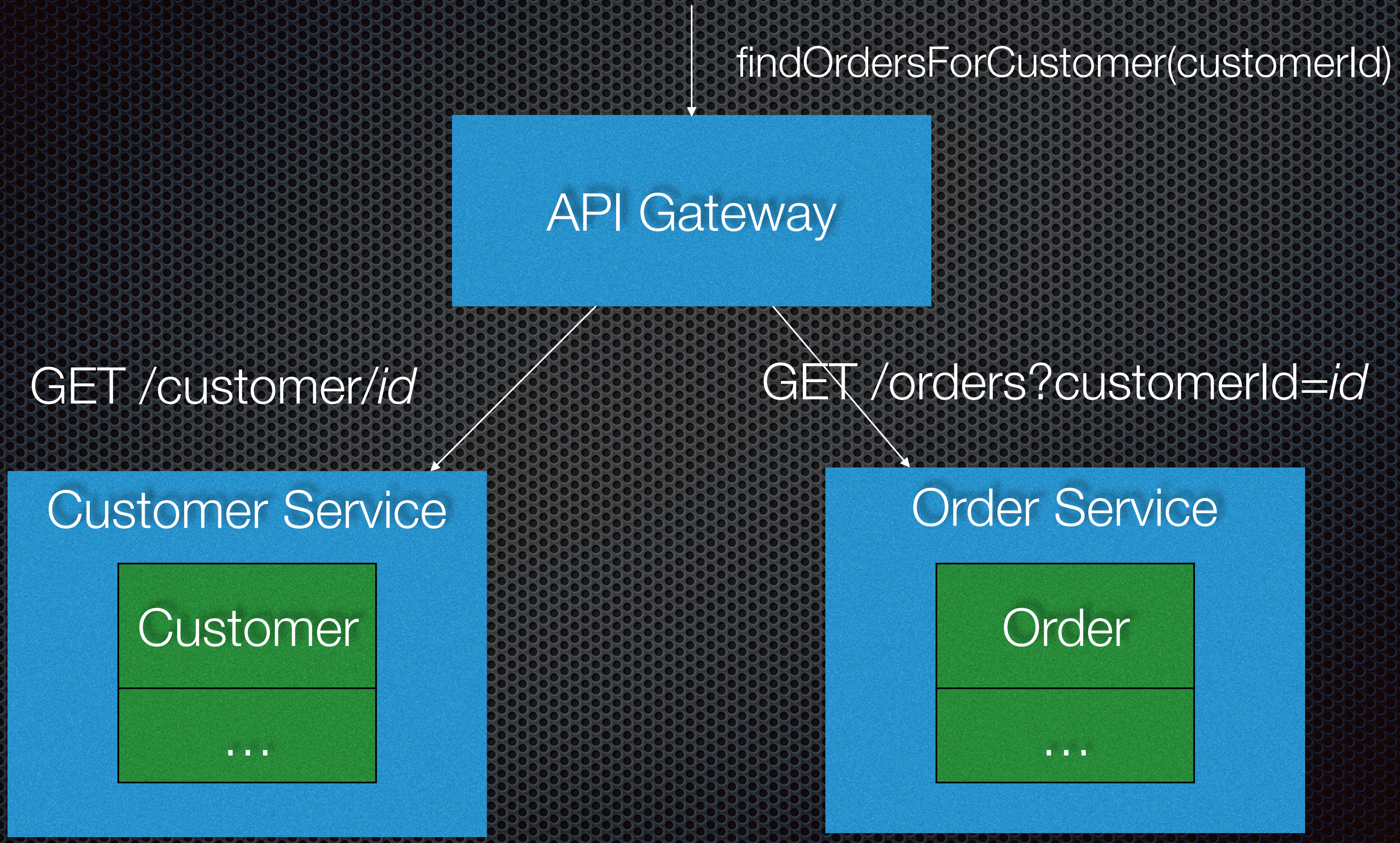
- Risk of smart sagas directing dumb services

Agenda

- Transactions, queries and microservices
- Managing transactions with sagas
- Implementing queries with CQRS
- Implementing transactional messaging

