Developing Microservices with Aggregates

Chris Richardson

Founder of Eventuate.io

Founder of the original CloudFoundry.com

Author of POJOs in Action

@crichardson

chris@chrisrichardson.net

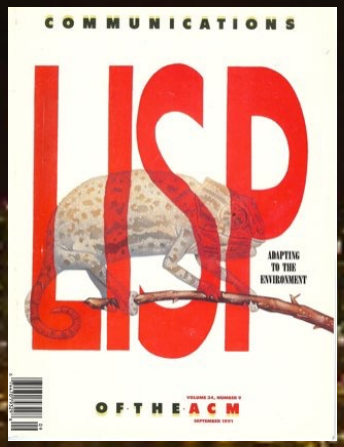
http://eventuate.io

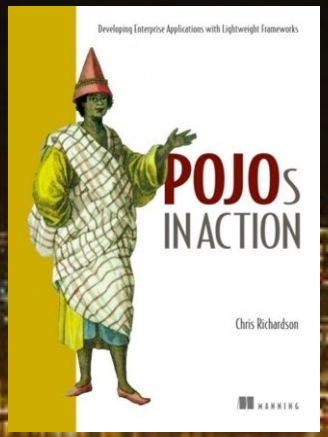
SpringOne Platform

Presentation goal

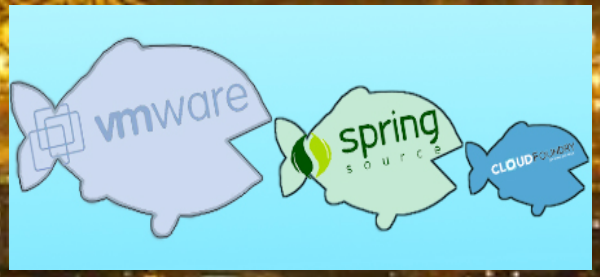
Show how
Domain Driven Design Aggregates
and Microservices
are a perfect match

About Chris


COMMUNICATIONS
LISP
ADAPTING TO THE ENVIRONMENT
OF THE ACM


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


CLOUDFOUNDRY
SIGN IN
SIGN UP
SYSTEM ALERT. PLEASE READ: Cloud Foundry will be moving to a new URL.
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
APPLICATION DEMO
Deploying Web Applications To Amazon EC2 with Cloud Foundry


vmware
spring
source
CLOUDFOUNDRY

About Chris

Consultant and trainer focusing on modern application architectures including microservices (http://www.chrisrichardson.net/)

About Chris

Founder of a startup that is creating a platform that makes it easier for developers to write transactional microservices

(http://eventuate.io)

For more information

http://learnmicroservices.io

Agenda

- The problem with Domain Models and microservices
- Overview of aggregates
- Maintaining consistency between aggregates
- Using event sourcing with Aggregates
- Example application

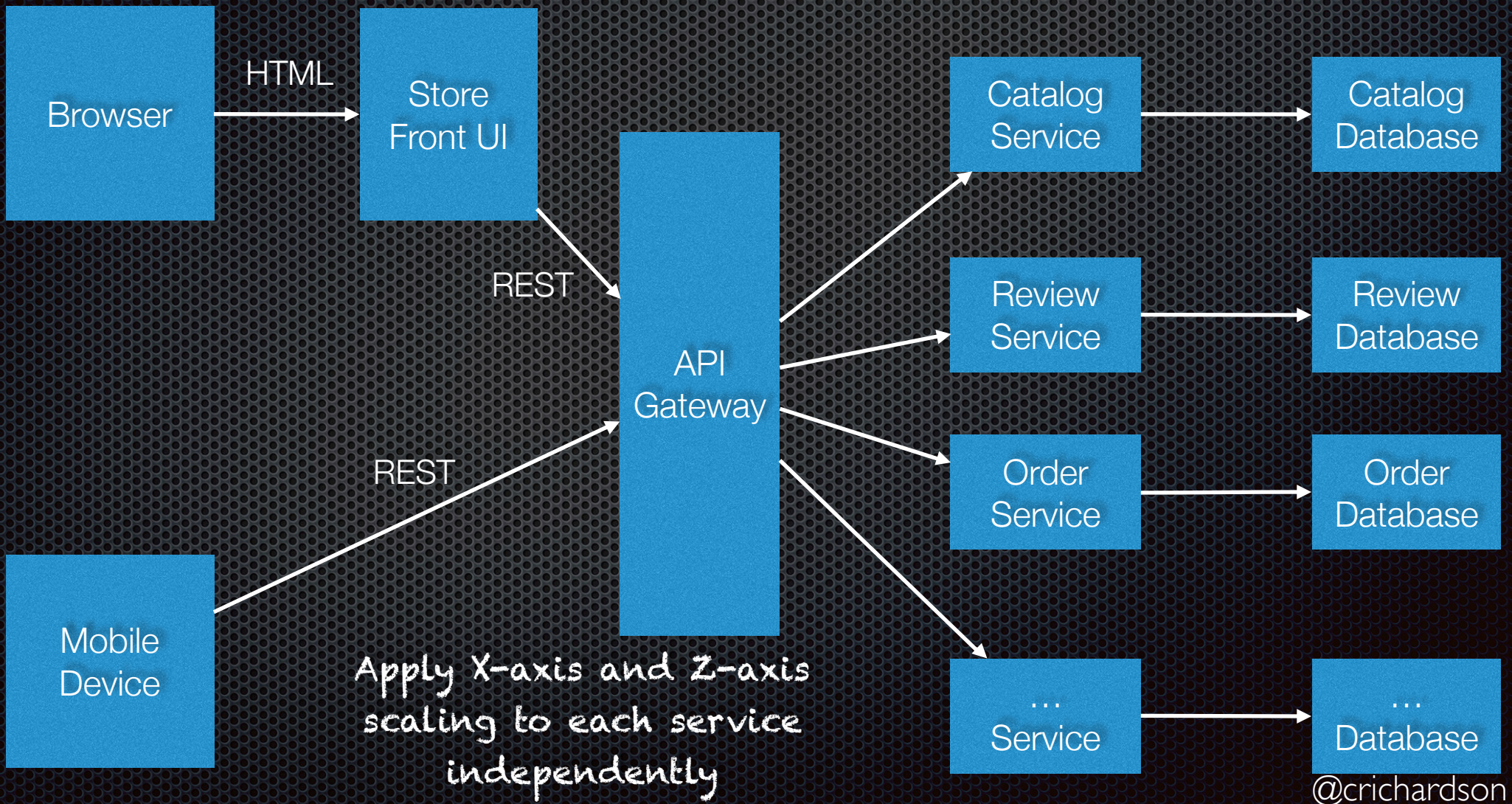
The Microservice architecture tackles complexity through modularization

Microservice

=

Business capability

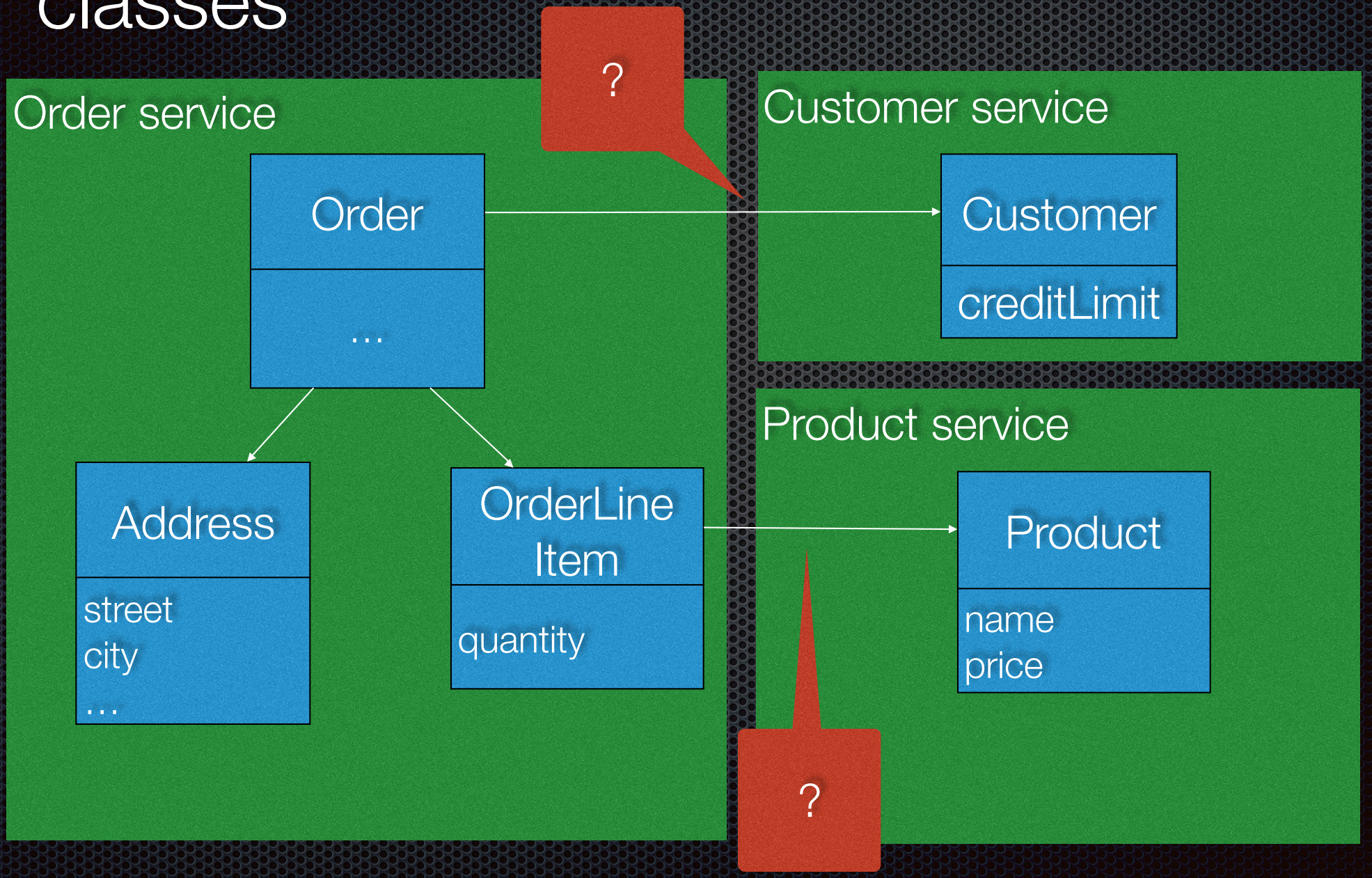
Microservice architecture



Service boundaries
enforce modularity

But there are challenges...

