

Developing event-driven microservices with event sourcing and CQRS

Chris Richardson

Author of POJOs in Action

Founder of the original CloudFoundry.com

Twitter: @crichtardson

chris@chrisrichardson.net

<http://plainoldobjects.com>

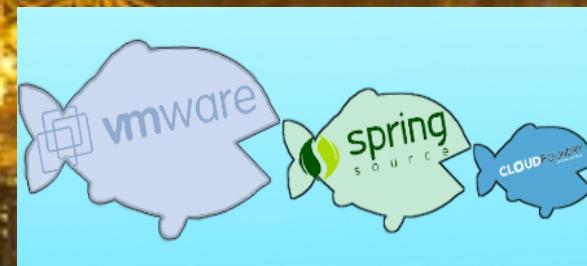
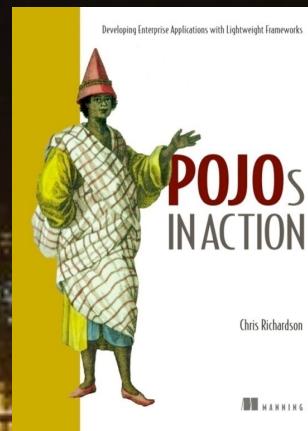
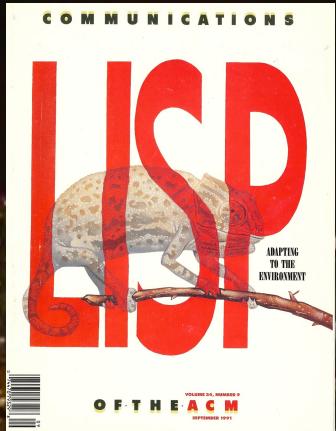
<http://microservices.io>



Presentation goal

Show how Event Sourcing and
Command Query Responsibility Segregation
(CQRS)
are a great way to implement microservices

About Chris



About Chris

Consultant and trainer focusing
on microservices
(<http://www.chrisrichardson.net/>)

About Chris

Founder of a startup that is creating
a platform that makes it easy for
application developers write
microservices
(<http://bit.ly/trialeventuate>)

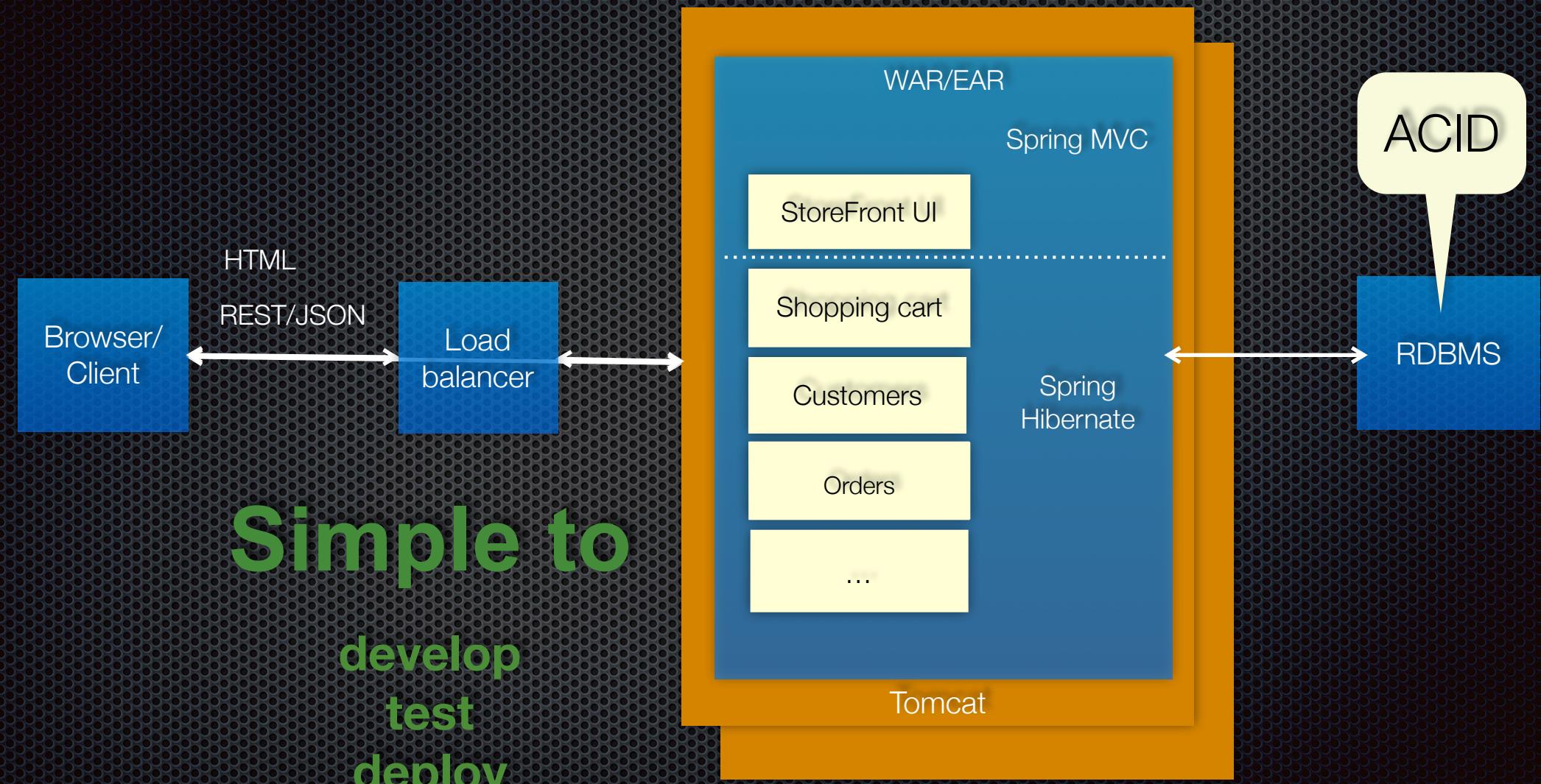
For more information

- <https://github.com/cer/event-sourcing-examples>
- <http://microservices.io>
- <http://plainoldobjects.com/>
- <https://twitter.com/crichardson>
- <http://eventuate.io/>

Agenda

- Why event sourcing?
- Overview of event sourcing
- ACID-free design
- Designing a domain model based on event sourcing
- Implementing queries in an event sourced application
- Event sourcing and microservices