Queries often retrieve data owned by multiple services

API Composition pattern



https://microservices.io/patterns/data/api-composition.html

Find recent, valuable customers

Customer Service

Order Service

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

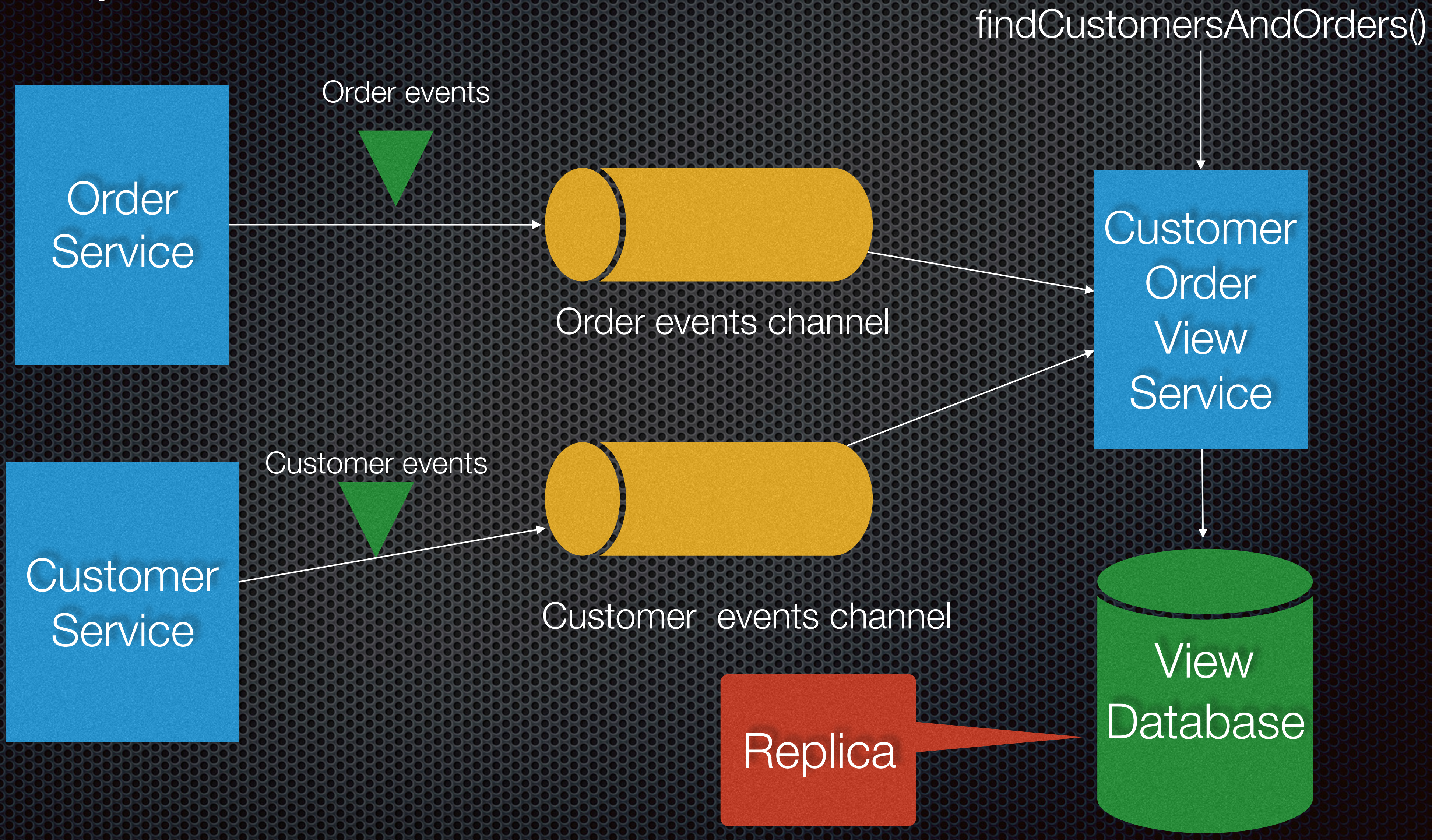
Not efficiently implemented using API Composition