Domain model = tangled web of classes



Reliance on ACID transactions to enforce invariants

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

Simple and ACID 😀

But it violates encapsulation…

```
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

.. and requires 2PC

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

2PC is not a viable option

- Guarantees consistency

BUT

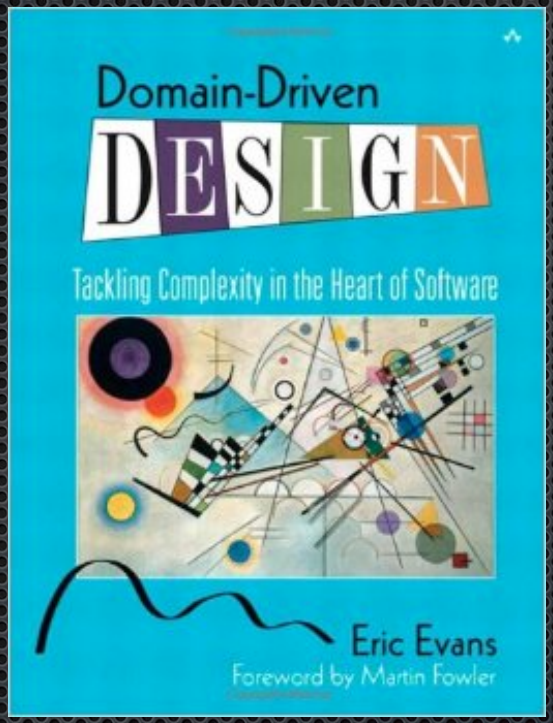
- 2PC is best avoided
- Not supported by many NoSQL databases etc.
- CAP theorem ⇒ 2PC impacts availability
- ….

Doesn't fit with the
NoSQL DB "transaction" model

Agenda

- The problem with Domain Models and microservices
- Overview of aggregates
- Maintaining consistency between aggregates
- Using event sourcing with Aggregates
- Example application

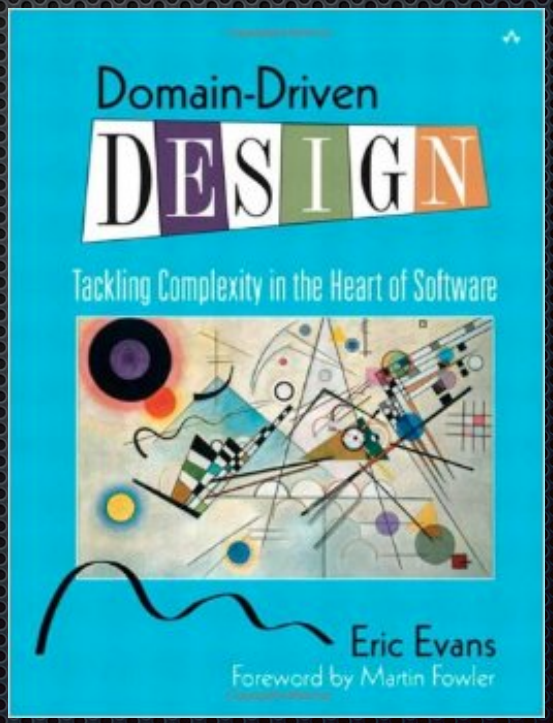
Domain Driven Design: read it!



Domain Driven Design - building blocks

- Entity
- Value object
- Services
- Repositories
- Aggregates

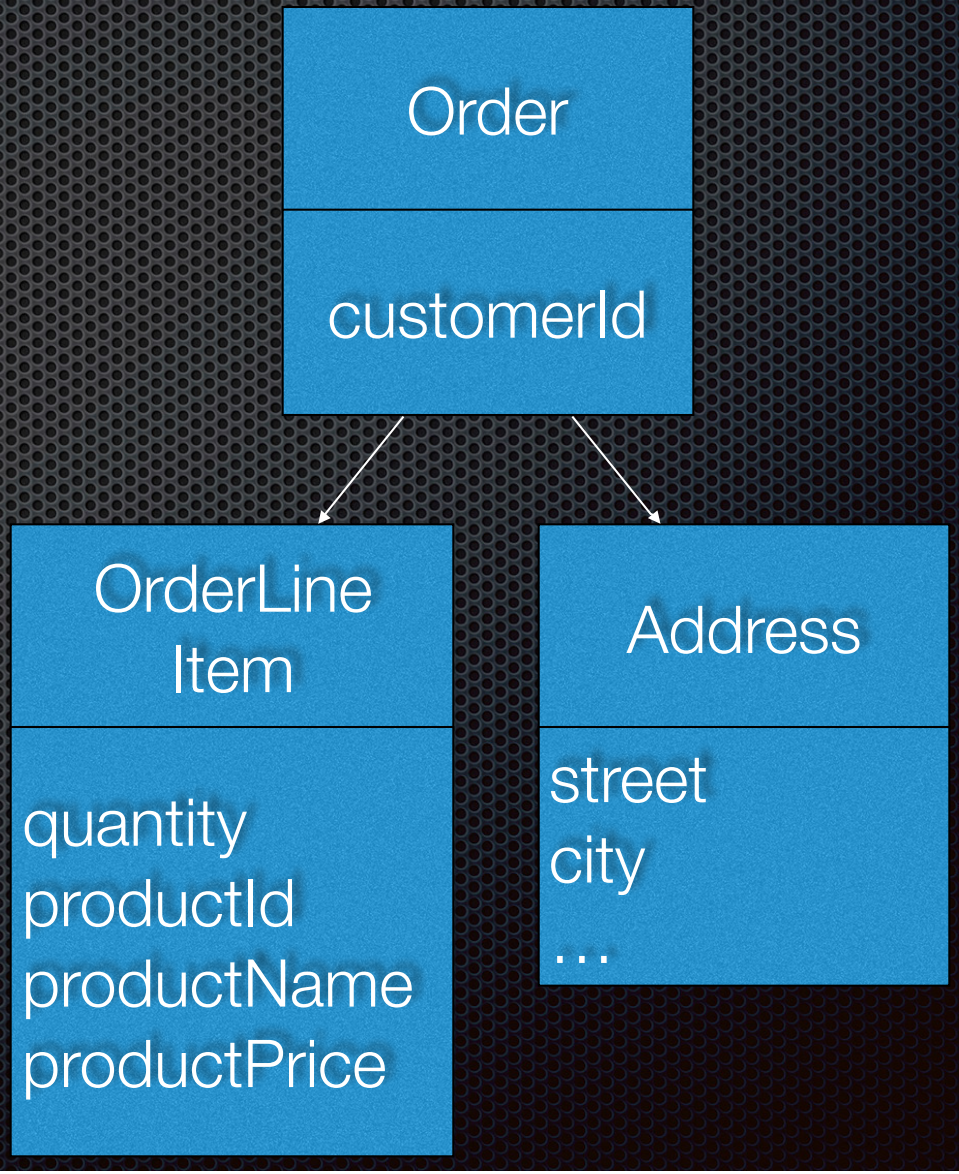
Adopted



Ignored