Traditional monolithic architecture



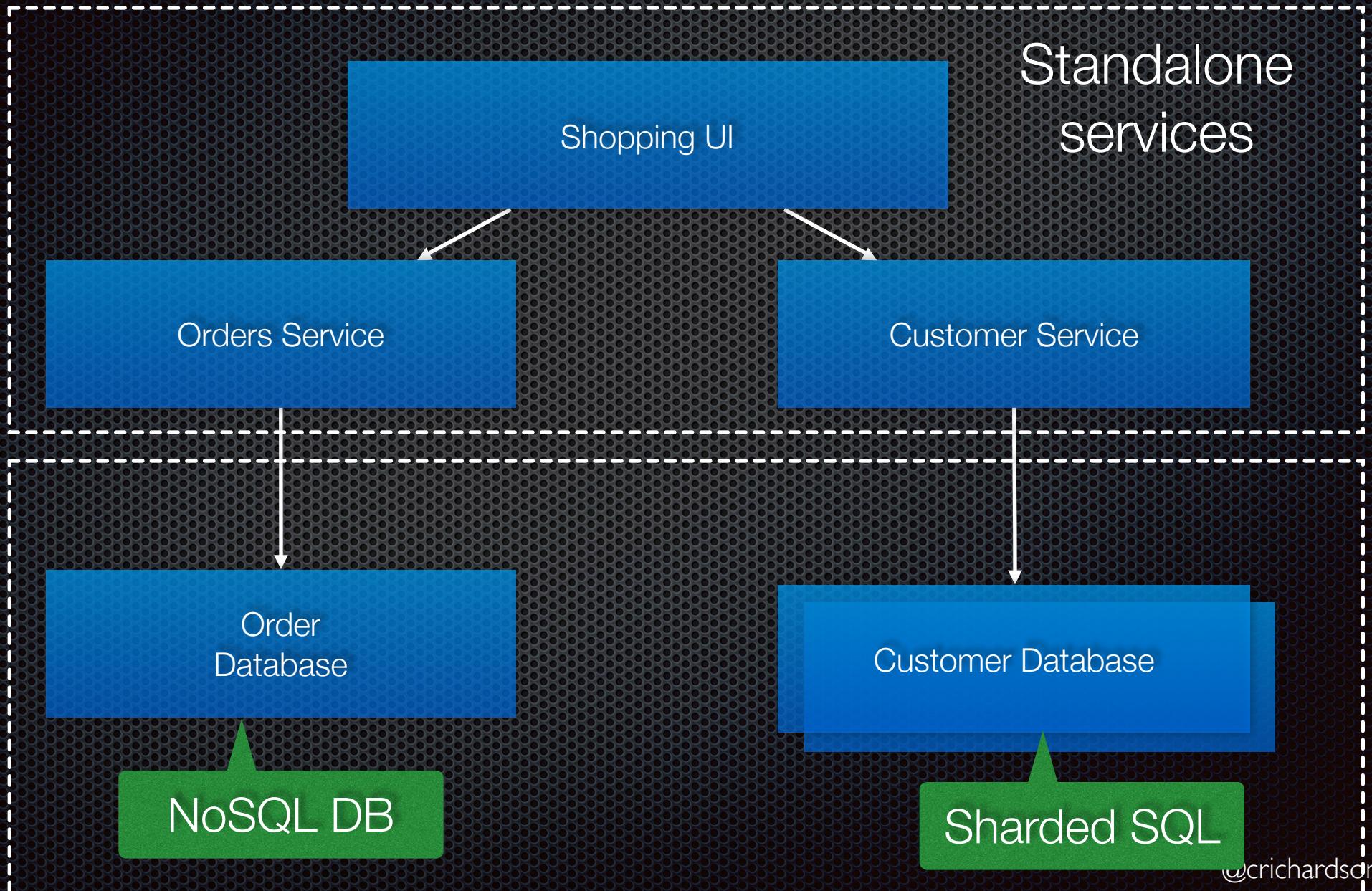
Simple to
develop
test
deploy
scale

A photograph of the Great Pyramid of Giza. The pyramid's massive, polished stone blocks are arranged in a series of terraced layers that rise towards a flat top. In the foreground, several people are standing on the pyramid's base, which emphasizes the enormous size of the structure. The stone has a warm, yellowish-brown hue.

But that leads* to
monolithic hell

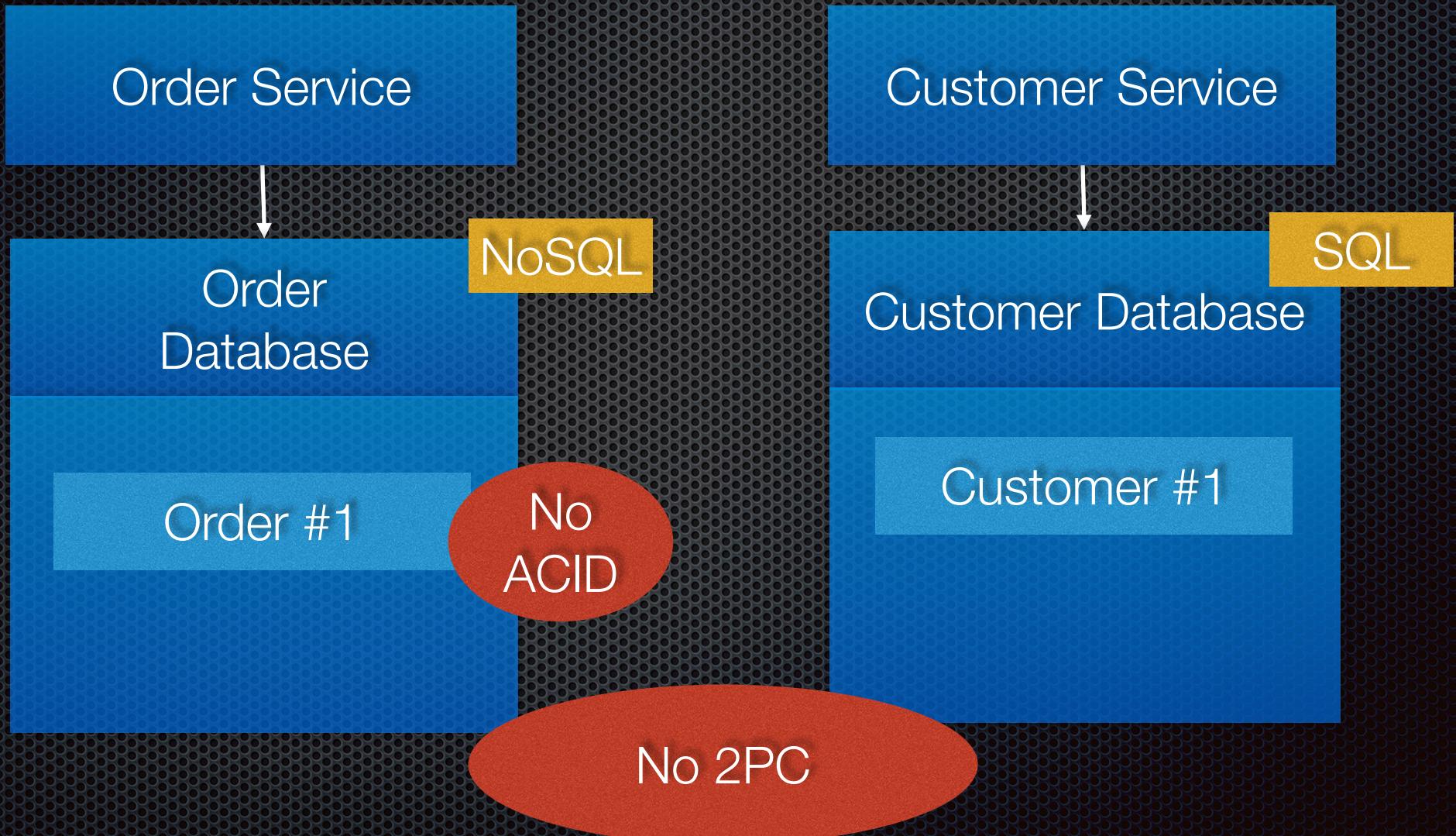
For large and/or complex applications...