API Composition would be inefficient

- 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

Using events to update a queryable replica = CQRS



https://microservices.io/patterns/data/cqrs.html

Persisting a customer and order history in MongoDB

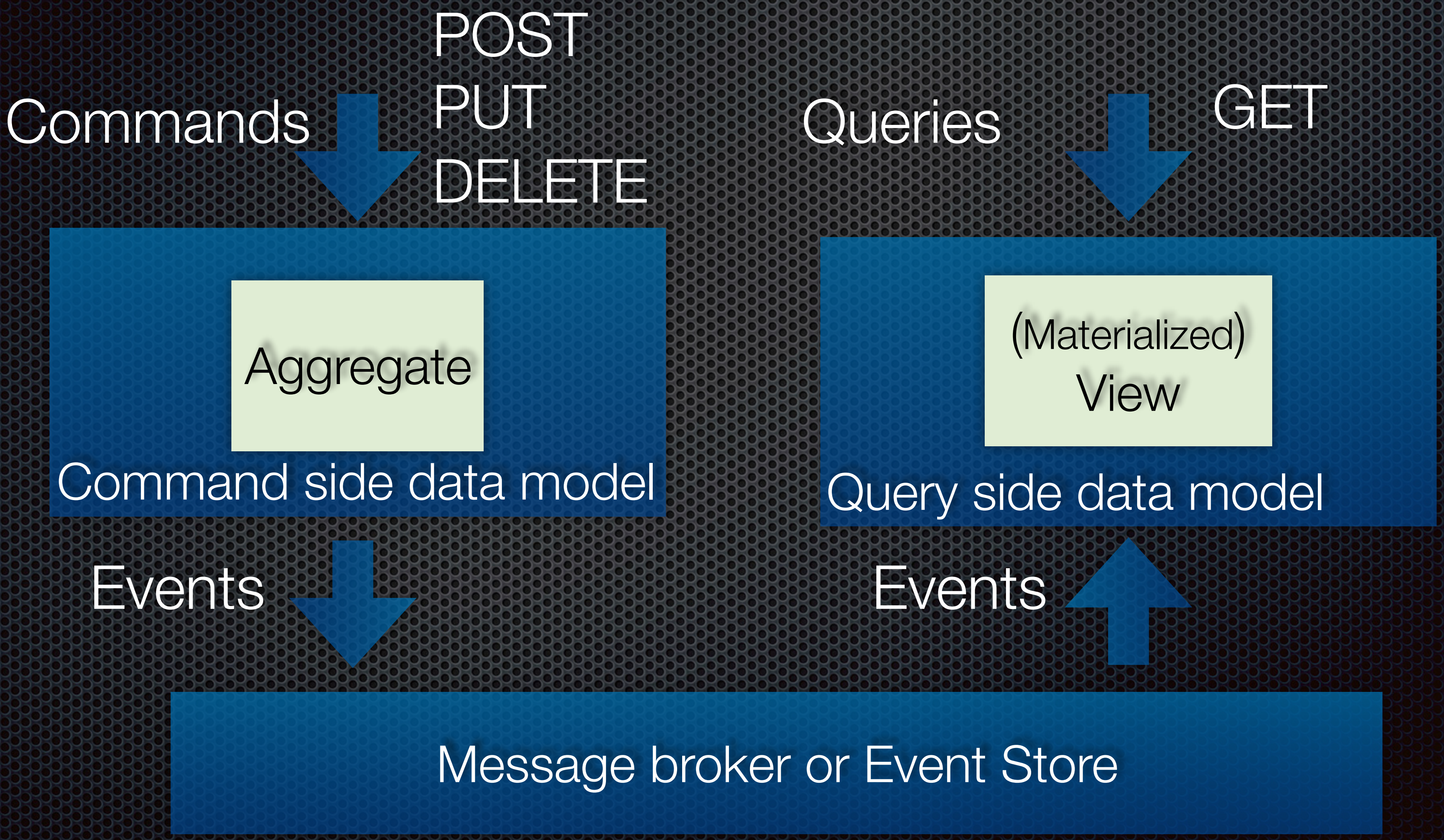
```
{
    "_id" : "0000014f9a45004b 0a00270000000000",
    "name" : "Fred",
    "creditLimit" : {
        "amount" : "2000"
    },
    "orders" : {
        "0000014f9a450063 0a00270000000000" : {
            "state" : "APPROVED",
            "orderId" : "0000014f9a450063 0a00270000000000",
         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

Customer information