(at least by me)

About Aggregates

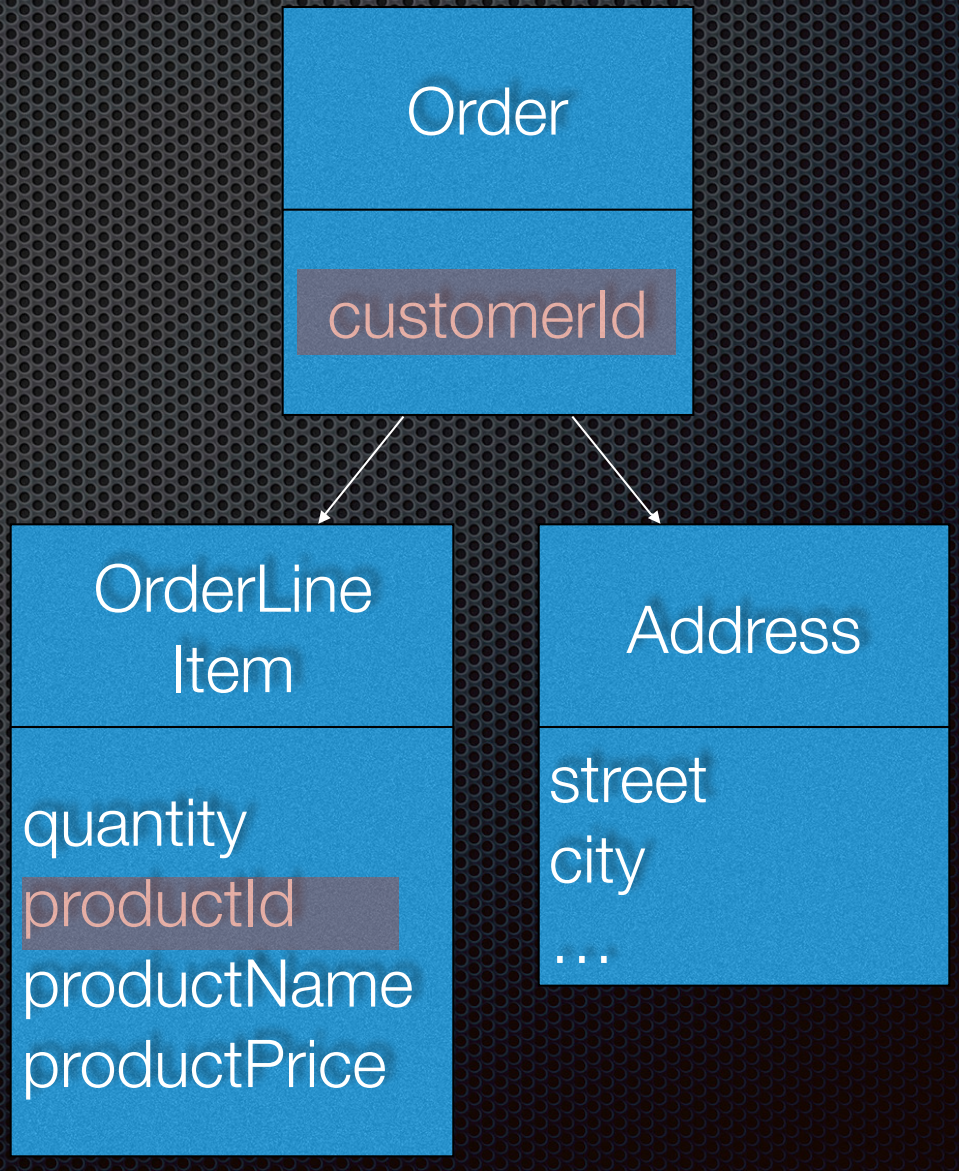
- Cluster of objects that can be treated as a unit
- Graph consisting of a root entity and one or more other entities and value objects
- Typically business entities are Aggregates, e.g. customer, Account, Order, Product, ….



Aggregate: rule #1

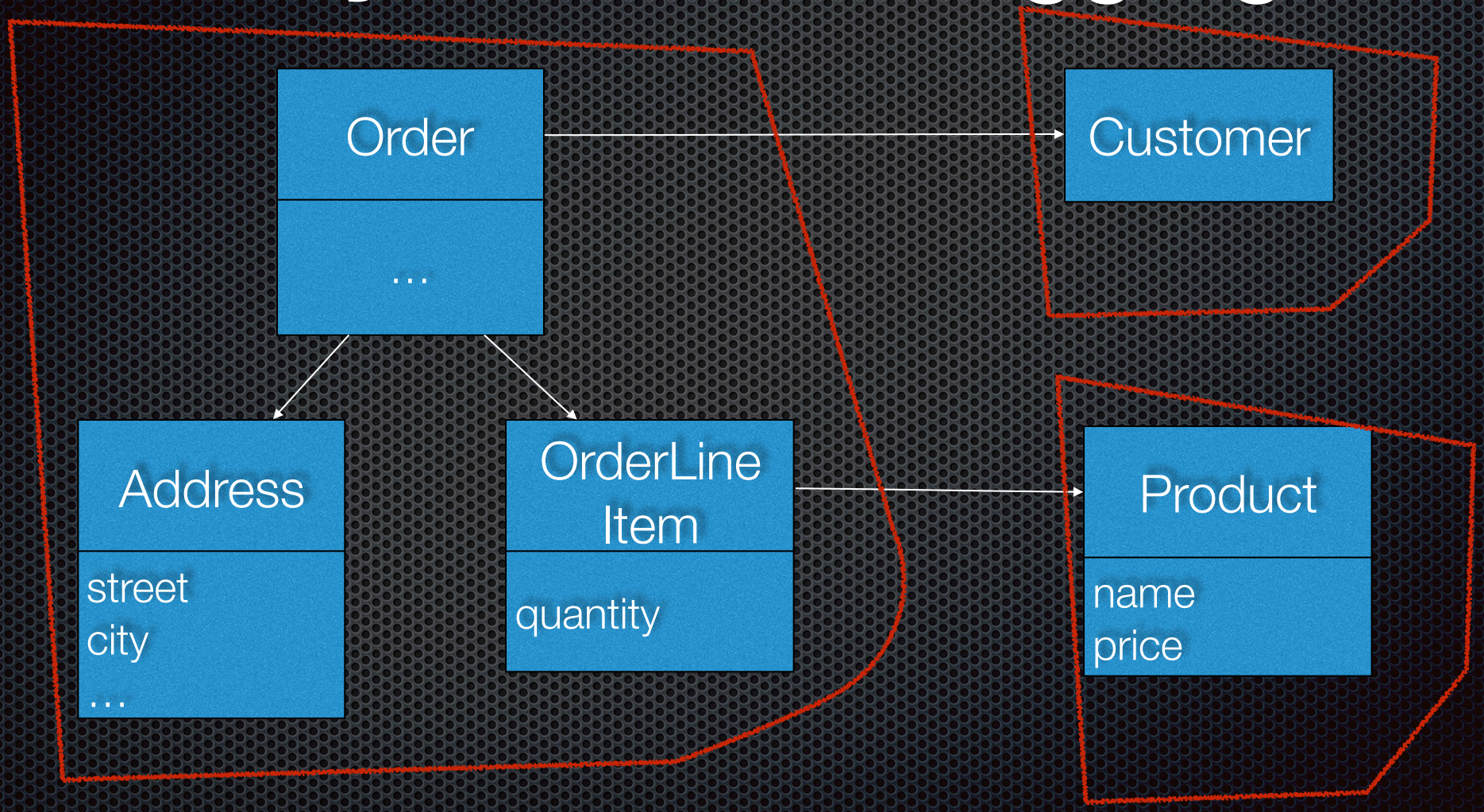
Reference other aggregate roots via **identity**

(primary key)

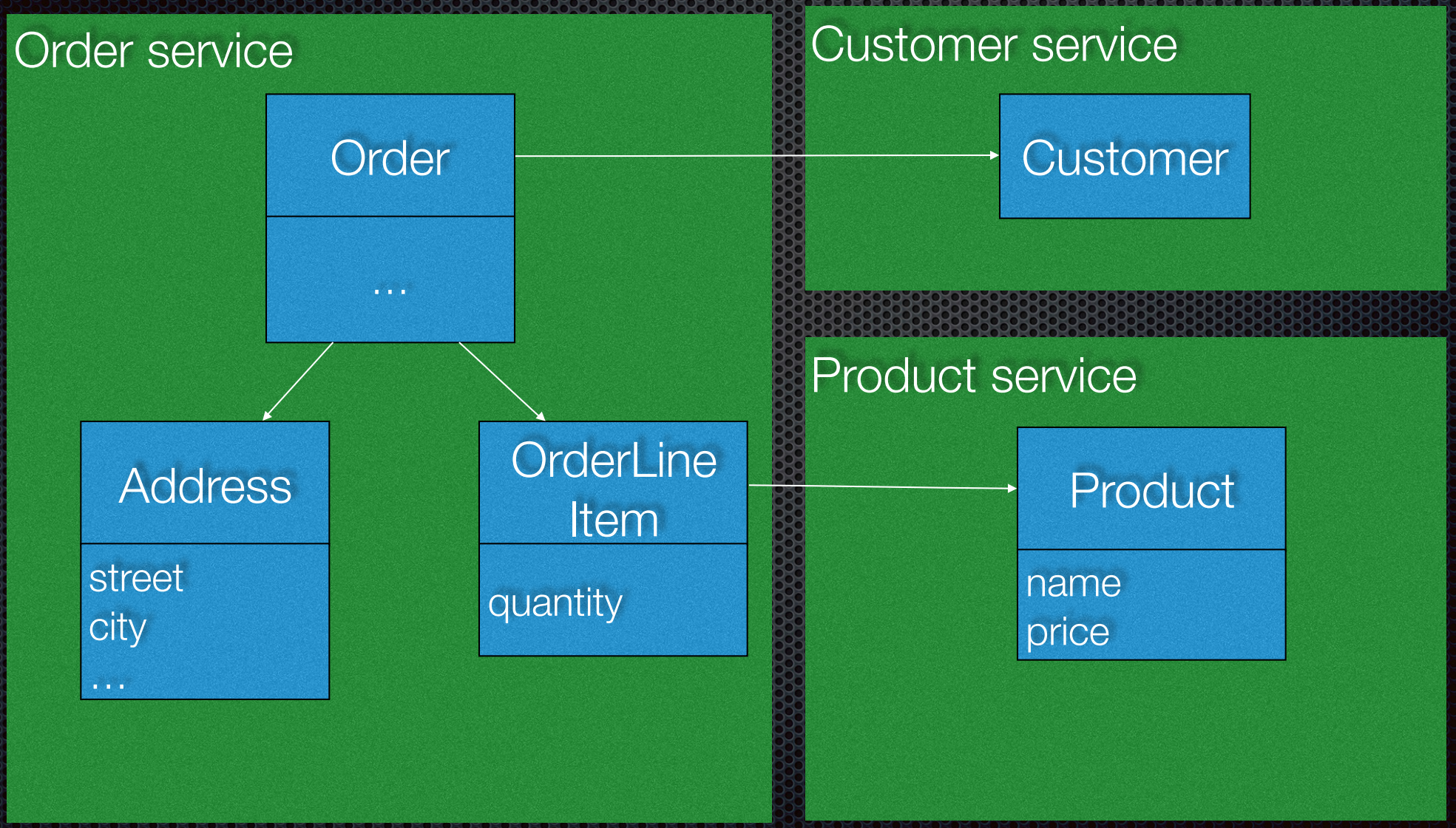


Foreign keys in a Domain Model?!?

Domain model = collection of **loosely** connected aggregates



Easily partition into microservices



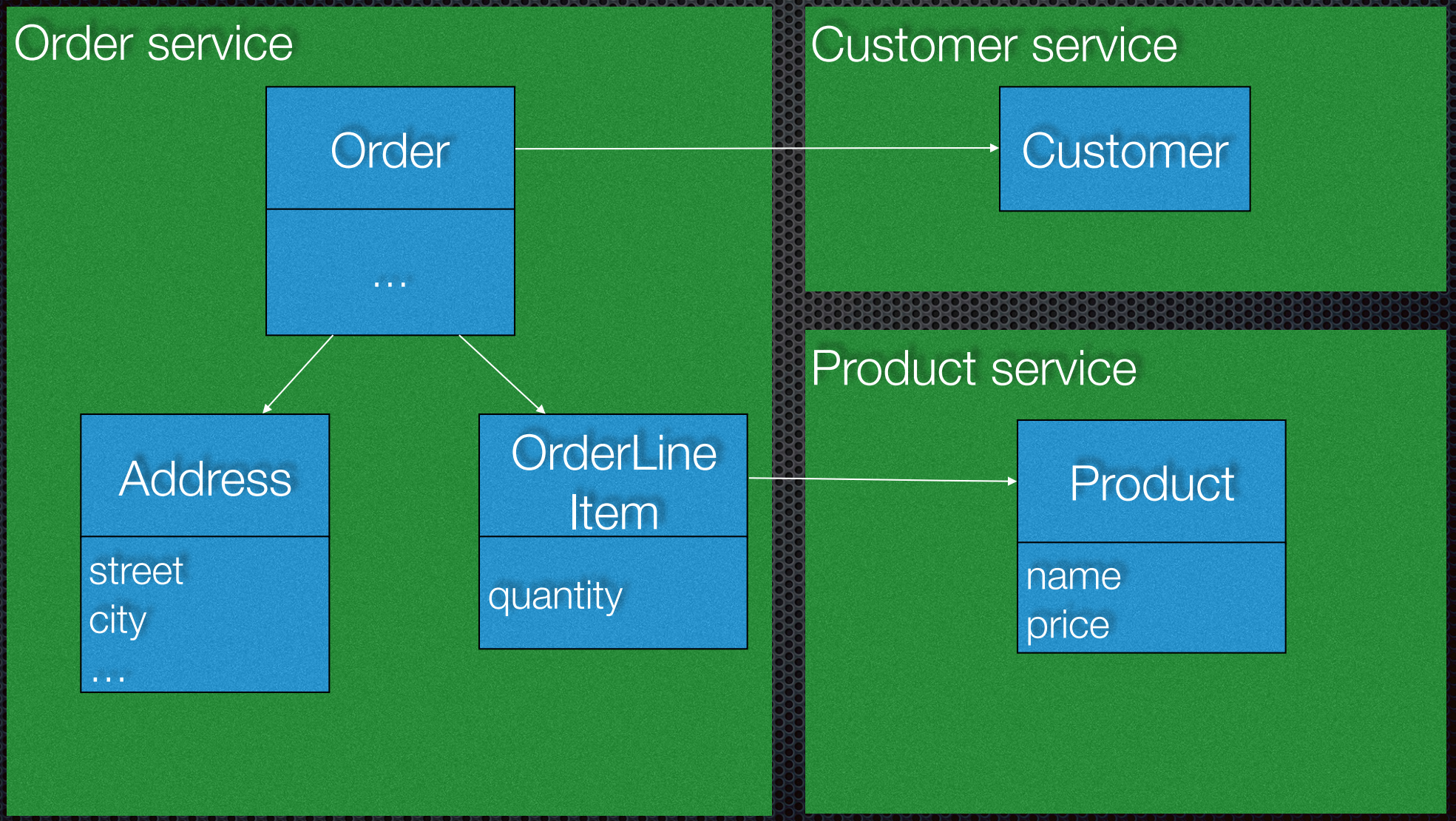
Aggregate rule #2

Transaction

=

processing **one** command by **one** aggregate

Transaction scope = service



Transaction scope

=

NoSQL database

“transaction”

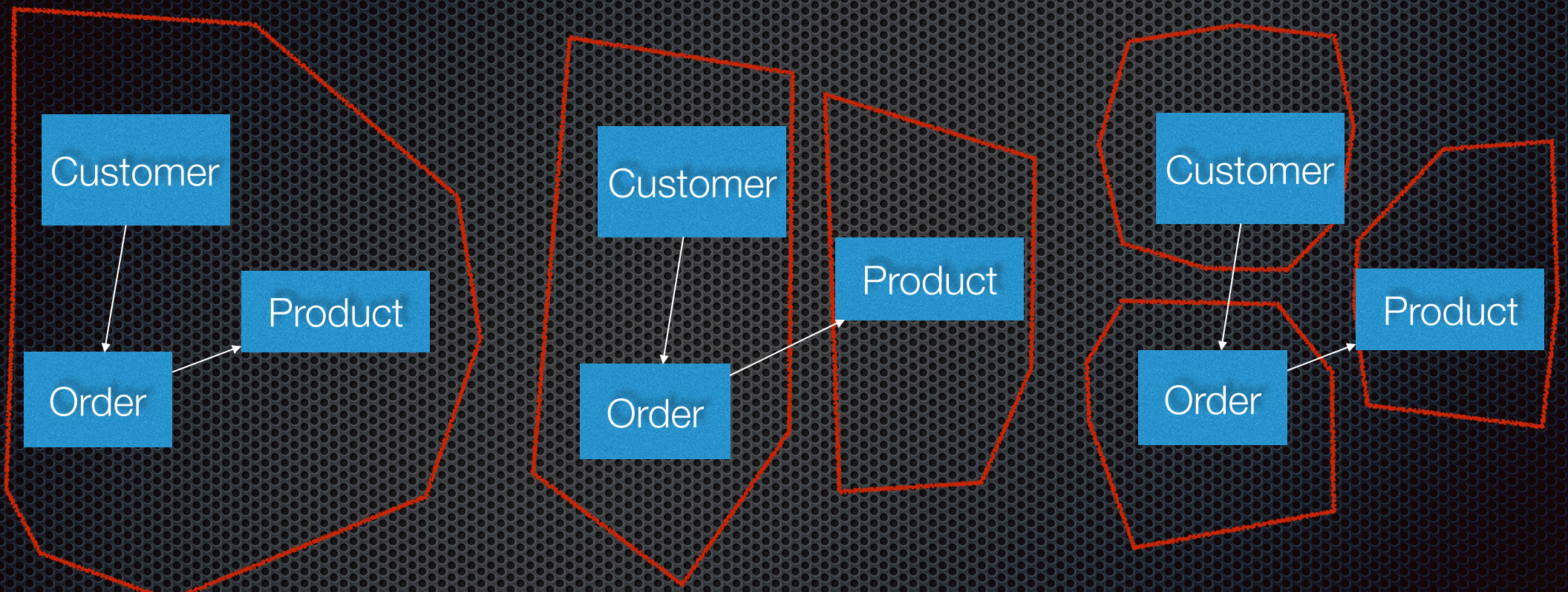
Aggregate granularity

- If an update must be atomic then it must be handled by a single aggregate

Therefore

- Aggregate granularity is important

Aggregate granularity