Today: use a microservice, polyglot architecture

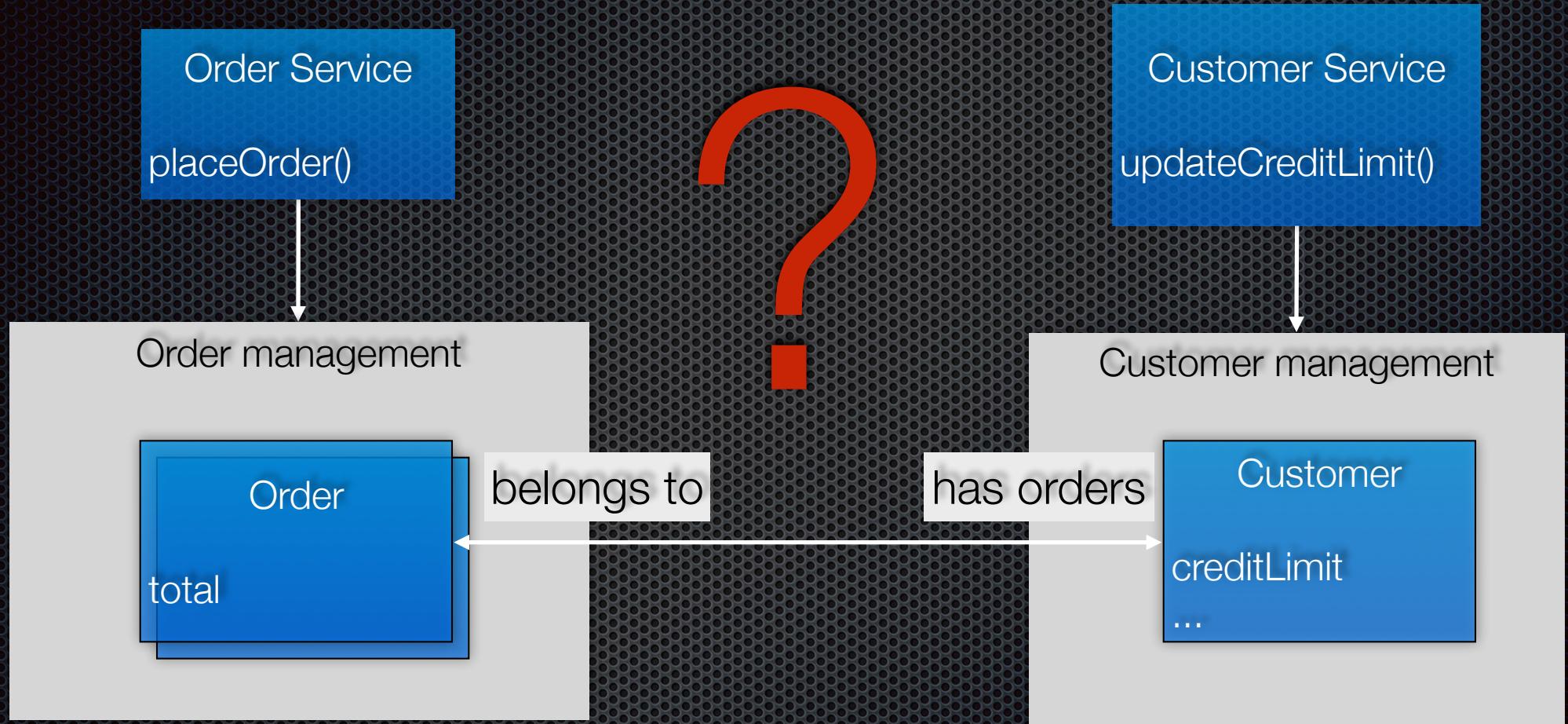


But now we have
distributed data management
problems

Example: placing an order

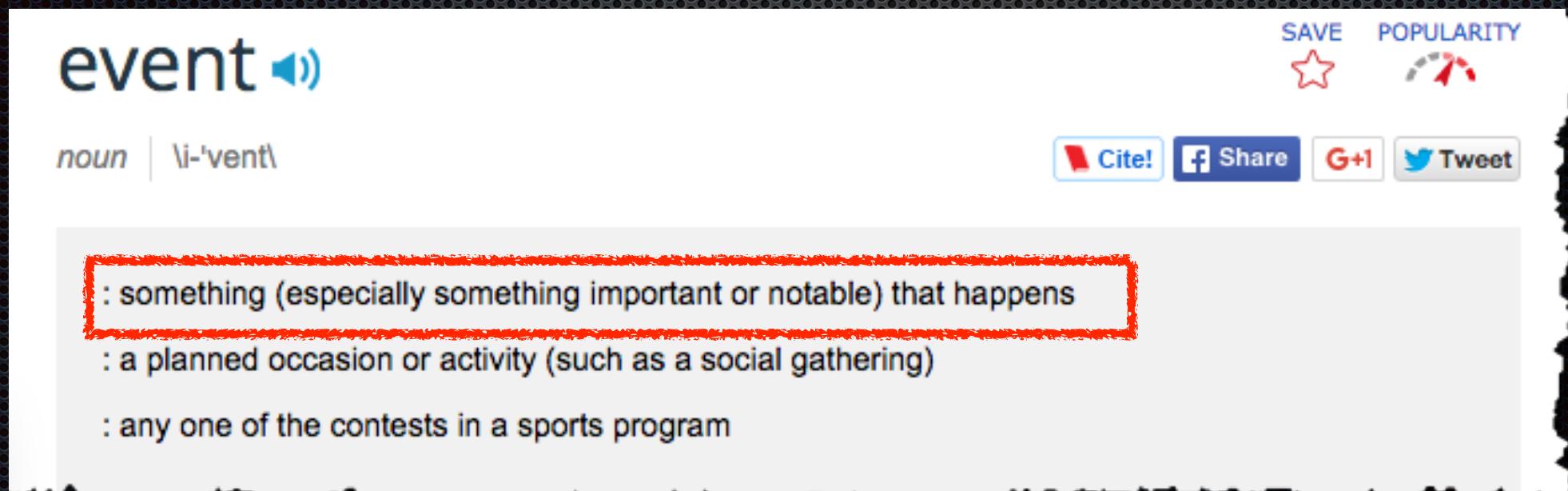


How to maintain invariants?



Invariant:
 $\text{sum(open order.total)} \leq \text{customer.creditLimit}$

Use an event-driven architecture....



The image shows a screenshot of the Merriam-Webster dictionary website. The word 'event' is highlighted in blue. Below it, the definition is listed in a box with a red border. The first definition is highlighted with a red box.

event 

noun | *\i-'vent*

: something (especially something important or notable) that happens

: a planned occasion or activity (such as a social gathering)

: any one of the contests in a sports program

SAVE  POPULARITY 

 Cite!  Share  G+1  Tweet

<http://www.merriam-webster.com/dictionary/event>

....Use an event-driven architecture

- Services **publish** events when something important happens, e.g. state changes
- Services **subscribe** to events and update their state
 - Maintain **eventual consistency** across multiple aggregates (in multiple datastores)
 - Synchronize replicated data

Eventually consistent credit checking

createOrder()



Order Management

Order
id : 4567
total: 343
state = OPEN

Customer Management

Customer
creditLimit : 12000
creditReservations: { 4567 -> 343}

Subscribes to:
CreditReservedEvent

publishes:

OrderCreatedEvent

Subscribes to:
OrderCreatedEvent

Publishes:

CreditReservedEvent

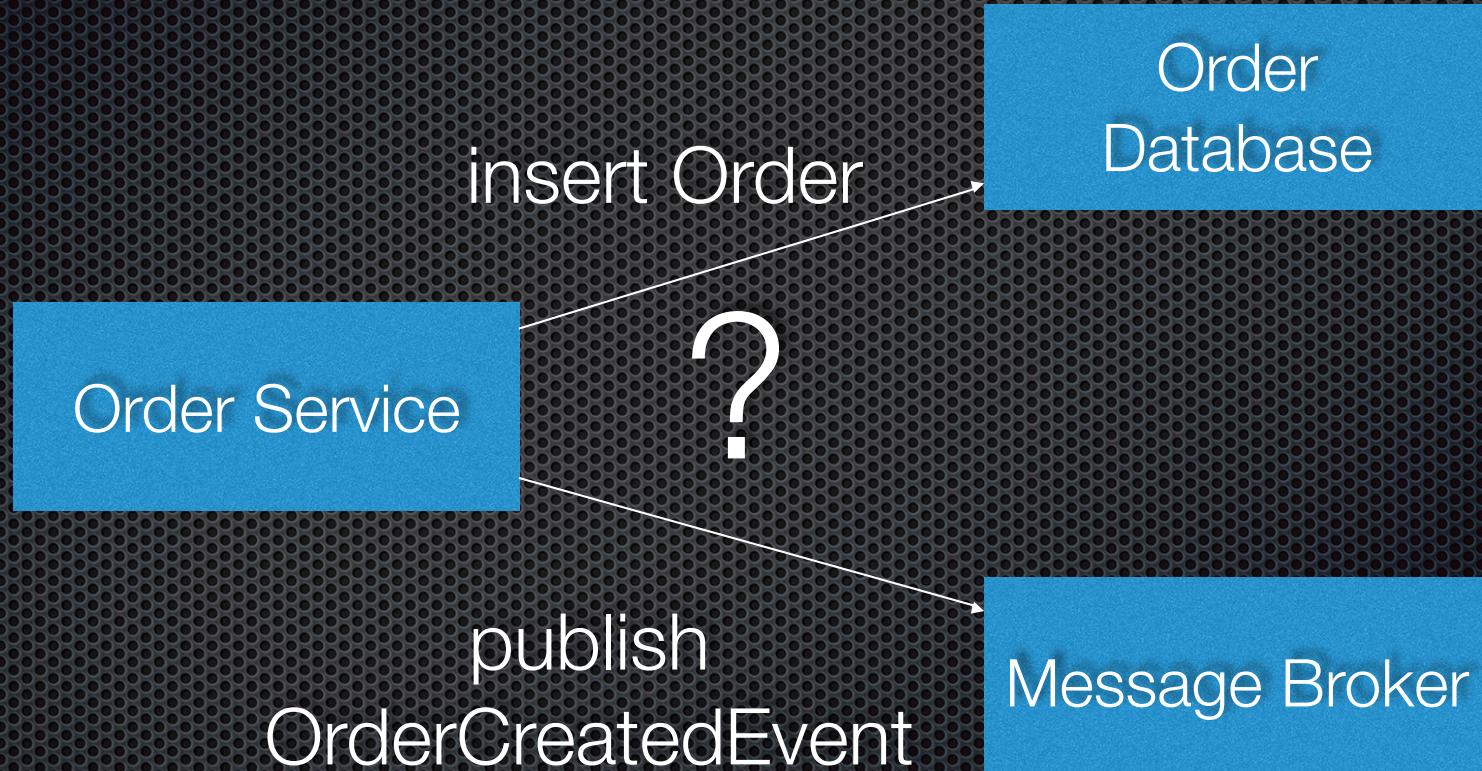
Message Bus

Now there are two problems
to solve....

Problem #1: How to design
eventually consistent business logic?

More on that later....

Problem #2: How atomicity update database and publish an event



dual write problem

Update and publish using 2PC

- Guaranteed atomicity **BUT**
- Need a distributed transaction manager
- Database and message broker must support 2PC
- Impacts reliability
- Not fashionable
- 2PC is best avoided

Transaction log tailing

- How:
 - Read the database “transaction log” = single source of truth
 - Publish events to message broker
- LinkedIn databus <https://github.com/linkedin/databus>
 - Supports Oracle and MySQL
 - Publish as events
- AWS DynamoDB streams
 - Ordered sequence of creates, updates, deletes made to a DynamoDB table
 - Last 24 hours
 - Subscribe to get changes
- MongoDB
 - Read the oplog

Transaction log tailing: benefits and drawbacks

- **Benefits**

- No 2PC
- No application changes required
- Guaranteed to be accurate

- **Drawbacks**

- Immature
- Database specific solutions
- Low-level DB changes rather business level events = need to reverse engineer domain events

Use database triggers

- Track changes to tables
- Insert events into an event table
 - Use datastore as a message queue
- Pull events from event table and write to message broker

Database triggers: benefits and drawbacks

- **Benefits**

- No 2PC
- No application changes required

- **Drawbacks**

- Requires the database to support them
- Database specific solutions
- Low-level DB changes rather business level events = need to reverse engineer domain events
- Error-prone, e.g. missing trigger

Application created events

- Use datastore as a message queue
 - Txn #1: Update database: new entity state & event
 - Txn #2: Consume event
 - Txn #3: Mark event as consumed
- Eventually consistent mechanism (used by eBay)
- See BASE: An Acid Alternative, <http://bit.ly/ebaybase>

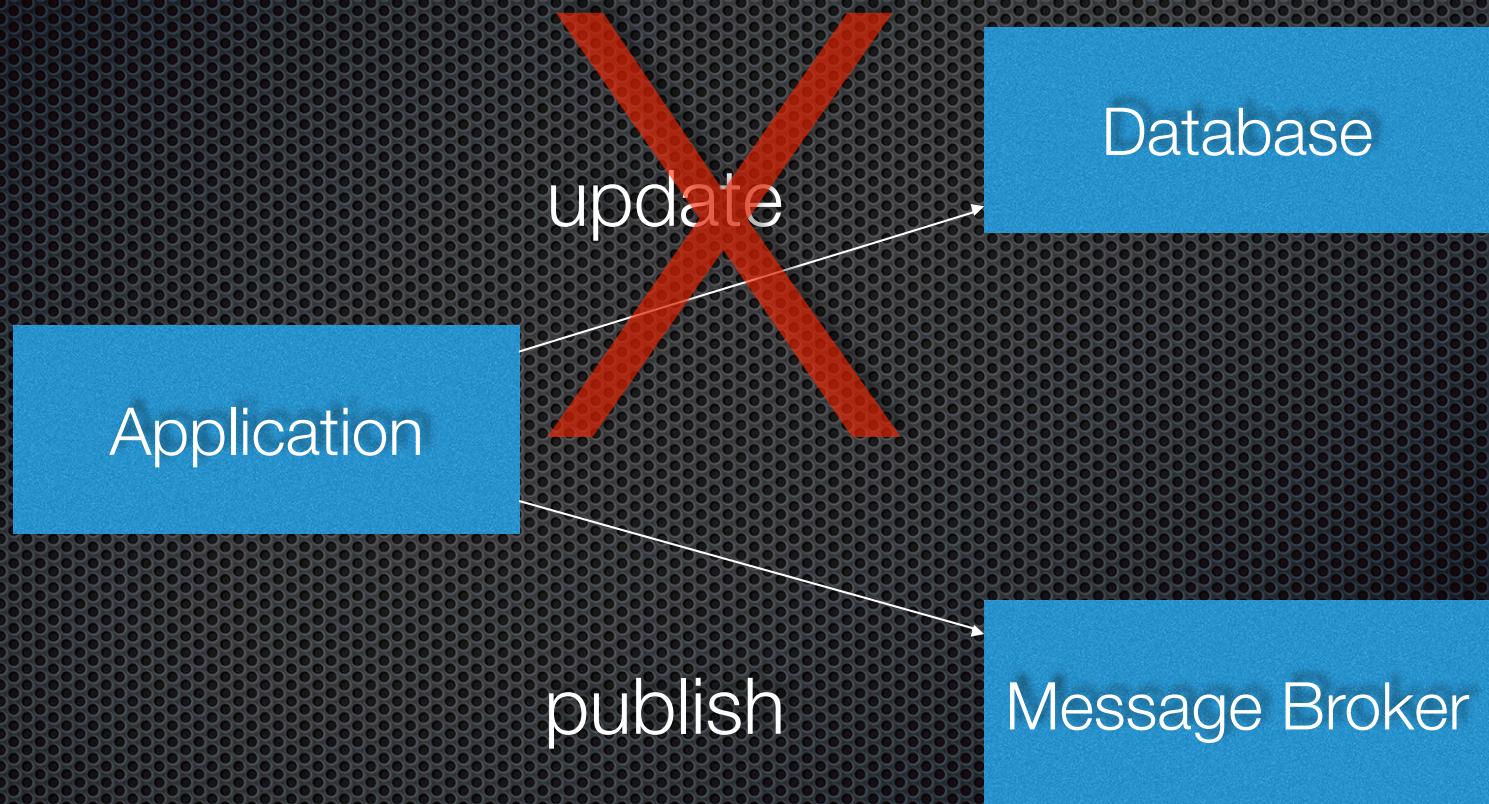
Application created events

- ✖ **Benefits**
 - ✖ High-level domain events
 - ✖ No 2PC
- ✖ **Drawbacks**
 - ✖ Requires changes to the application
 - ✖ Only works for SQL and **some** NoSQL databases
 - ✖ Error-prone