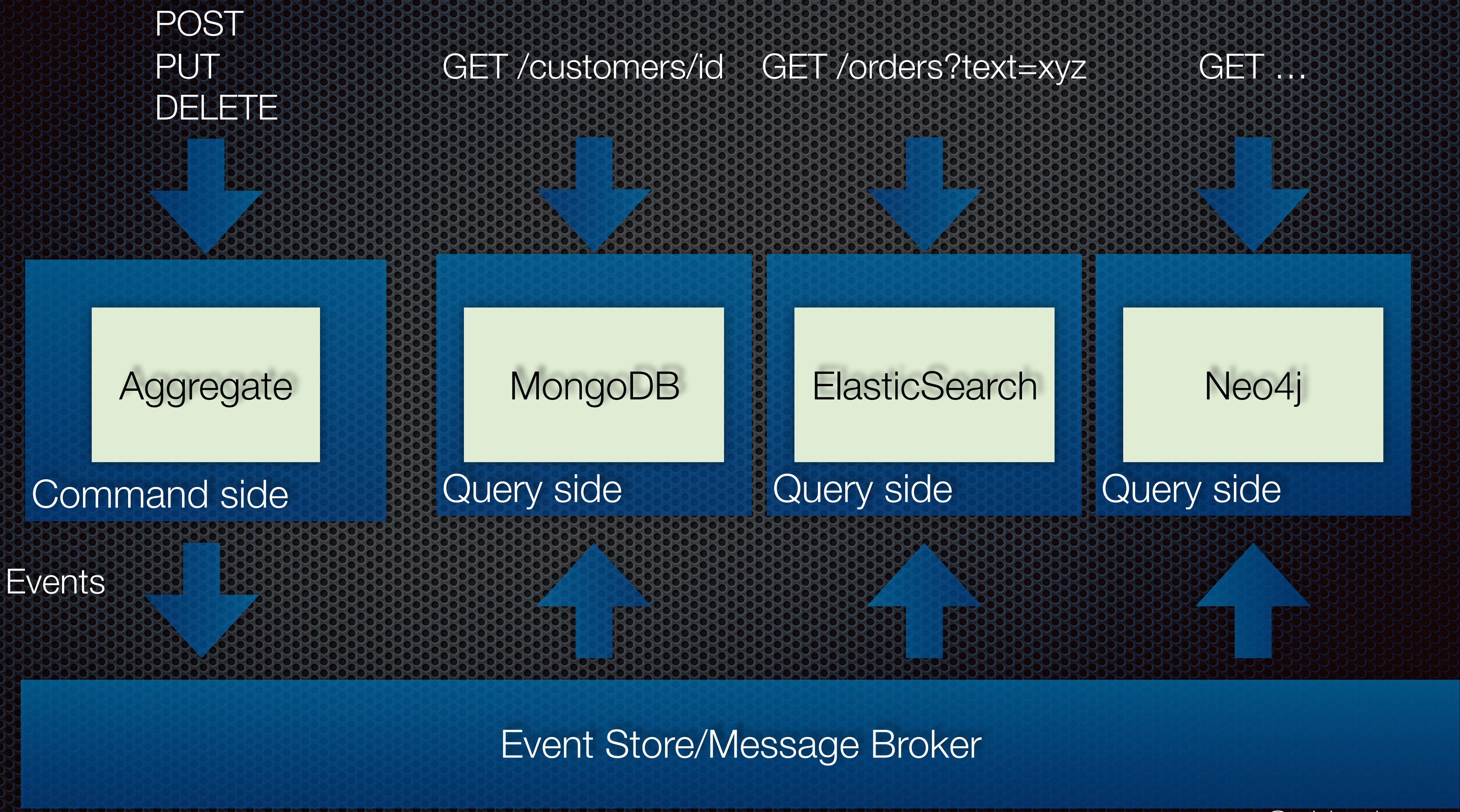
Order information

Denormalized = efficient lookup

Command Query Responsibility Segregation (CQRS)



Queries ⇒ database (type)



CQRS views are disposable

- Rebuild when needed from *source of truth*
- Using event sourcing
  - (Conceptually) replay all events from beginning of time
- Using traditional persistence
  - “ETL” from *source of truth* databases

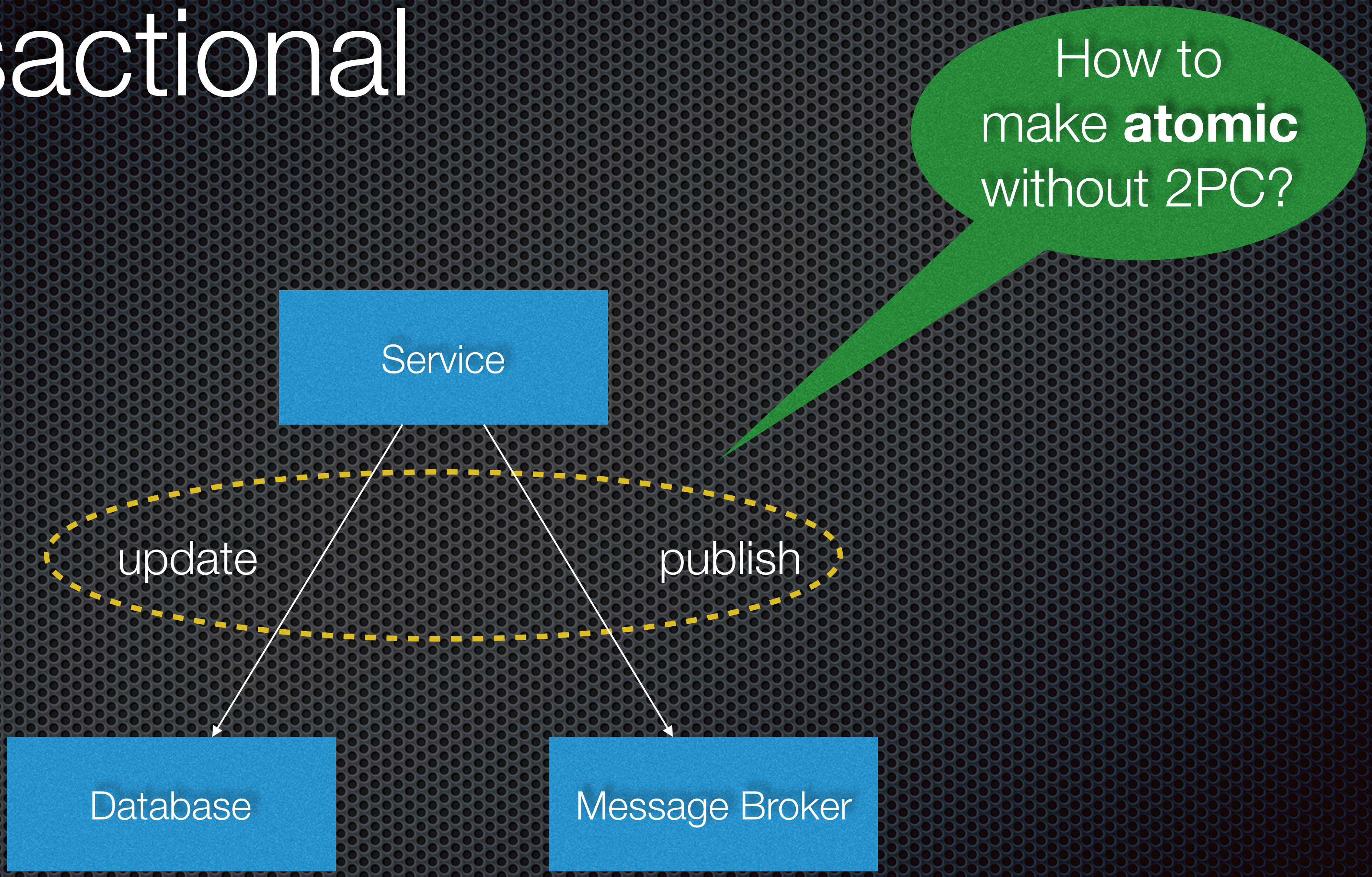
Handling replication lag

- Lag between updating command side and CQRS view
- Risk of showing stale data to user
- Either:
  - Update UI/client-side model without querying
  - Use updated aggregate version to “wait” for query view to be updated

Agenda