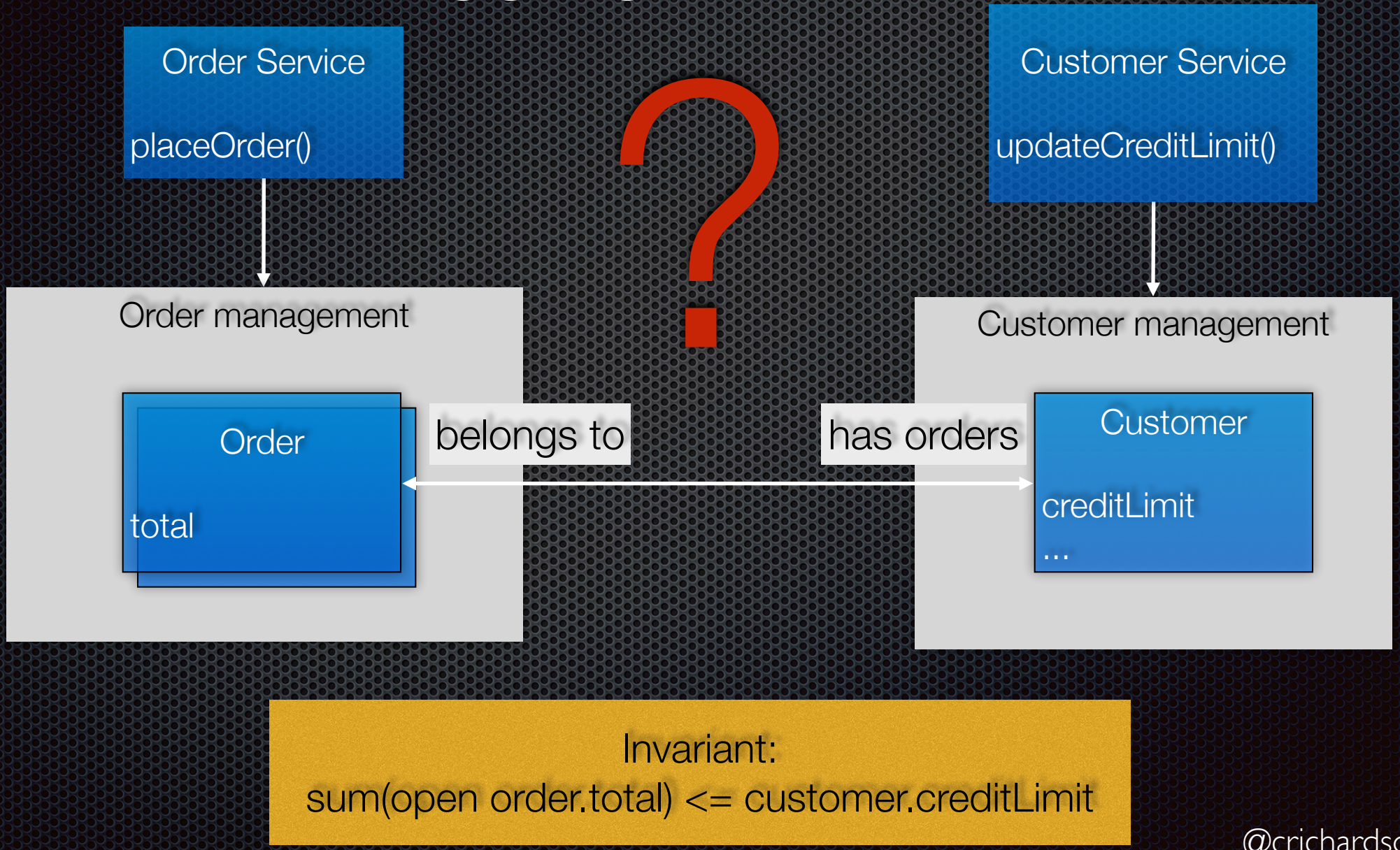
Consistency

Scalability/
User experience

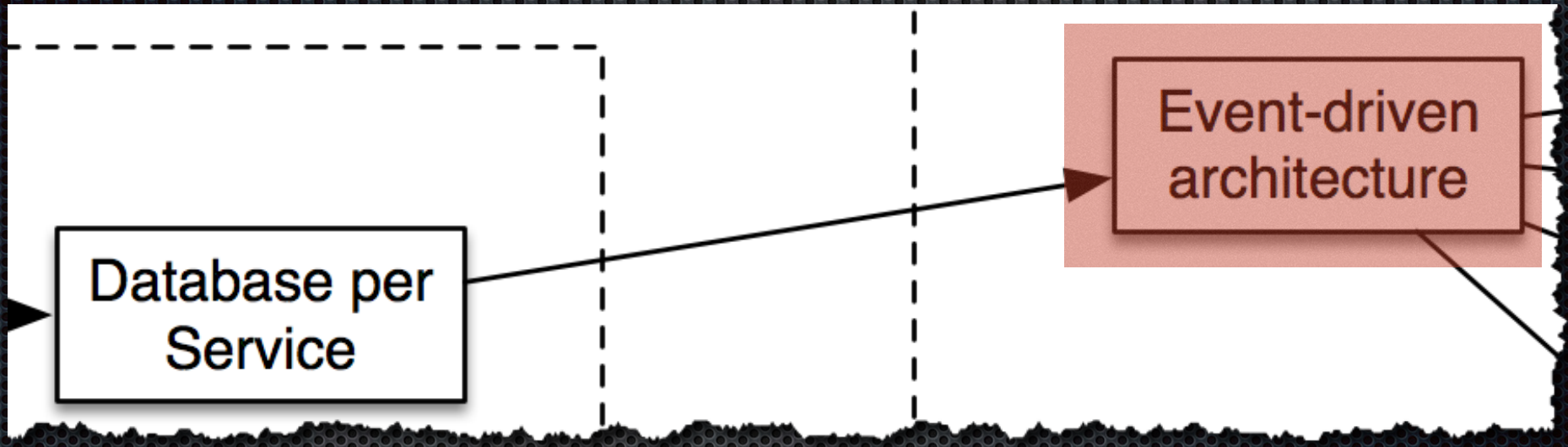
Agenda

- The problem with Domain Models and microservices
- Overview of aggregates
- Maintaining consistency between aggregates
- Using event sourcing with Aggregates
- Example application

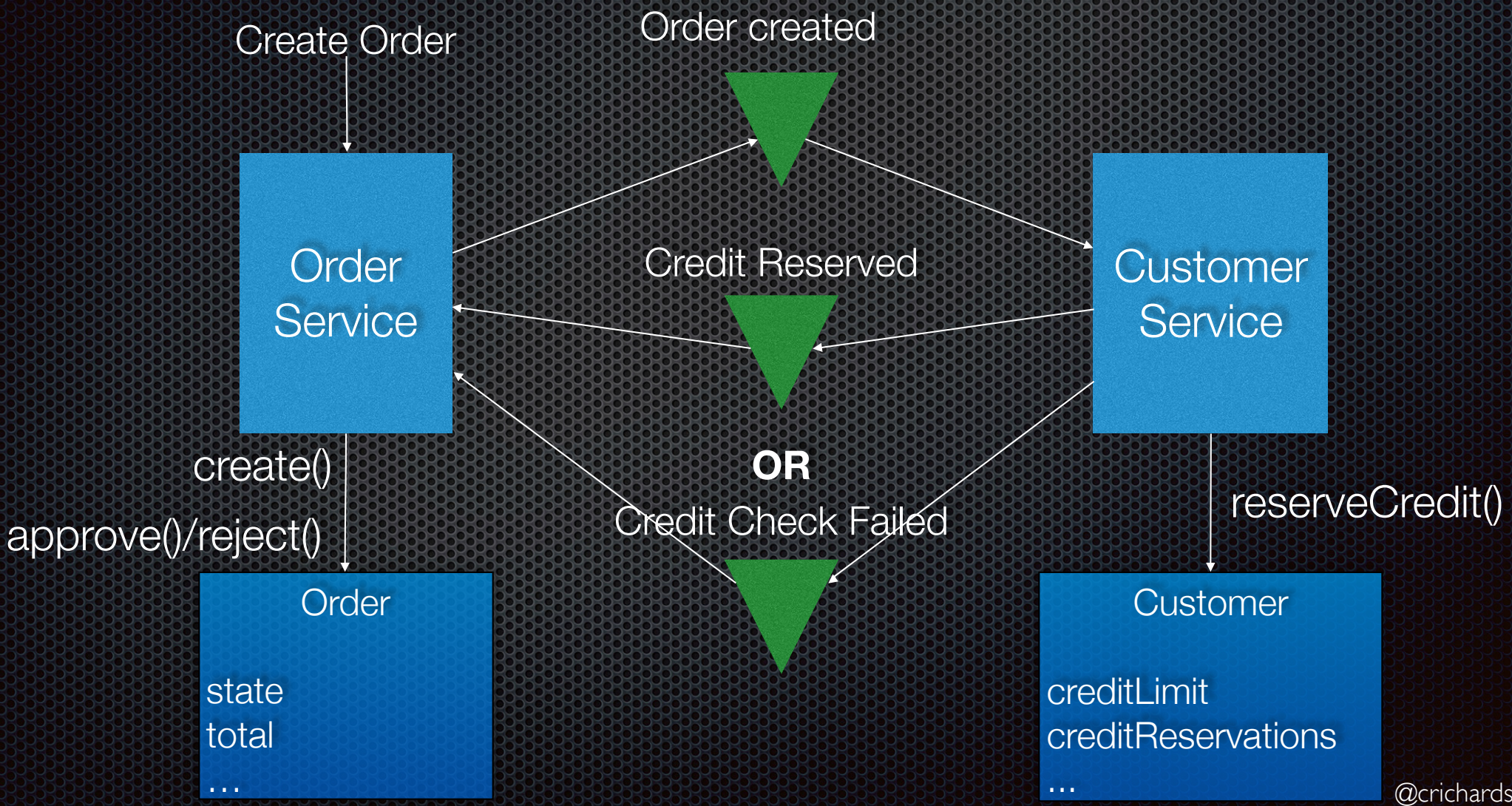
How to maintain data consistency between aggregates?



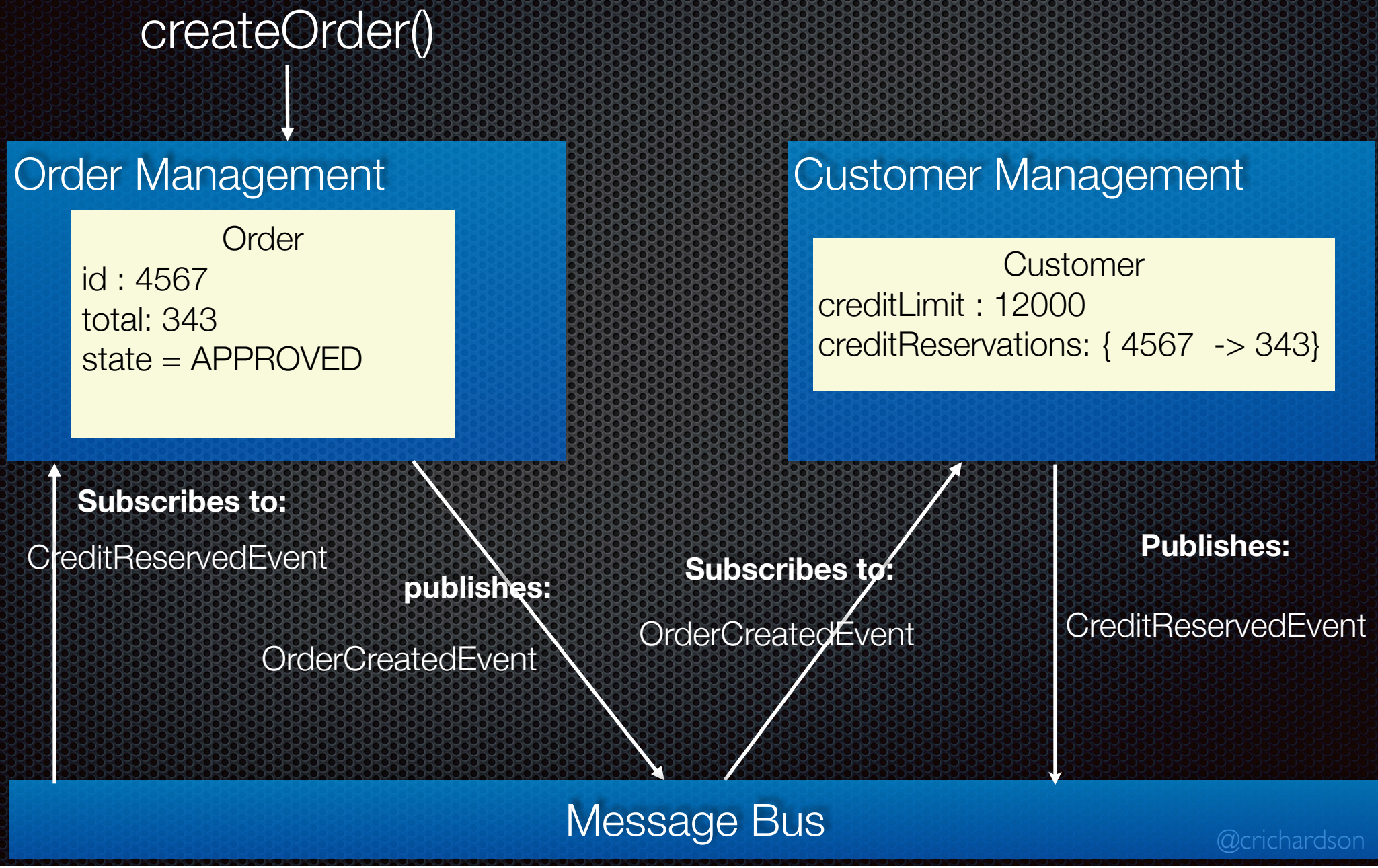
Event-driven architecture



Use event-driven, eventually consistent order processing



Eventually consistent credit checking



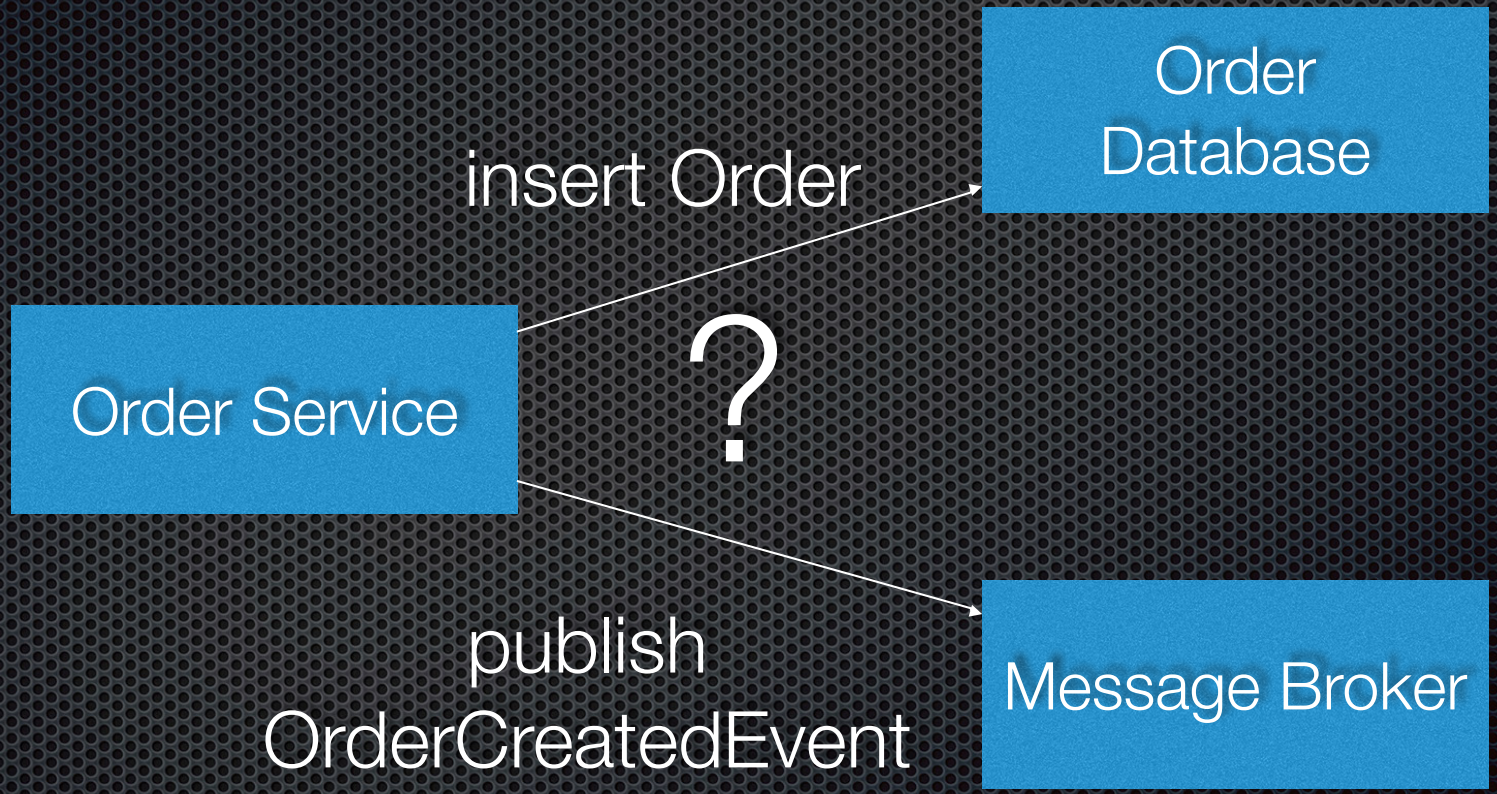
Complexity of compensating transactions

- ACID transactions can simply rollback

BUT

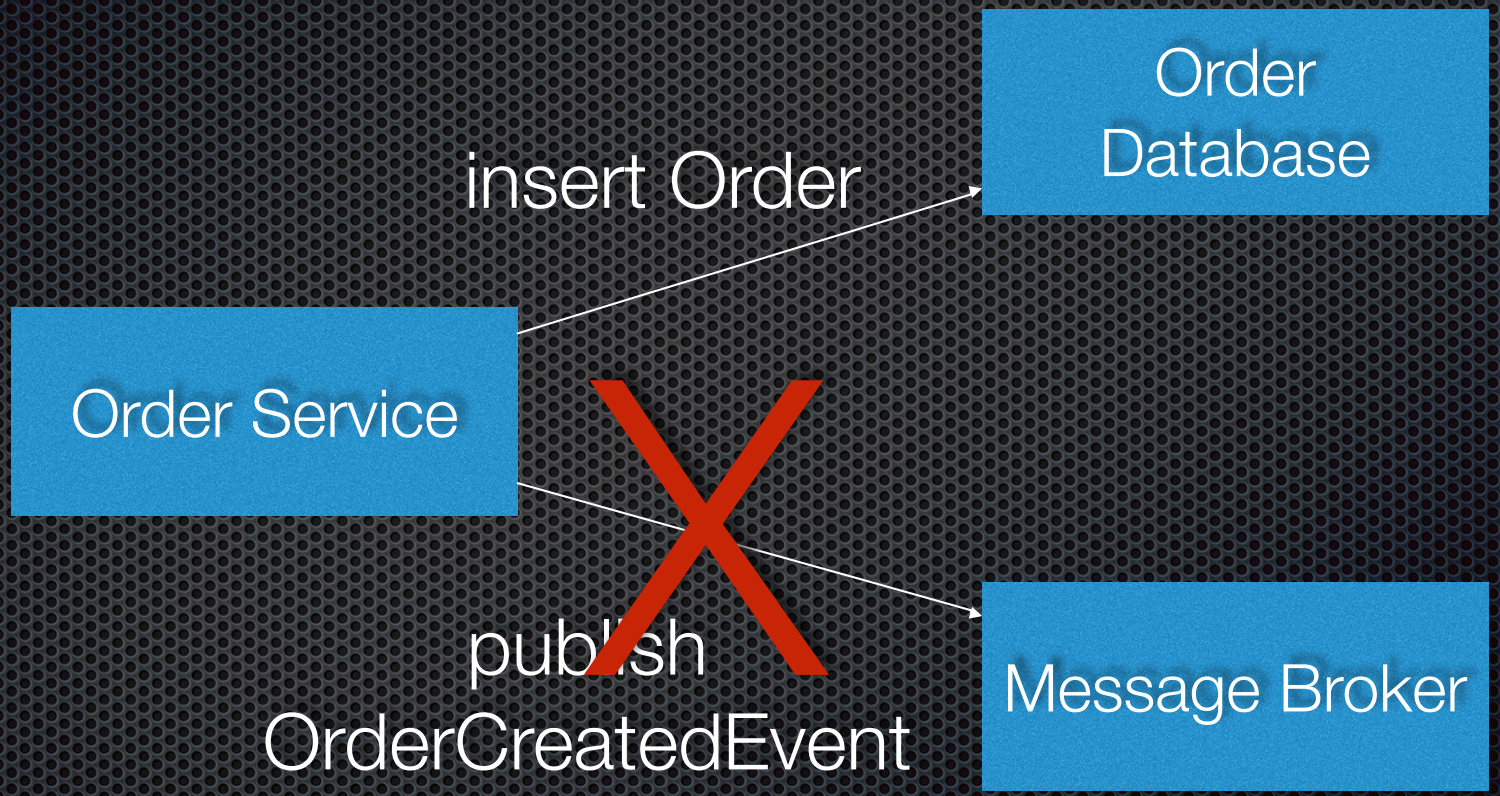
- Developer must write application logic to “rollback” eventually consistent transactions
- For example:
  - CreditCheckFailed => cancel Order
  - Money transfer destination account closed => credit source account
- Careful design required!

How atomically update database and publish an event