Agenda

- Why event sourcing?
- Overview of event sourcing
- ACID-free design
- Designing a domain model based on event sourcing
- Implementing queries in an event sourced application
- Event sourcing and microservices

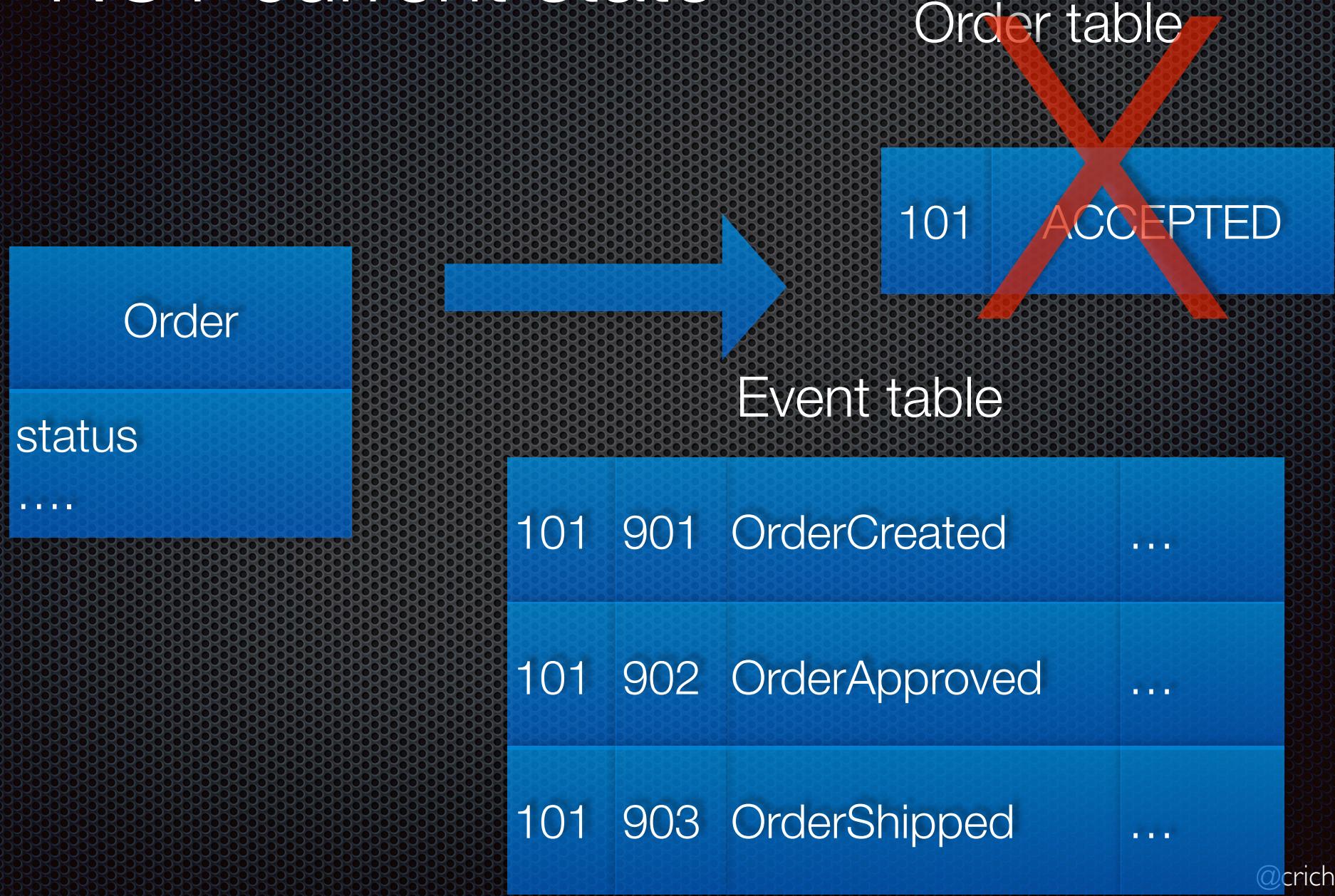
Just publish events



Event sourcing

- For each aggregate (business entity):
 - Identify (state-changing) domain events
 - Define Event classes
- For example,
 - ShoppingCart: ItemAddedEvent, ItemRemovedEvent, OrderPlacedEvent
 - Order: OrderCreated, OrderCancelled, OrderApproved, OrderRejected, OrderShipped

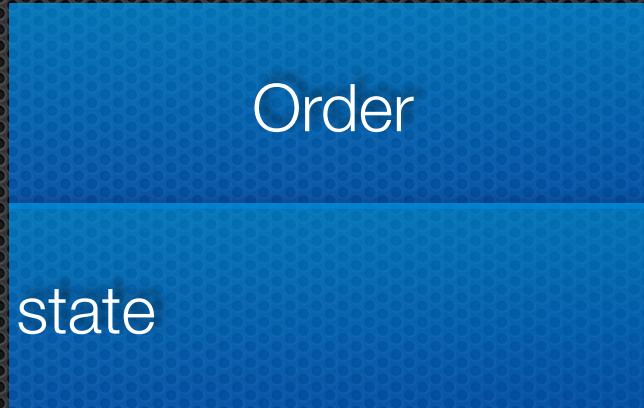
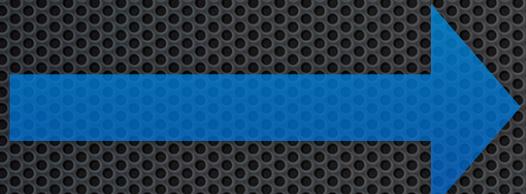
Persists events NOT current state



Replay events to recreate state

Events

OrderCreated(...)
OrderAccepted(...)
OrderShipped(...)

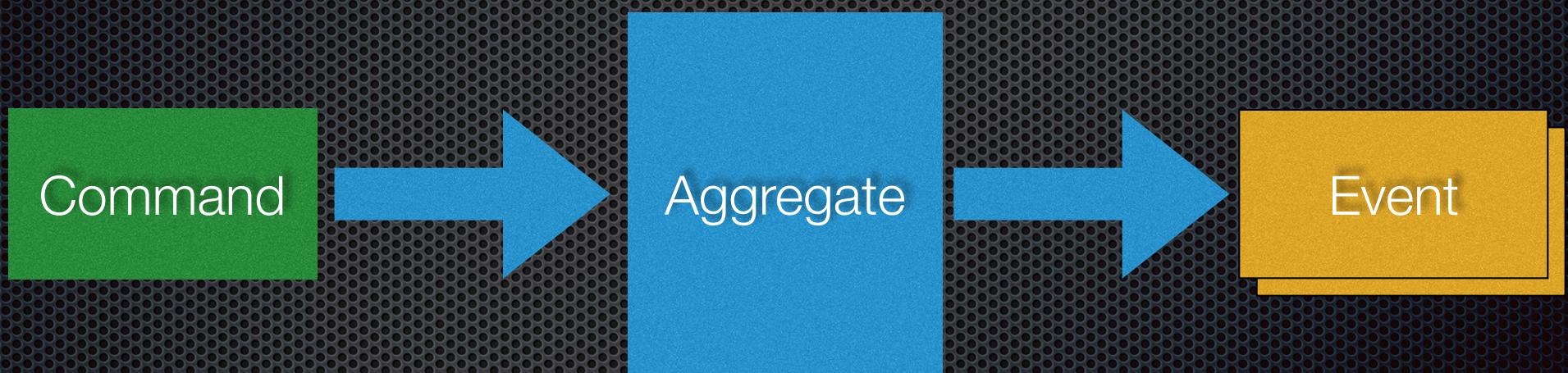


Periodically snapshot to avoid loading all events

The present is a fold over
history

currentState = foldl(applyEvent, initialState, events)