- Transactions, queries and microservices
- Managing transactions with sagas
- Implementing queries with CQRS
- Implementing transactional messaging

Messaging must be transactional

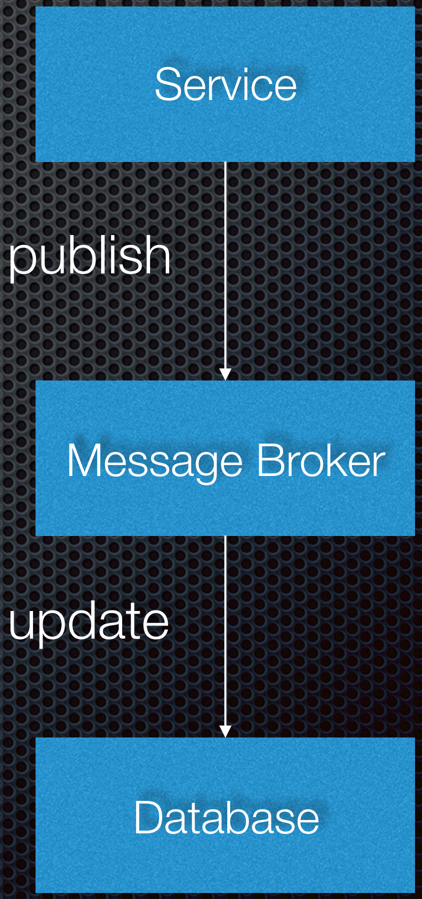


Publish to message broker first?

- Guarantees atomicity

BUT

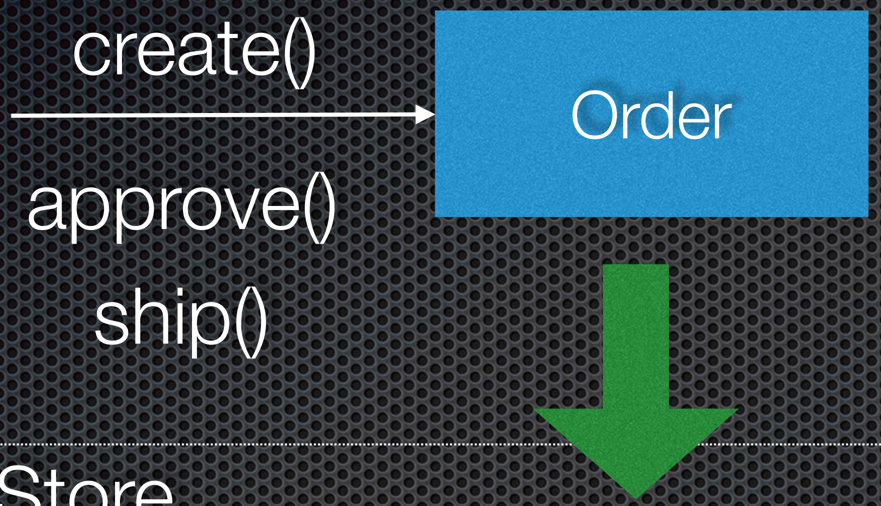
- Service can't read its own writes
- Difficult to write business logic



Option: Event sourcing
=
Event centric approach to business logic and persistence

http://eventuate.io/

Event sourcing: persists an object as a sequence of events

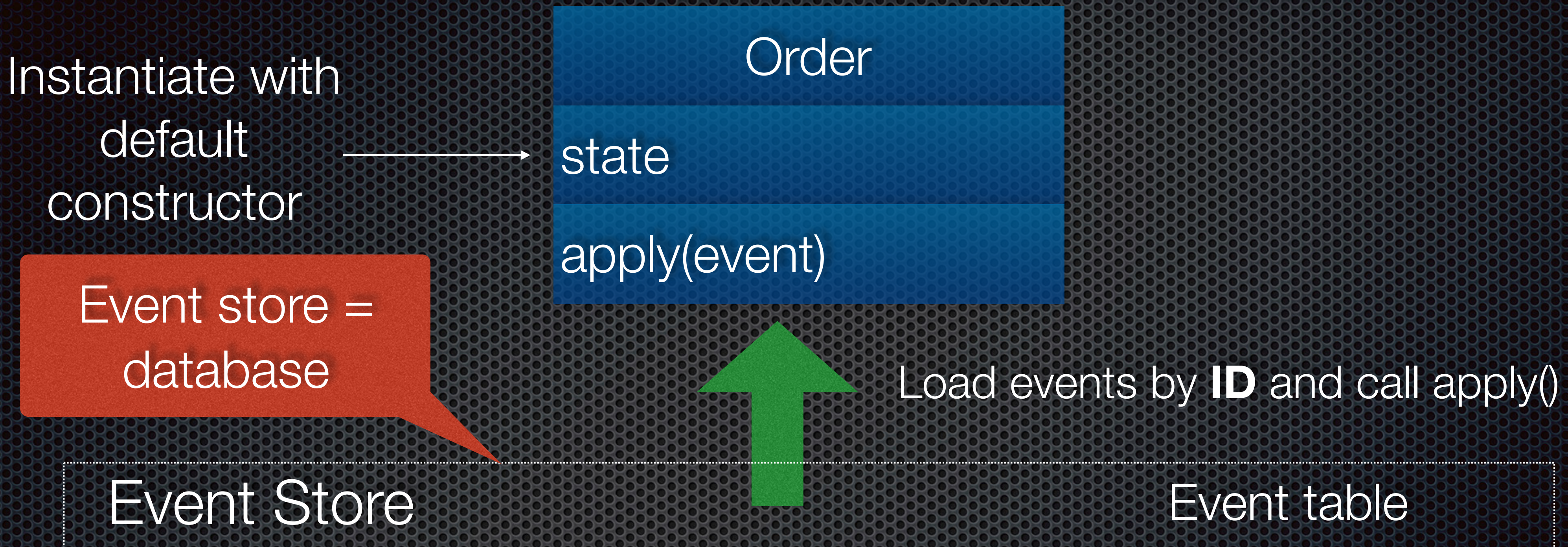


Event Store

Event table

Entity id	Entity type	Event id	Event type	Event data