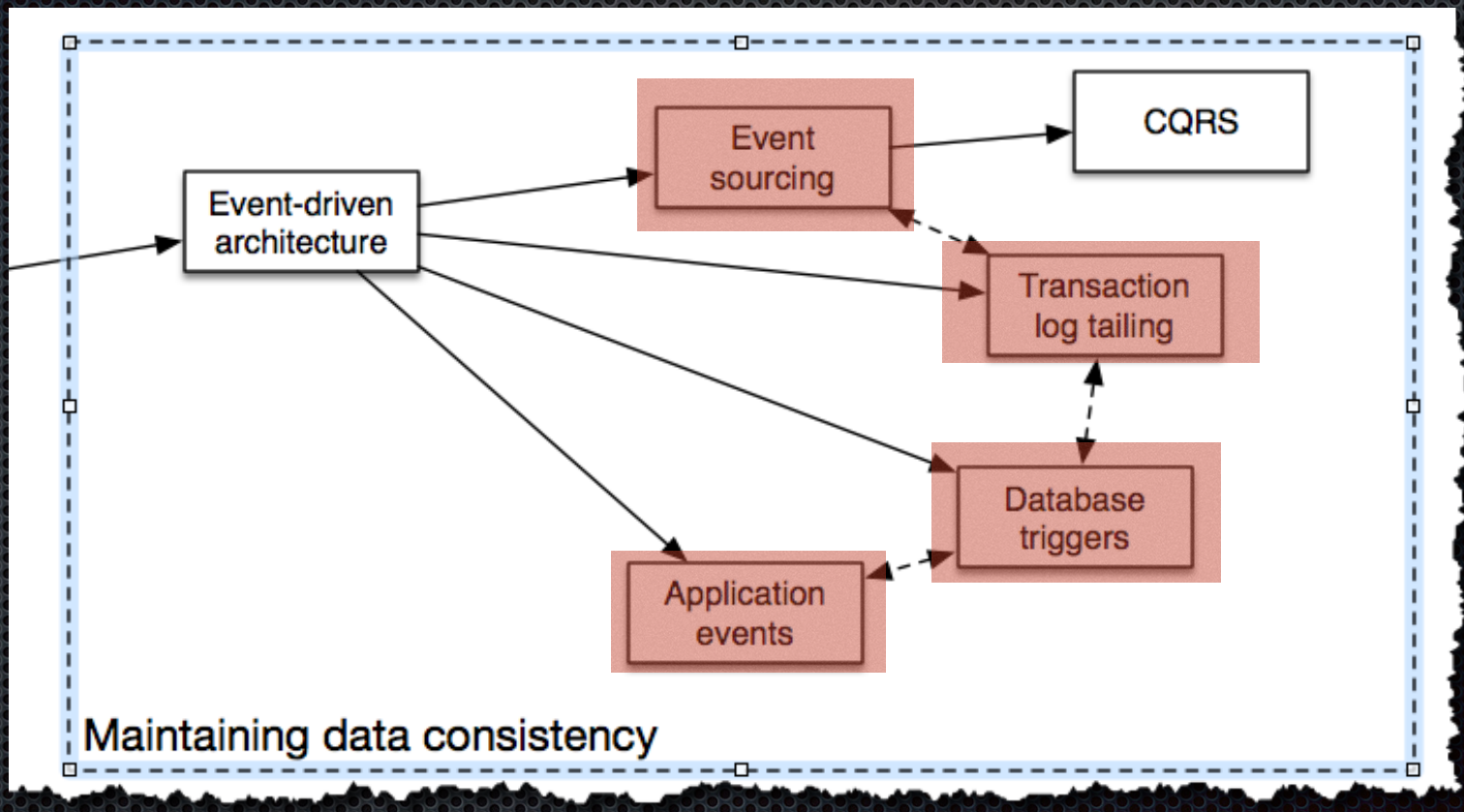
dual write problem

Failure = inconsistent system



2PC is not an option

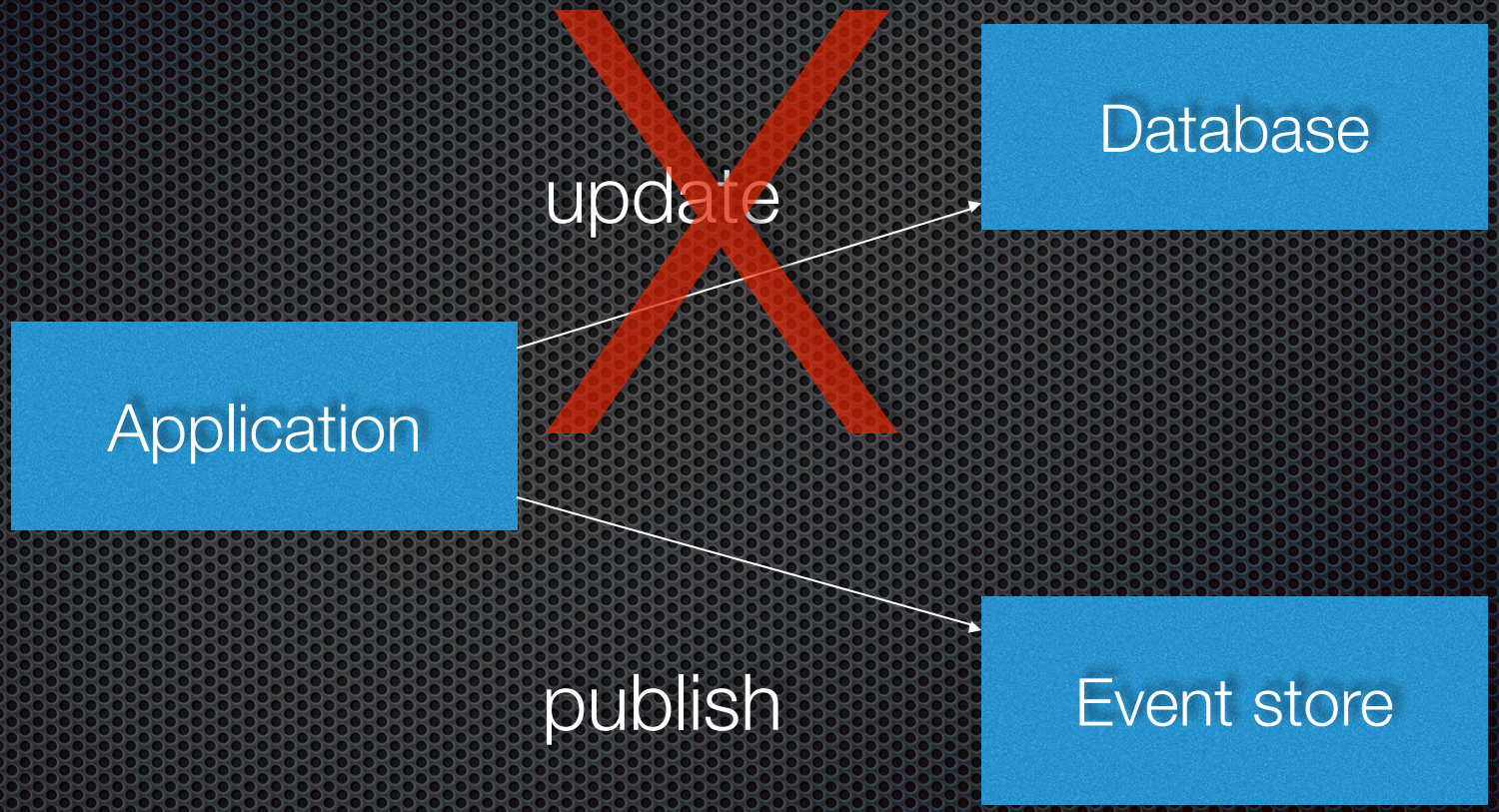
Reliably publish events when state changes



Agenda

- The problem with Domain Models and microservices
- Overview of aggregates
- Maintaining consistency between aggregates
- Using event sourcing with Aggregates
- Example application

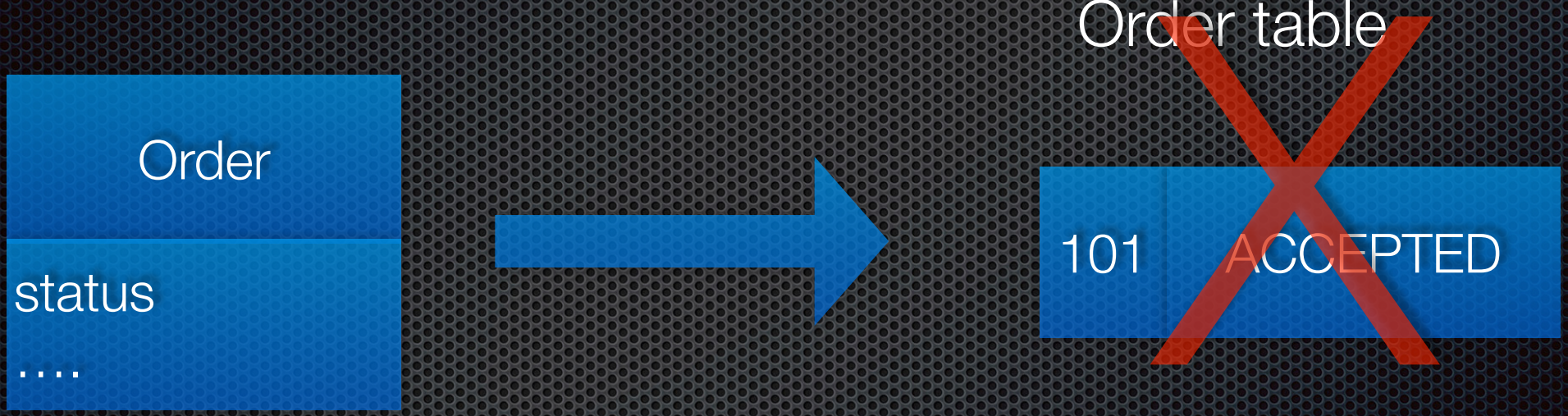
Event sourcing = event-centric persistence



Event sourcing

- For each domain object (i.e. DDD aggregate):
  - Identify (state changing) domain events, e.g. use Event Storming
  - Define Event classes
- For example, Order events: OrderCreated, OrderCancelled, OrderApproved, OrderRejected, OrderShipped

Persists events
NOT current state

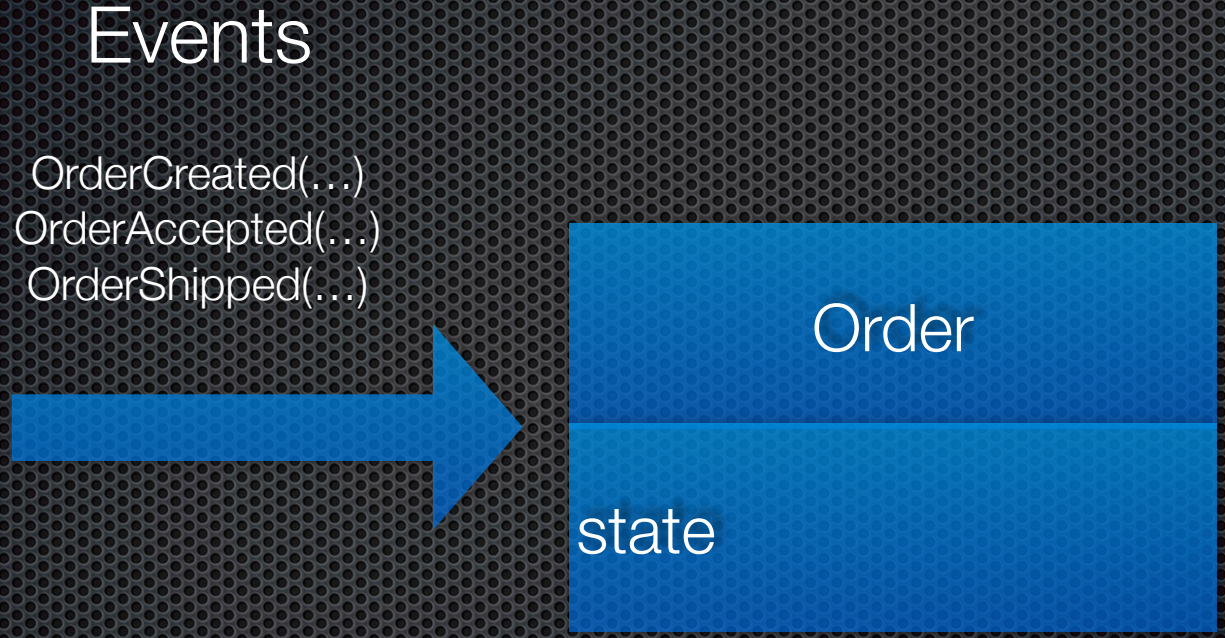


Persists events NOT current state

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 state

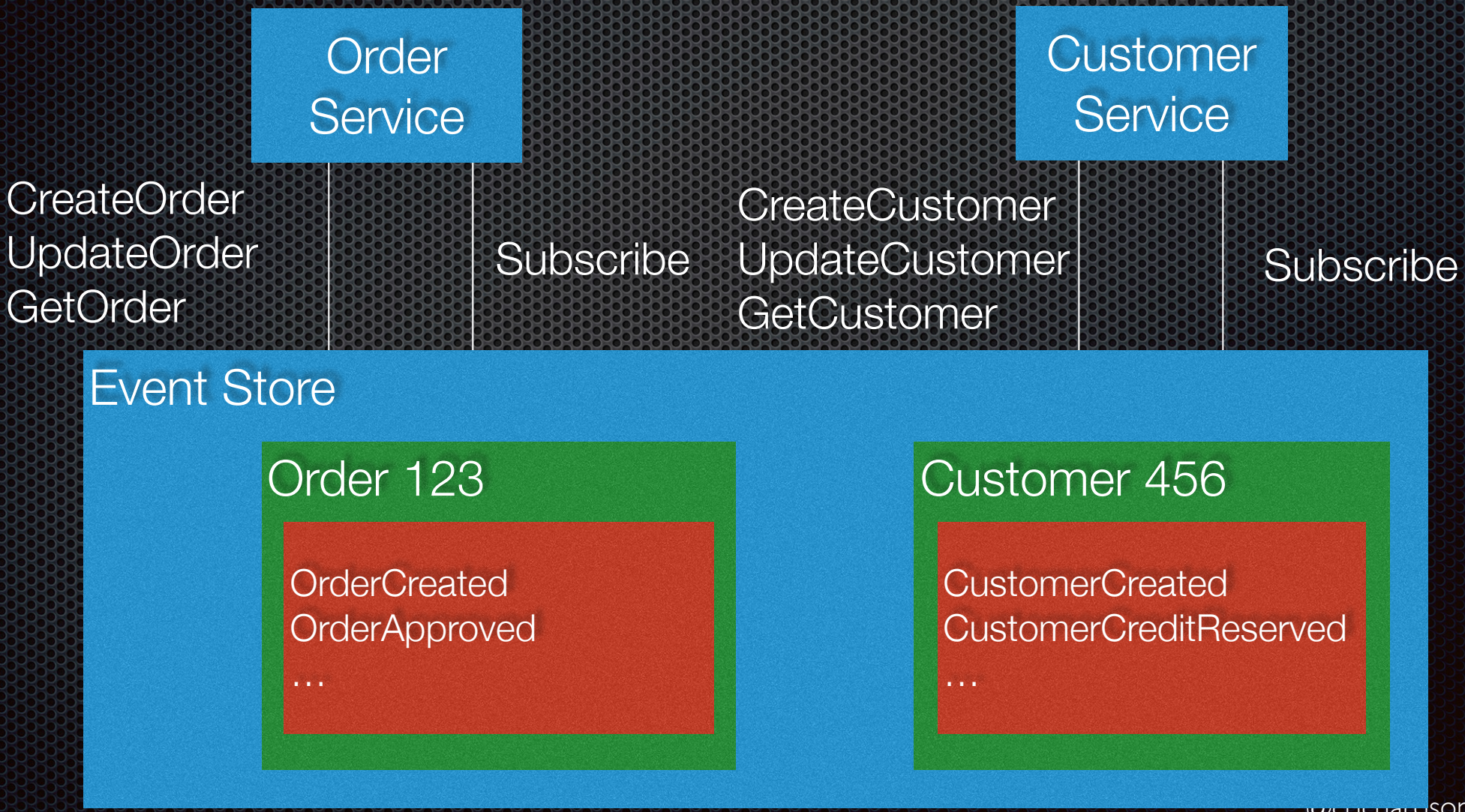


Periodically snapshot to avoid loading all events