Aggregates: Command ⇒ Events

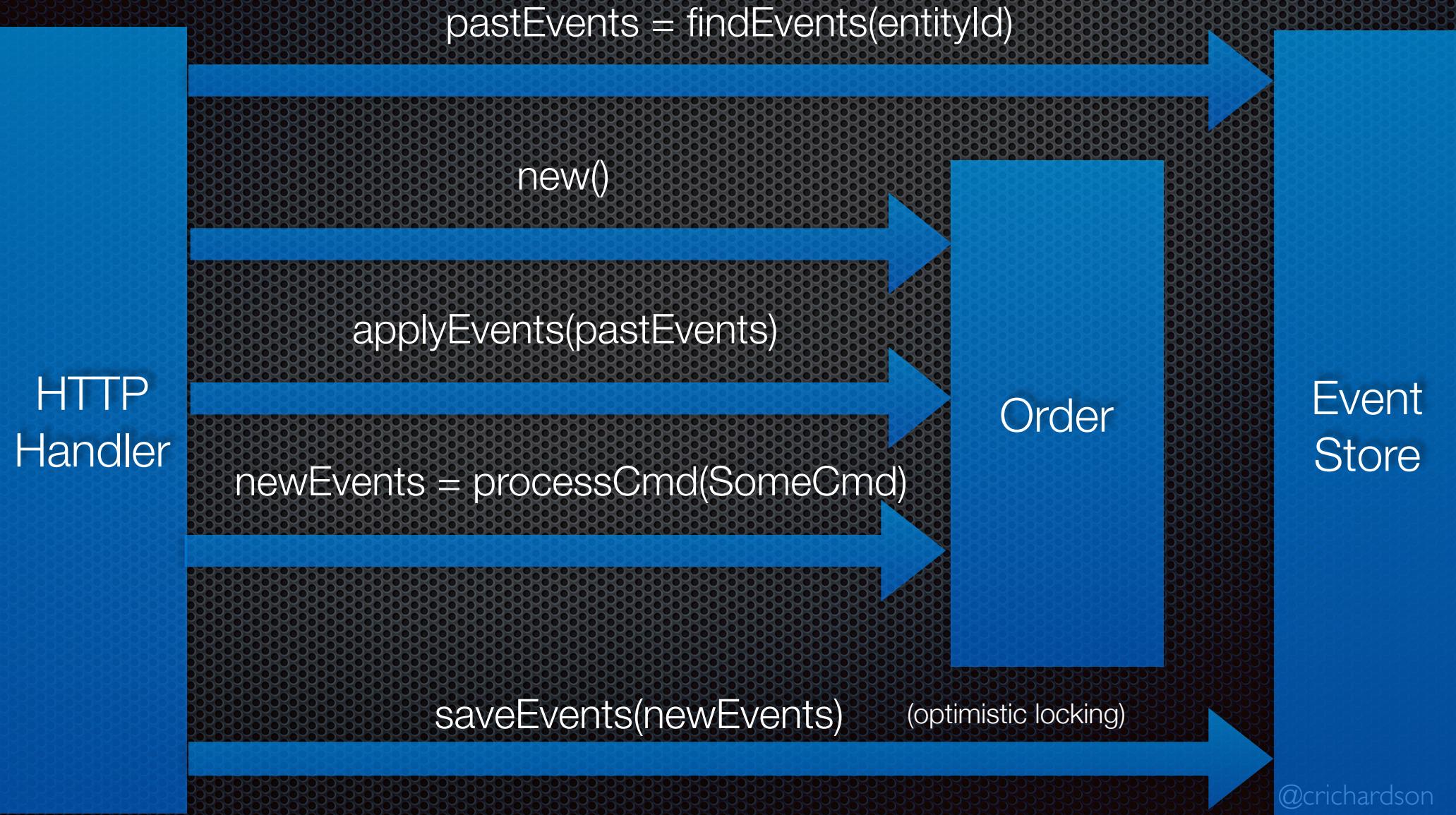


Aggregates: Event \Rightarrow Updated aggregate

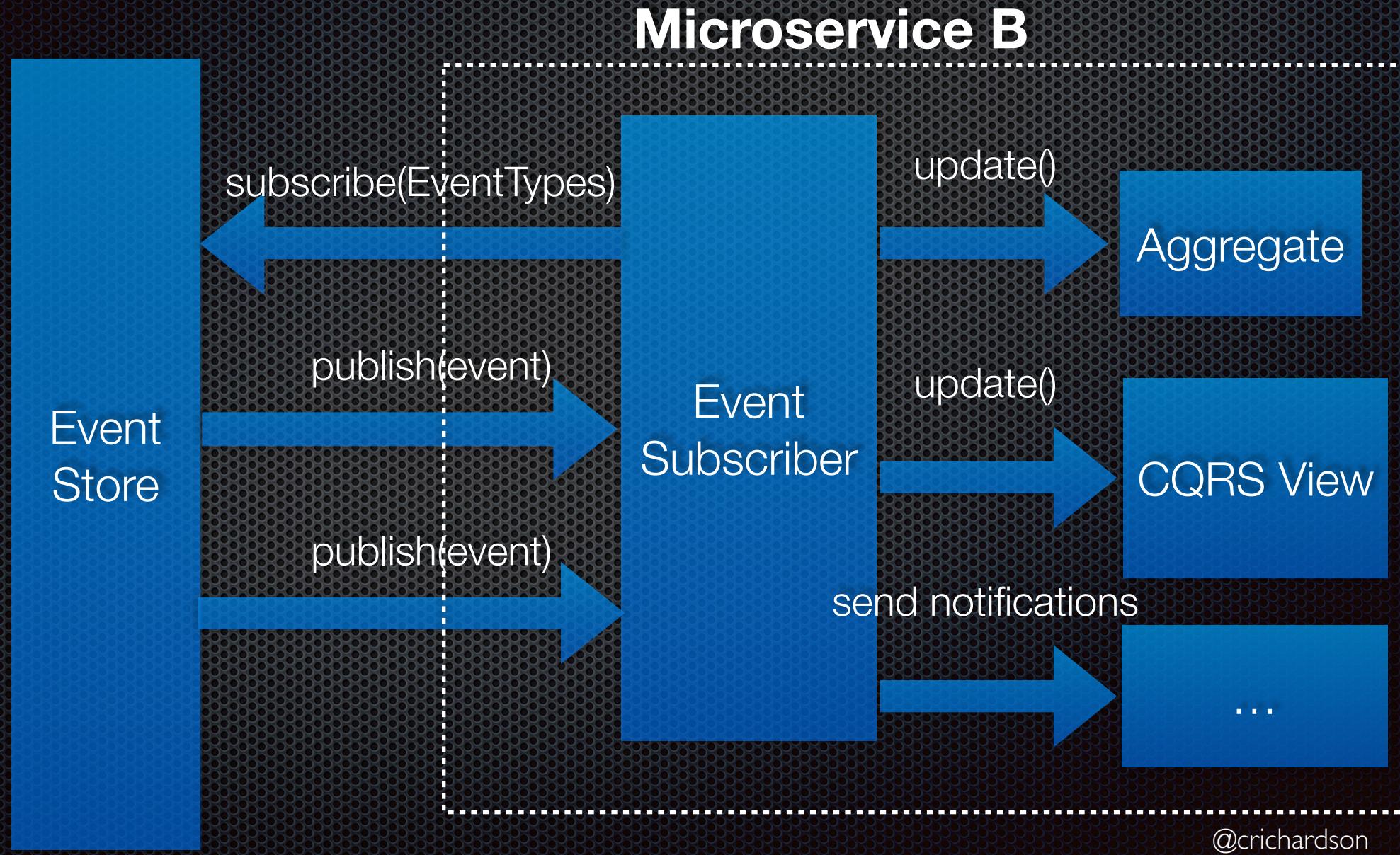


Request handling in an event-sourced application

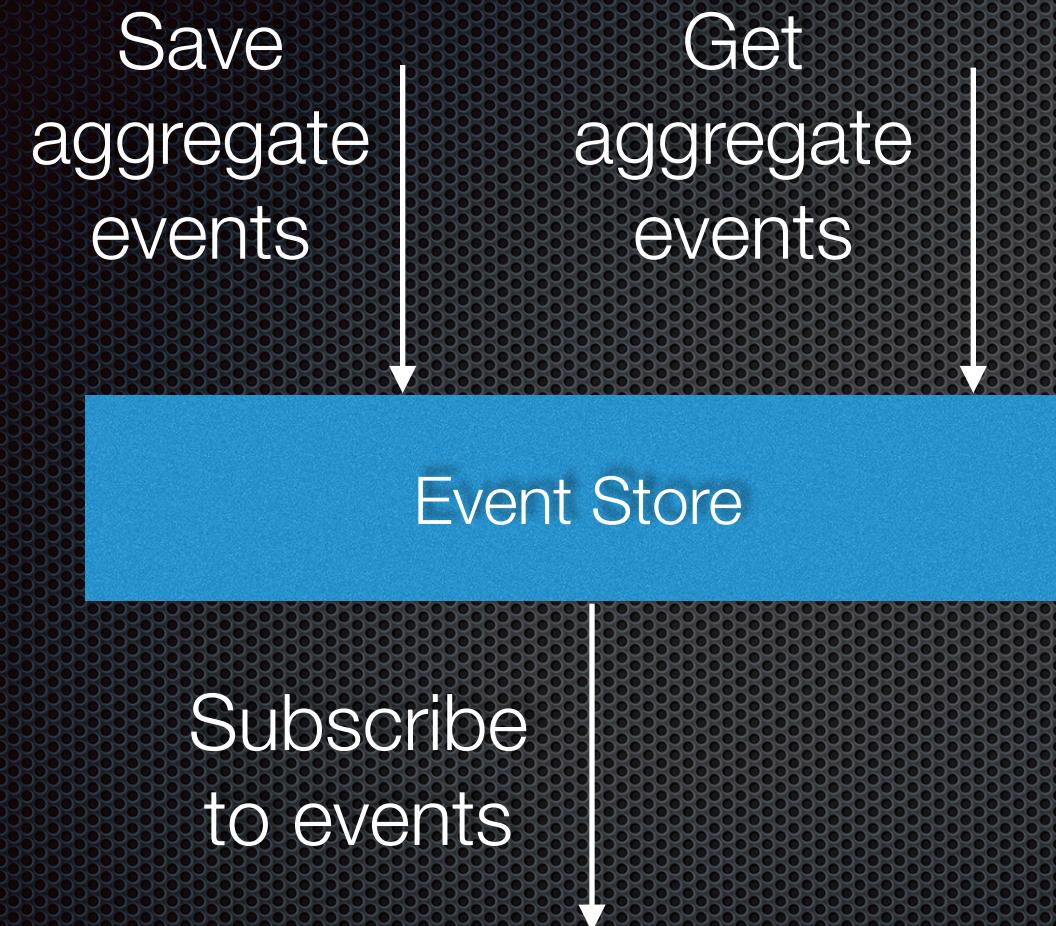
Microservice A



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

- Weird and unfamiliar
- Events = a historical record of your bad design decisions
- Handling duplicate events can be tricky
- Application must handle eventually consistent data
- Event store only directly supports PK-based lookup => use Command Query Responsibility Segregation (CQRS) to handle queries

Agenda

- Why event sourcing?
- Overview of event sourcing
- ACID-free design
- Designing a domain model based on event sourcing
- Implementing queries in an event sourced application
- Event sourcing and microservices

Implementing createOrder()

POST /orders



```
class OrderServiceImpl {  
  
    public Order createOrder() {  
        ... Creates Order ...  
    }  
  
}
```

Implement requirements and preserve invariants

Story

As a customer
I want to place an order
So that I get the needed products

Scenario

Given that my available credit is \$1500
When I place a \$250 order
Then the order is created