101	Order	901	OrderCreated	…
101	Order	902	OrderApproved	…
101	Order	903	OrderShipped	…

Replay events to recreate in memory state

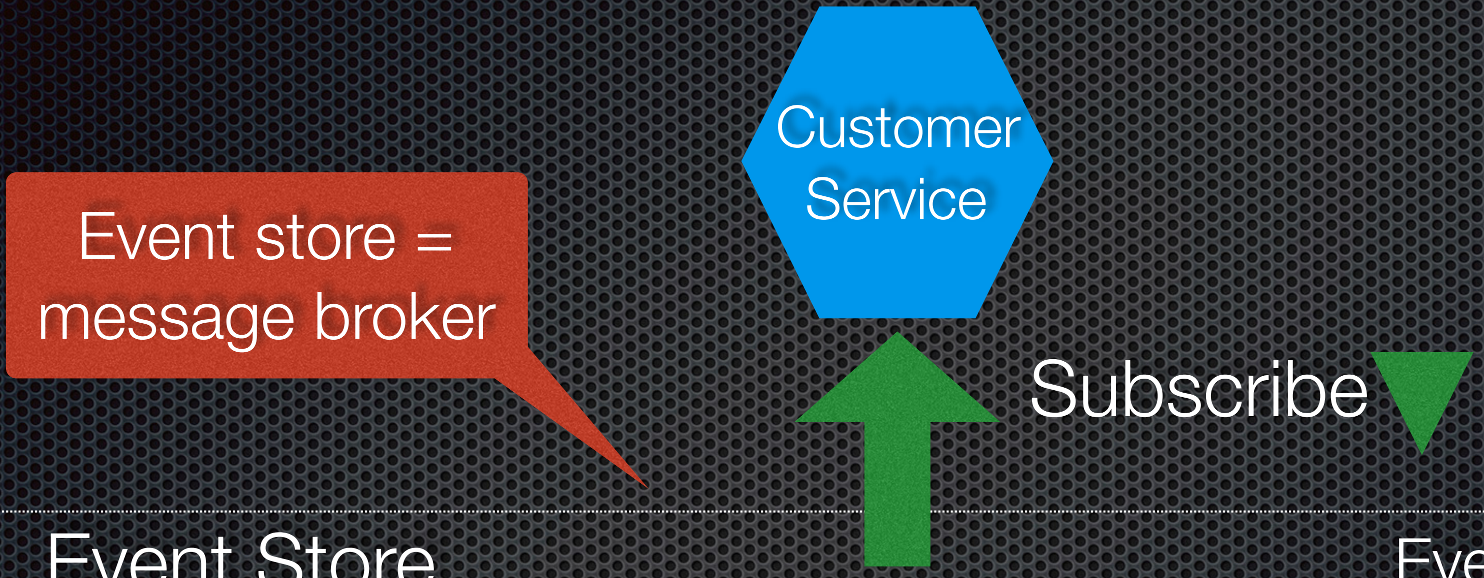


Entity id	Entity type	Event id	Event type	Event data
101	Order	901	OrderCreated	…
101	Order	902	OrderApproved	…
101	Order	903	OrderShipped	…

FYI:

Apache Kafka != event store

Event sourcing guarantees: state change ⇒ event is published



Event Store

Event table

Entity id	Entity type	Event id	Event type	Event data
101	Order	901	OrderCreated	…
101	Order	902	OrderApproved	…
101	Order	903	OrderShipped	…

@crichardson

Other benefits of event sourcing

Preserves history of domain objects