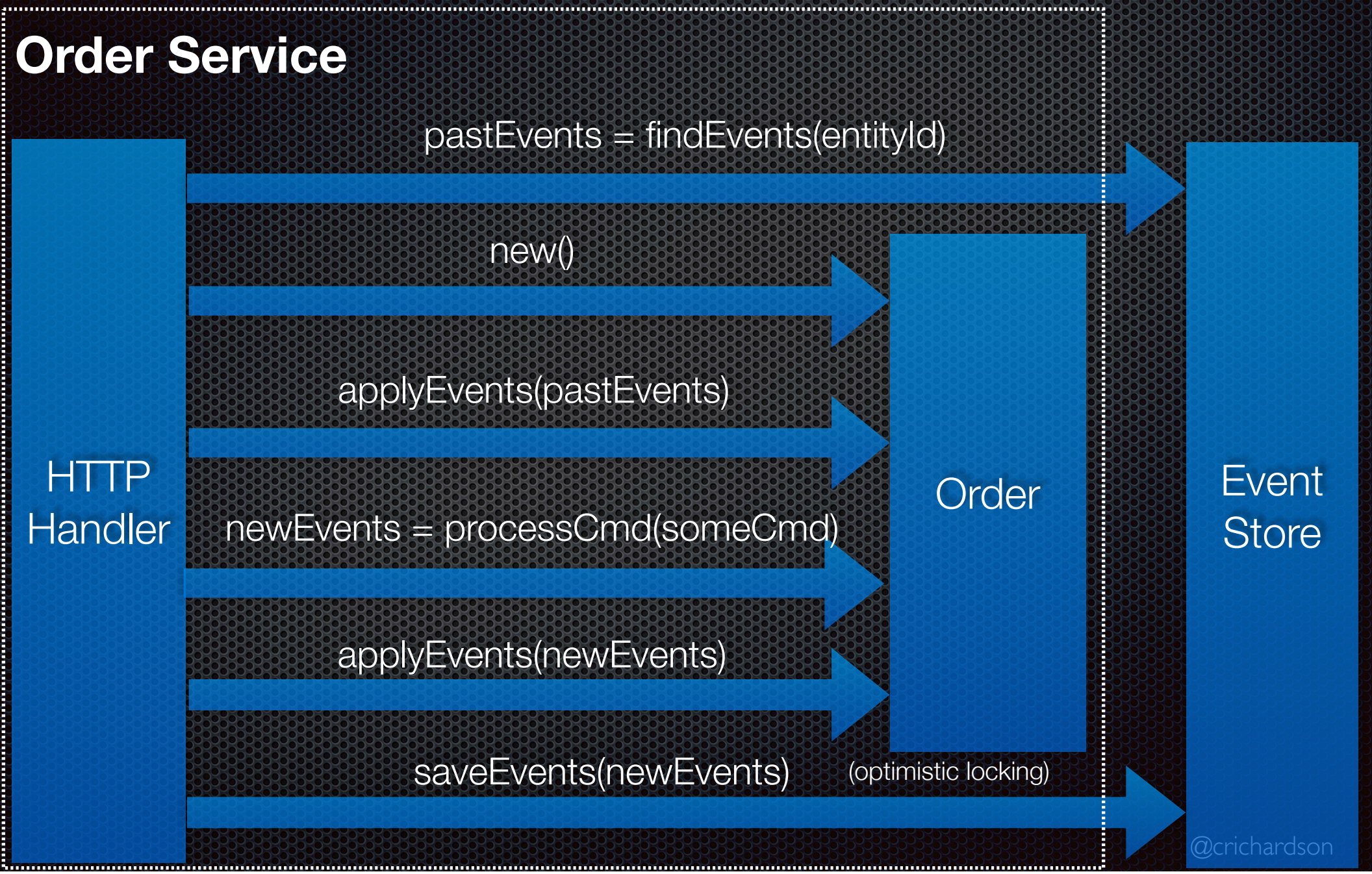
The present is a fold over history

currentState = foldl(applyEvent, initialState, events)

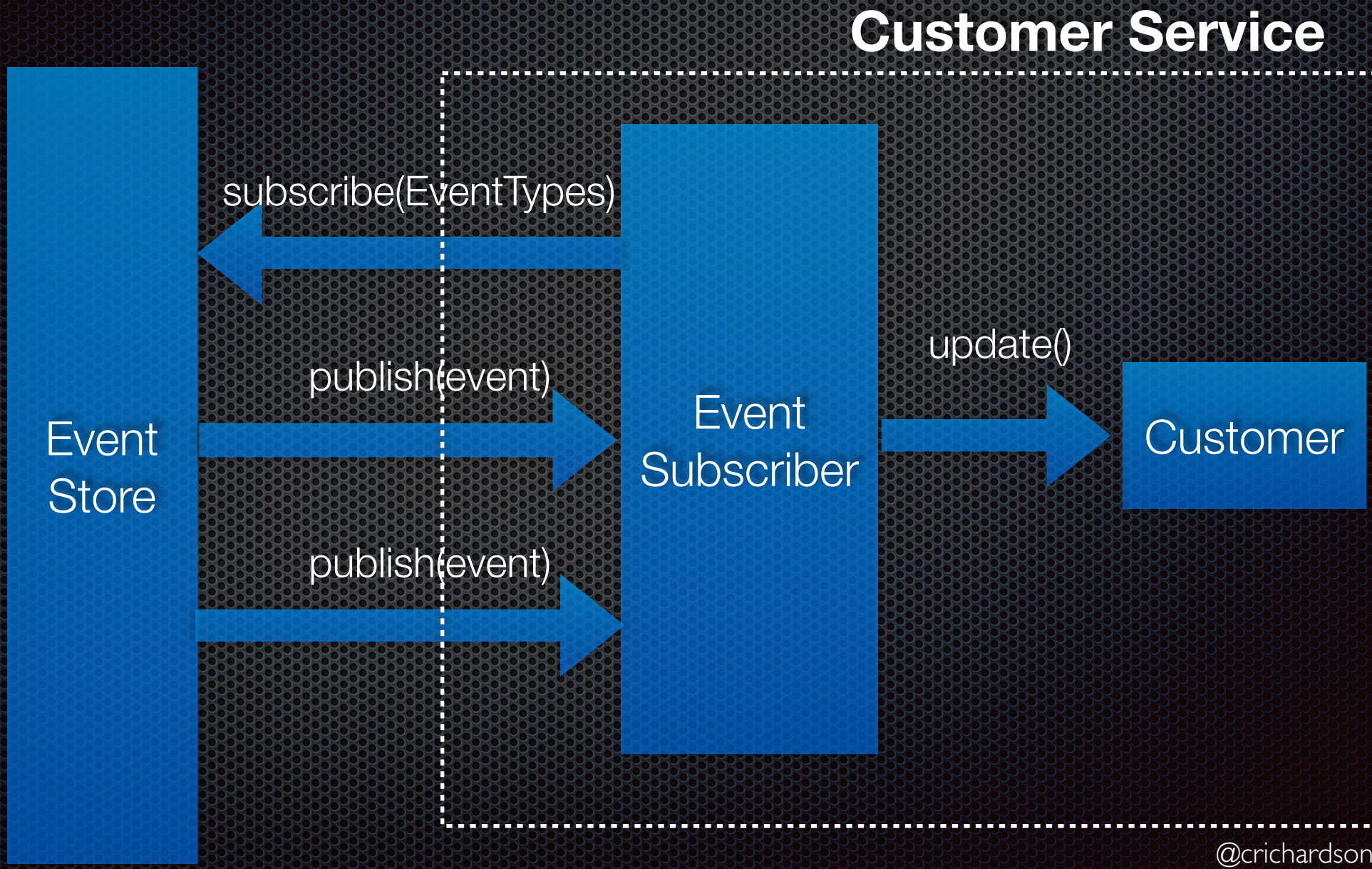
Application architecture



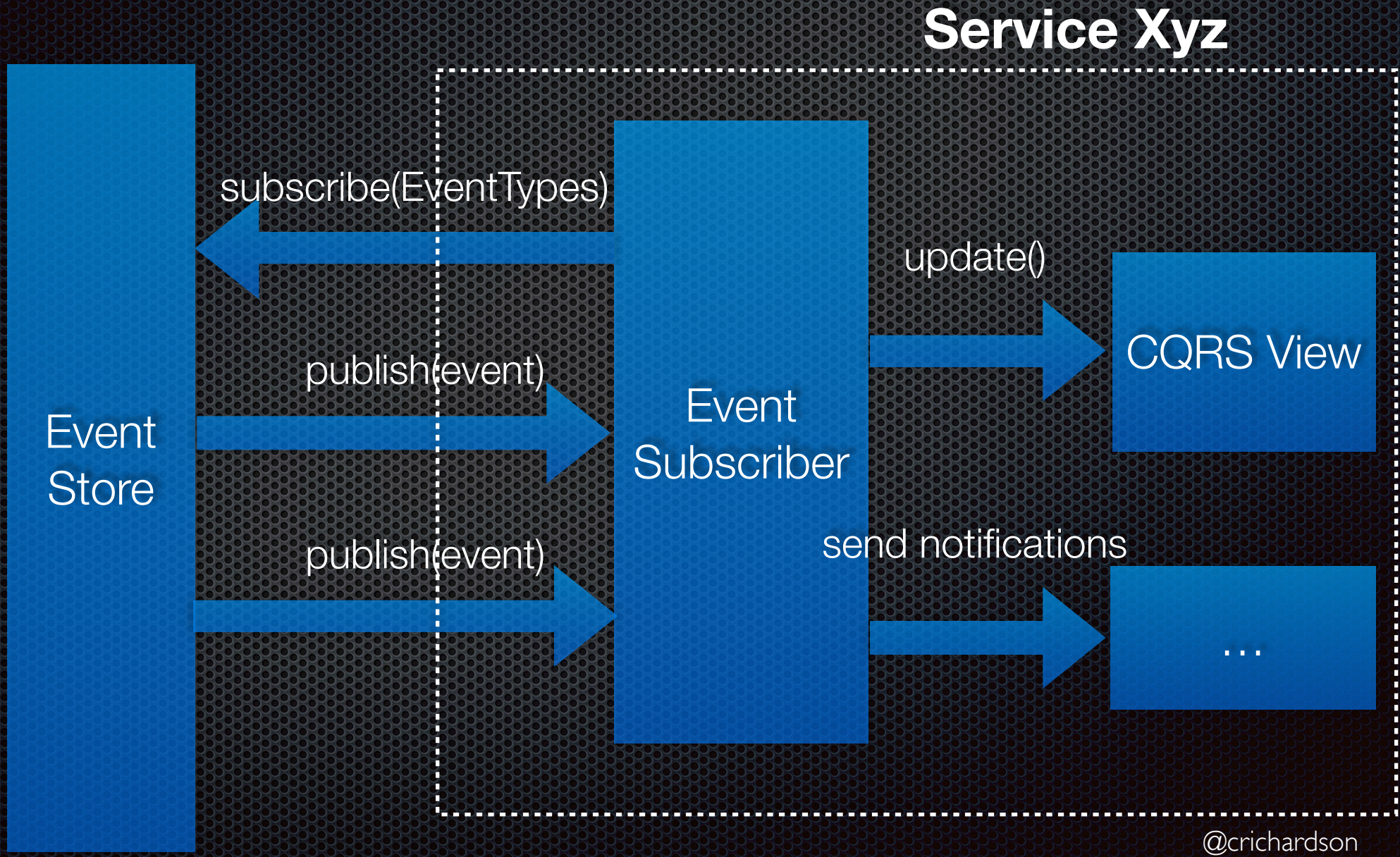
Request handling in an event sourced application



Event Store publishes events consumed by other services



Event Store publishes events consumed by other services



Event store = database + message broker

- Hybrid database and message broker
- Implementations:
  - Home grown/DIY
  - geteventstore.com by Greg Young
  - http://eventuate.io (mine)

Benefits of event sourcing

- Solves data consistency issues in a Microservice/NoSQL based architecture
- Reliable event publishing: publishes events needed by predictive analytics etc, user notifications,…
- Eliminates O/R mapping problem (mostly)
- Reifies state changes:
  - Built in, reliable audit log
  - temporal queries
- Preserved history ⇒ More easily implement future requirements

Drawbacks of event sourcing…