Then my available credit is \$1250

Scenario

Given that my available credit is \$1500
When I place a \$2500 order
Then the order is rejected
Then my available credit is \$1500

Invariant:

$\text{sum(open order.total)} \leq \text{customer.creditLimit}$

Pre conditions

Post conditions

createOrder() updates multiple aggregates

- Updates **Customer** aggregate's available credit
- **Creates** Order aggregate
- Associates **Order** with **Customer**

Old-style ACID....

BEGIN TRANSACTION

... VERIFY CREDIT ...

... CREATE ORDER...

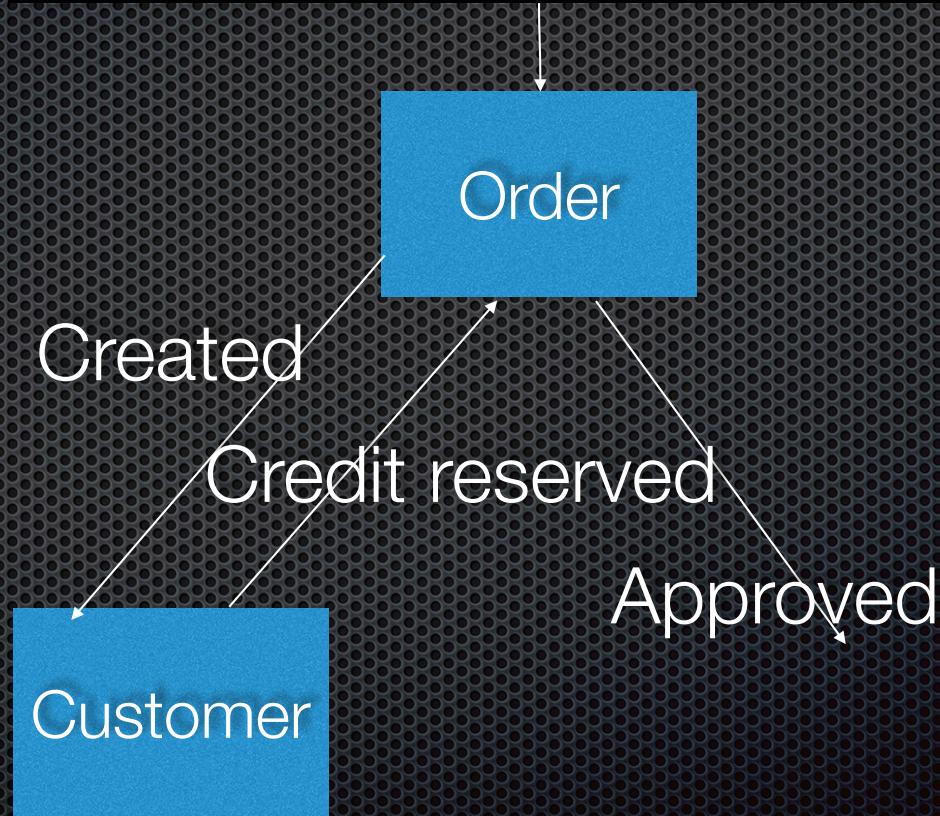
COMMIT TRANSACTION

.... becomes eventually consistent (BASE)

- Updating multiple aggregates
 - multi-step, event-driven flow
 - each step updates one Aggregate
- Service creates saga to coordinate workflow
 - A state machine
 - Part of the domain, e.g. Order aggregate OR Synthetic aggregate
- Post-conditions and invariants eventually become true

Creating an order

```
public Order createOrder() {  
    ... Creates Order ...  
}
```



Need compensating transactions

- Pre-conditions might be false when attempting to update an aggregate
- Credit check fails \Rightarrow cancel order
- Credit check succeeded but customer cancels order \Rightarrow undo credit reservation
- ...