Supports temporal queries

Simplifies retroactive correction

Built-in auditing

Drawbacks of event sourcing

Unfamiliar programming model

Evolving the schema of long-lived events

Event store only supports PK-based access ⇒ requires CQRS ⇒ less consistent reads

Drawbacks of event sourcing

Good fit for choreography-based sagas
BUT orchestration is more challenging
⇒
Use event handler to translate event
into command/reply message

Option:

Traditional persistence (JPA, MyBatis,…)

+

Transactional outbox pattern

https://github.com/eventuate-tram/eventuate-tram-core

Spring Data for JPA example

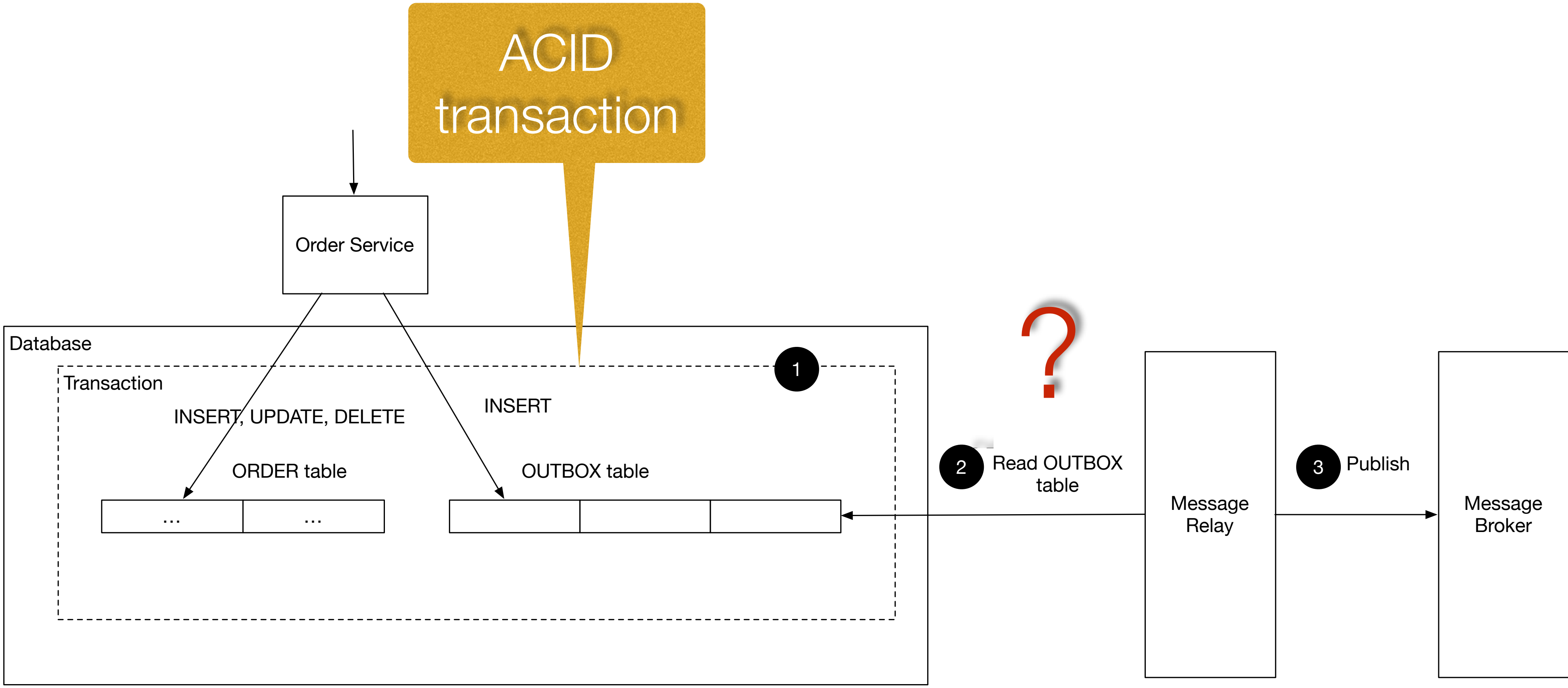
```java
public class OrderService {