- Requires application rewrite
- Weird and unfamiliar style of programming
- Events = a historical record of your bad design decisions
- Must detect and ignore duplicate events
  - Idempotent event handlers
  - Track most recent event and ignore older ones
  - …

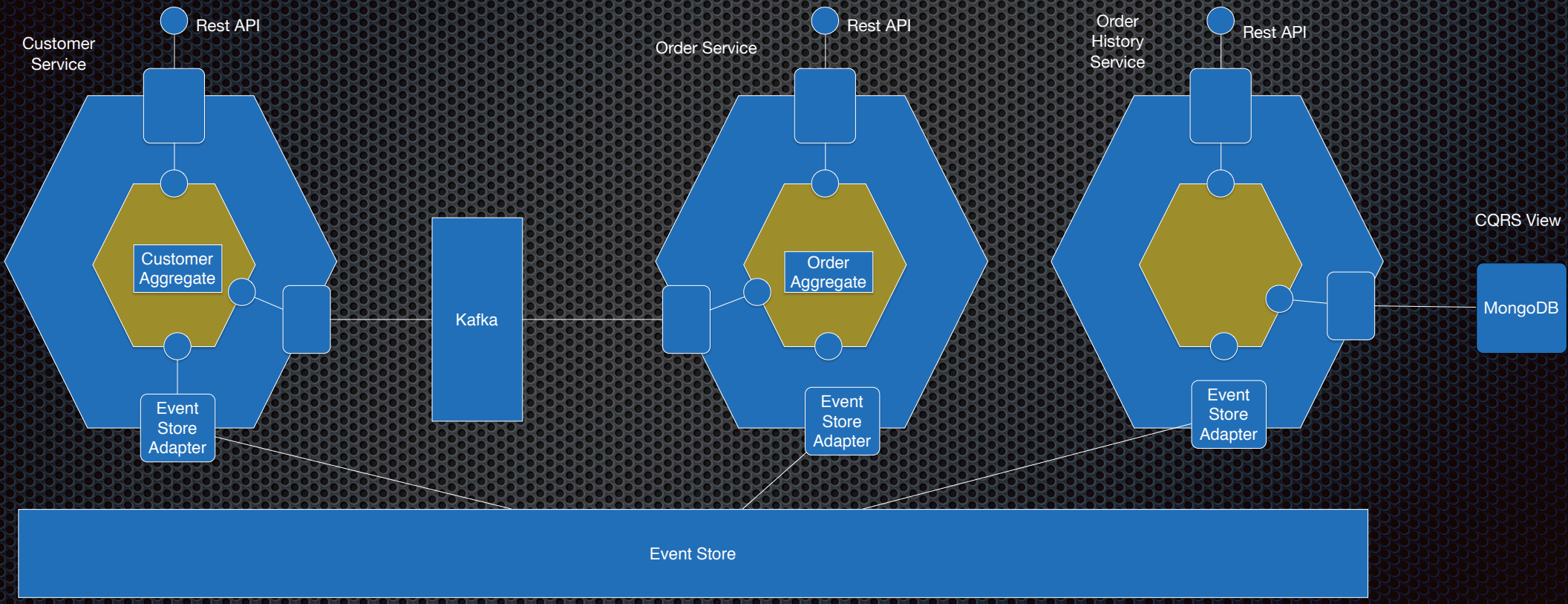
… Drawbacks of event sourcing

- Querying the event store can be challenging
- Some queries might be complex/inefficient, e.g. accounts with a balance > X
- Event store might only support lookup of events by entity id
- Must use Command Query Responsibility Segregation (CQRS) to handle queries ⇒ application must handle eventually consistent data

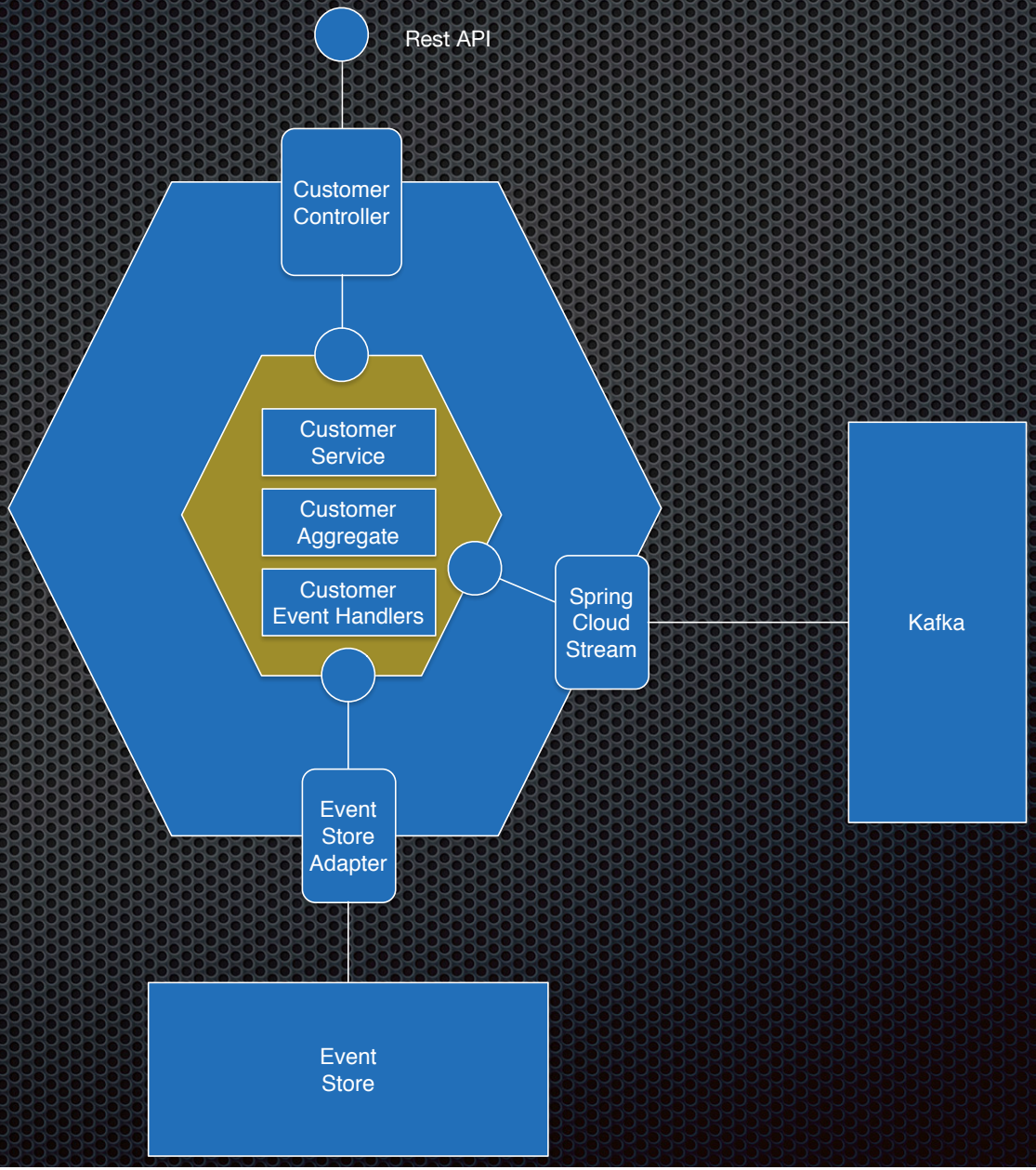
Agenda

- The problem with Domain Models and microservices
- Overview of aggregates
- Maintaining consistency between aggregates
- Using event sourcing with Aggregates
- Example application

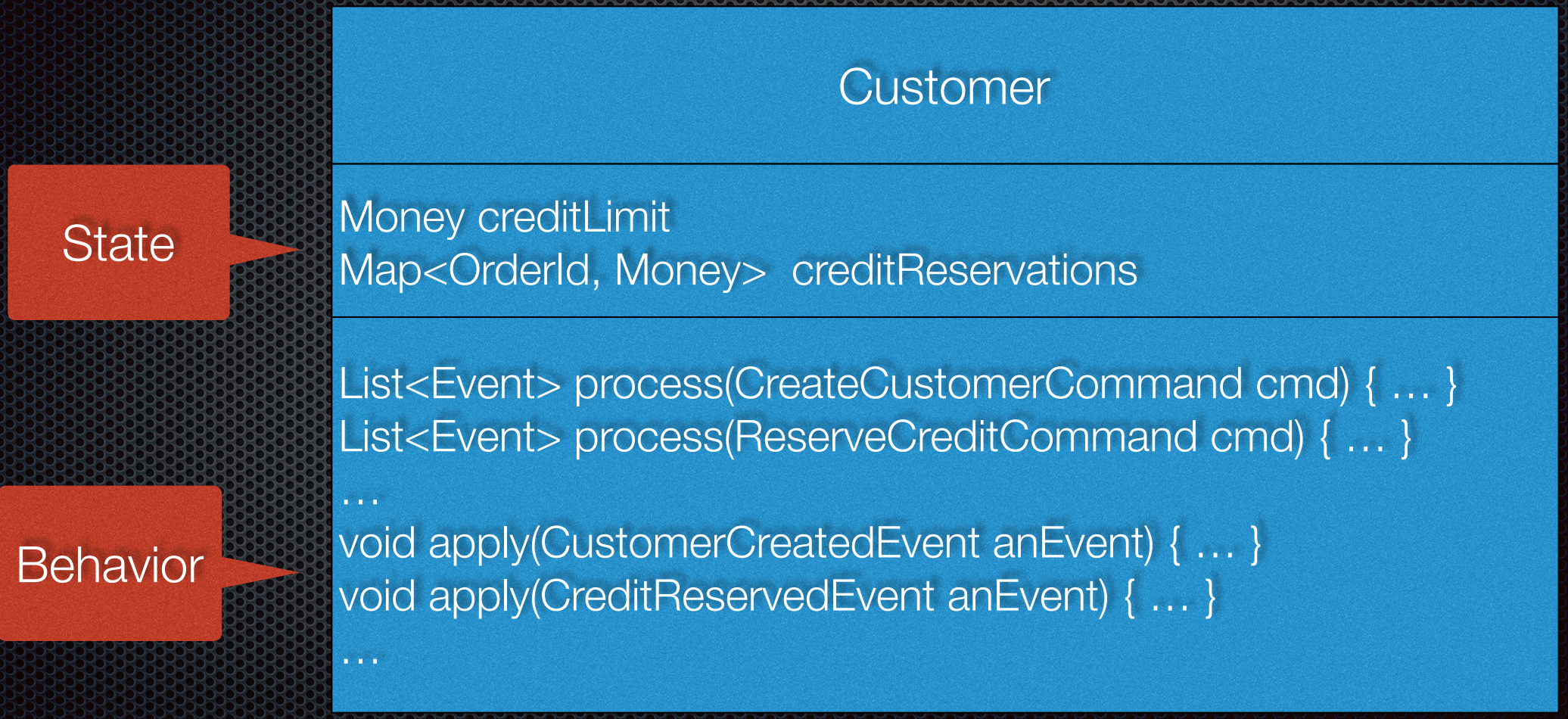
Orders and Customers example



Customer microservice



The Customer aggregate



Familiar concepts restructured

```
class Customer {

  public void reserveCredit(
       orderId : String,
       amount : Money) {

    // verify

    // update state
    this.xyz = …
}
```

```
public List<Event> process(
       ReserveCreditCommand cmd) {

   // verify
    …
   return singletonList(
     new CreditReservedEvent(…);
    )
  }
```

```
 public void apply(
      CreditReservedEvent event) {

     // update state
     this.xyz = event.xyz
}
```

Customer command processing

```
public class Customer extends ReflectiveMutableCommandProcessingAggregate<Customer, CustomerCommand> {

  private Money creditLimit;
  private Map<EntityIdentifier, Money> creditReservations;

  Money availableCredit() {
    return creditLimit.subtract(creditReservations.values().stream().reduce(Money.ZERO, Money::add));
  }

  Money getCreditLimit() { return creditLimit; }

  public List<Event> process(CreateCustomerCommand cmd) {
    return EventUtil.events(new CustomerCreatedEvent(cmd.getName(), cmd.getCreditLimit()));
  }

  public List<Event> process(ReserveCreditCommand cmd) {
    if (availableCredit().isGreaterThanOrEqual(cmd.getOrderTotal()))
      return EventUtil.events(new CustomerCreditReservedEvent(cmd.getOrderId(), cmd.getOrderTotal()));
    else
      return EventUtil.events(new CustomerCreditLimitedExceededEvent(cmd.getOrderId()));
  }

  public void apply(CustomerCreatedEvent event) {...}

  public void apply(CustomerCreditReservedEvent event) {...}

  public void apply(CustomerCreditLimitedExceededEvent event) {...}

}
```

Customer applying events

```
public class Customer extends ReflectiveMutableCommandProcessingAggregate<Customer, CustomerCommand> {

  private Money creditLimit;
  private Map<EntityIdentifier, Money> creditReservations;

  Money availableCredit() {...}

  Money getCreditLimit() { return creditLimit; }

  public List<Event> process(CreateCustomerCommand cmd) {...}

  public List<Event> process(ReserveCreditCommand cmd) {...}


  public void apply(CustomerCreatedEvent event) {
    this.creditLimit = event.getCreditLimit();
    this.creditReservations = new HashMap<>();
  }

  public void apply(CustomerCreditReservedEvent event) {
    this.creditReservations.put(event.getOrderId(), event.getOrderTotal());
  }

  public void apply(CustomerCreditLimitedExceededEvent event) {
    // Do nothing
  }

}
```

Customer Controller

```
@RestController
public class CustomerController {

    private CustomerService customerService;

    @Autowired
    public CustomerController(CustomerService customerService) {
        this.customerService = customerService;
    }

    @RequestMapping(value = "/customers", method = RequestMethod.POST)
    public CompletableFuture<CreateCustomerResponse>
                        createCustomer(@RequestBody CreateCustomerRequest createCustomerRequest) {
        return customerService.createCustomer(createCustomerRequest.getName(),
                                               createCustomerRequest.getCreditLimit())
                .thenApply(ewidv -> new CreateCustomerResponse(ewidv.getEntityId()));
    }
}
```

Creating a Customer

```
public class CustomerServiceImpl implements CustomerService {

  private final AggregateRepository<Customer, CustomerCommand> customerRepository;

  public CustomerServiceImpl(AggregateRepository<Customer, CustomerCommand> customerRepository) {
    this.customerRepository = customerRepository;
  }

  @Override
  public CompletableFuture<EntityWithIdAndVersion<Customer>>
                      createCustomer(String name, Money creditLimit) {
    return customerRepository.save(new CreateCustomerCommand(name, creditLimit));
  }
}
```

save() concisely specifies:
1. Creates Customer aggregate
2. Processes command
3. Applies events
4. Persists events

Event handling in Customers

Triggers BeanPostProcessor

Durable subscription name

```
@EventSubscriber(id = "customerWorkflow")
public class CustomerWorkflow {

  private Logger logger = LoggerFactory.getLogger(getClass());

  private Source source;

  public CustomerWorkflow(Source source) { this.source = source; }

  @EventHandlerMethod
  public CompletableFuture<?> reserveCredit(EventHandlerContext<OrderCreatedEvent> ctx) {
    OrderCreatedEvent event = ctx.getEvent();
    Money orderTotal = event.getOrderTotal();
    String customerId = event.getCustomerId();
    String orderId = ctx.getEntityId();

    return ctx.update(Customer.class, customerId, new ReserveCreditCommand(orderTotal, orderId))
            .handleAsync(
                    recoverWith(singletonMap(EntityNotFoundException.class, (t) -> {
                      rejectOrder(orderId);
                      return null;
                    })));
  }
```

1. Load Customer aggregate
2. Processes command
3. Applies events
4. Persists events

Publish message to Spring Cloud Stream (Kafka) when Customer not found

Summary

- Aggregates are the building blocks of microservices
- Use events to maintain consistency between aggregates
- Event sourcing is a good way to implement a event-driven architecture


@crichardson chris@chrisrichardson.net
Questions?
http://learnmicroservices.io