  @Autowired
  private DomainEventPublisher domainEventPublisher;

  @Autowired
  private OrderRepository orderRepository;

  @Transactional
  public Order createOrder(OrderDetails orderDetails) {
    Order order = Order.createOrder(orderDetails);
    orderRepository.save(order);
    domainEventPublisher.publish(Order.class,
            order.getId(),
            singletonList(new OrderCreatedEvent(order.getId(), orderDetails)));
    return order;
  }
}
```

Save order

Publish event

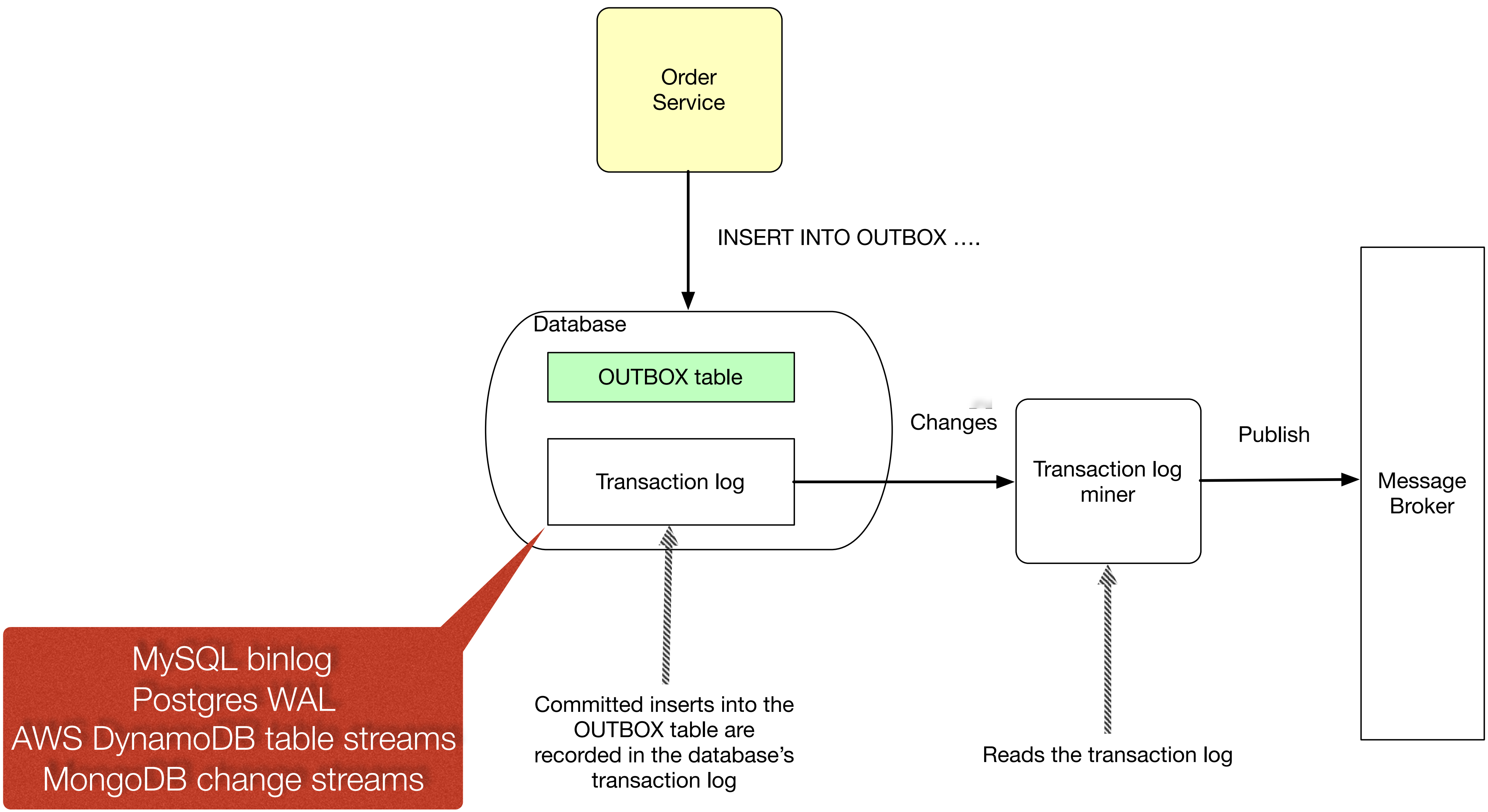
http://eventuate.io/exampleapps.html

Transactional Outbox pattern



- https://microservices.io/patterns/data/transactional-outbox.html
- https://eventuate.io/

Transaction log tailing



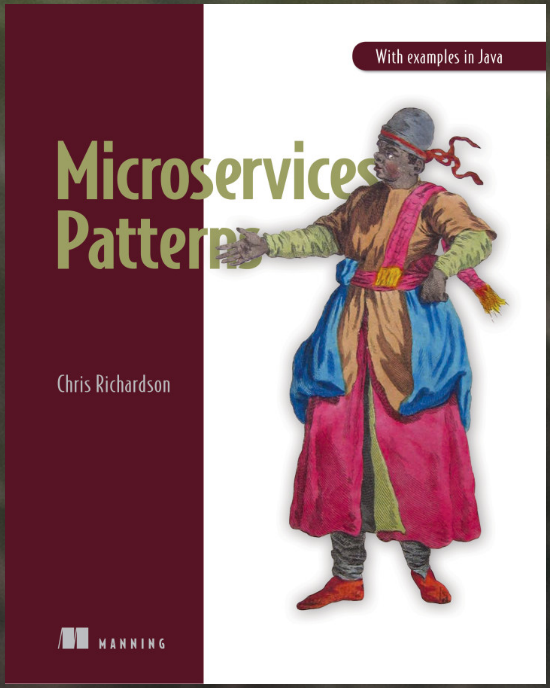
Polling the message table

- Simple
- Works for all databases
- BUT what about polling frequency

Summary

- Use asynchronous messaging to solve distributed data management problems
- Services publish events to implement
  - choreography-based sagas
  - queries using CQRS views
- Services send command/reply messages to implement orchestration-based sagas
- Services must atomically update state and send messages
  - Event sourcing
  - Transactional outbox

@crichardson chris@chrisrichardson.net



Get a 40% discount

Questions?

http://bit.ly/gotochgo2019-microservices