Agenda

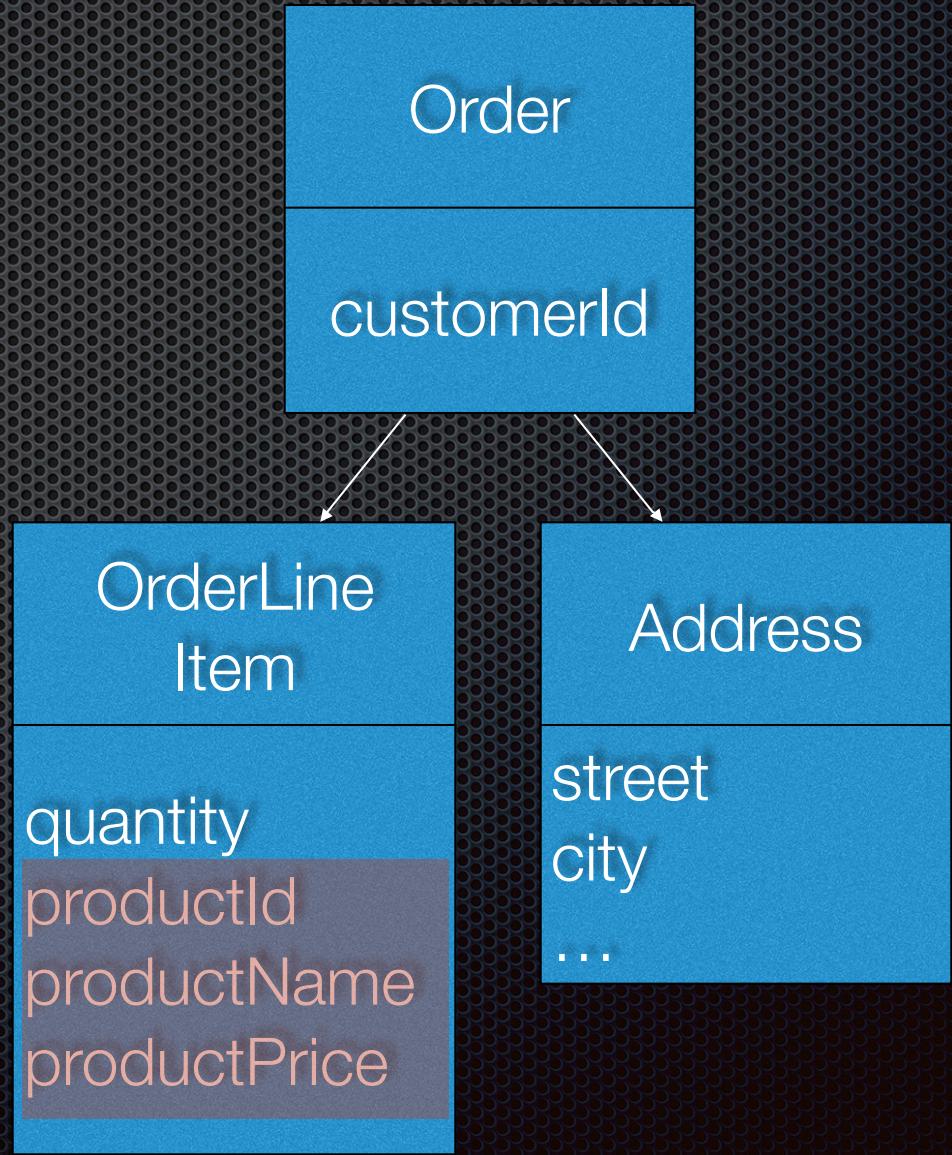
- Why event sourcing?
- Overview of event sourcing
- ACID-free design
- Designing a domain model based on event sourcing
- Implementing queries in an event sourced application
- Event sourcing and microservices

Use the familiar building blocks of DDD

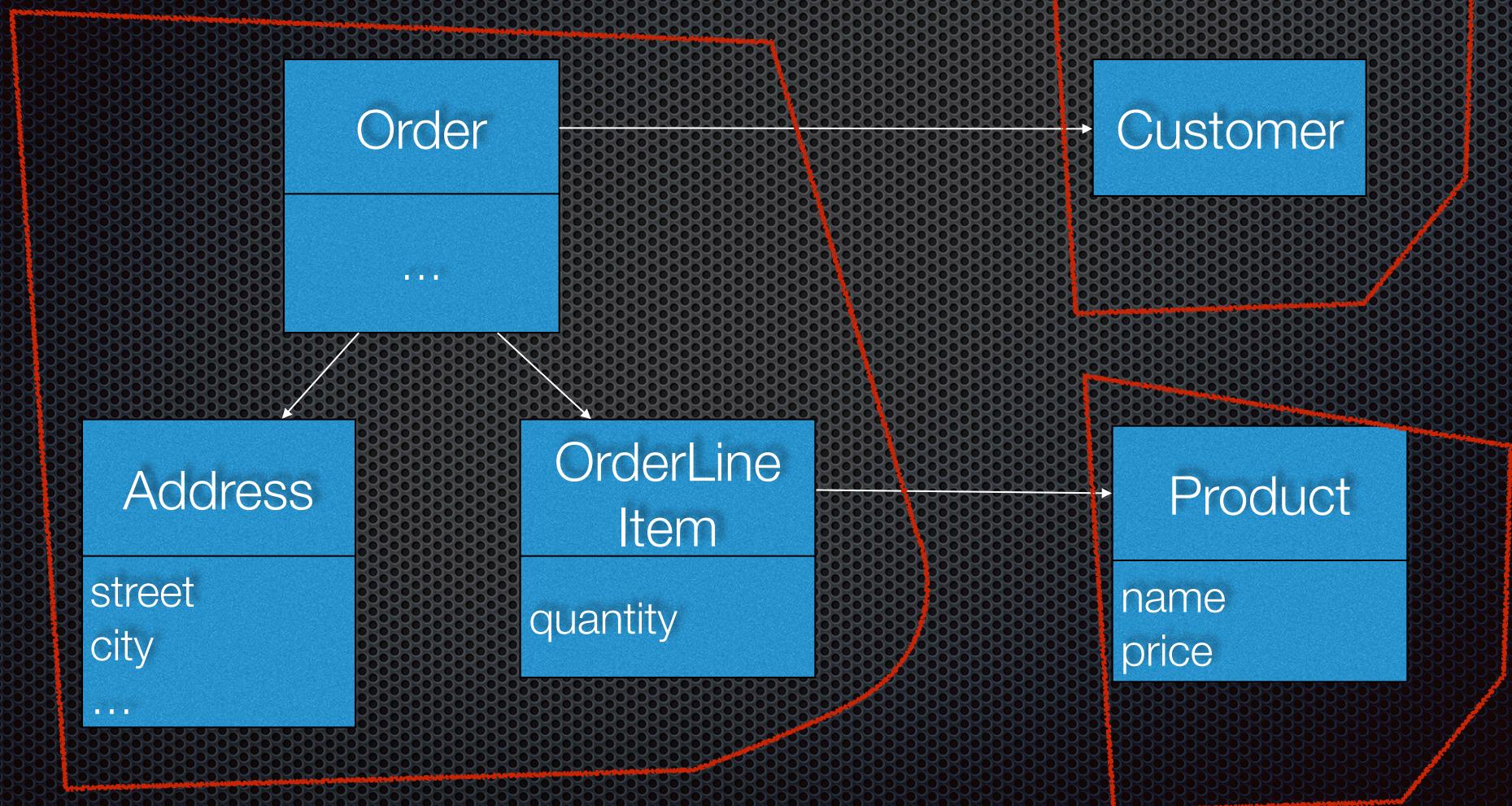
- Entity
- Value object
- Services
- Repositories
- Aggregates <= **essential**

Aggregate design

- Graph consisting of a root entity and one or more other entities and value objects
- Each core business entity = Aggregate: e.g. customer, Account, Order, Product,
- Reference other aggregate roots via primary key
- Often contains partial copy of other aggregates' data



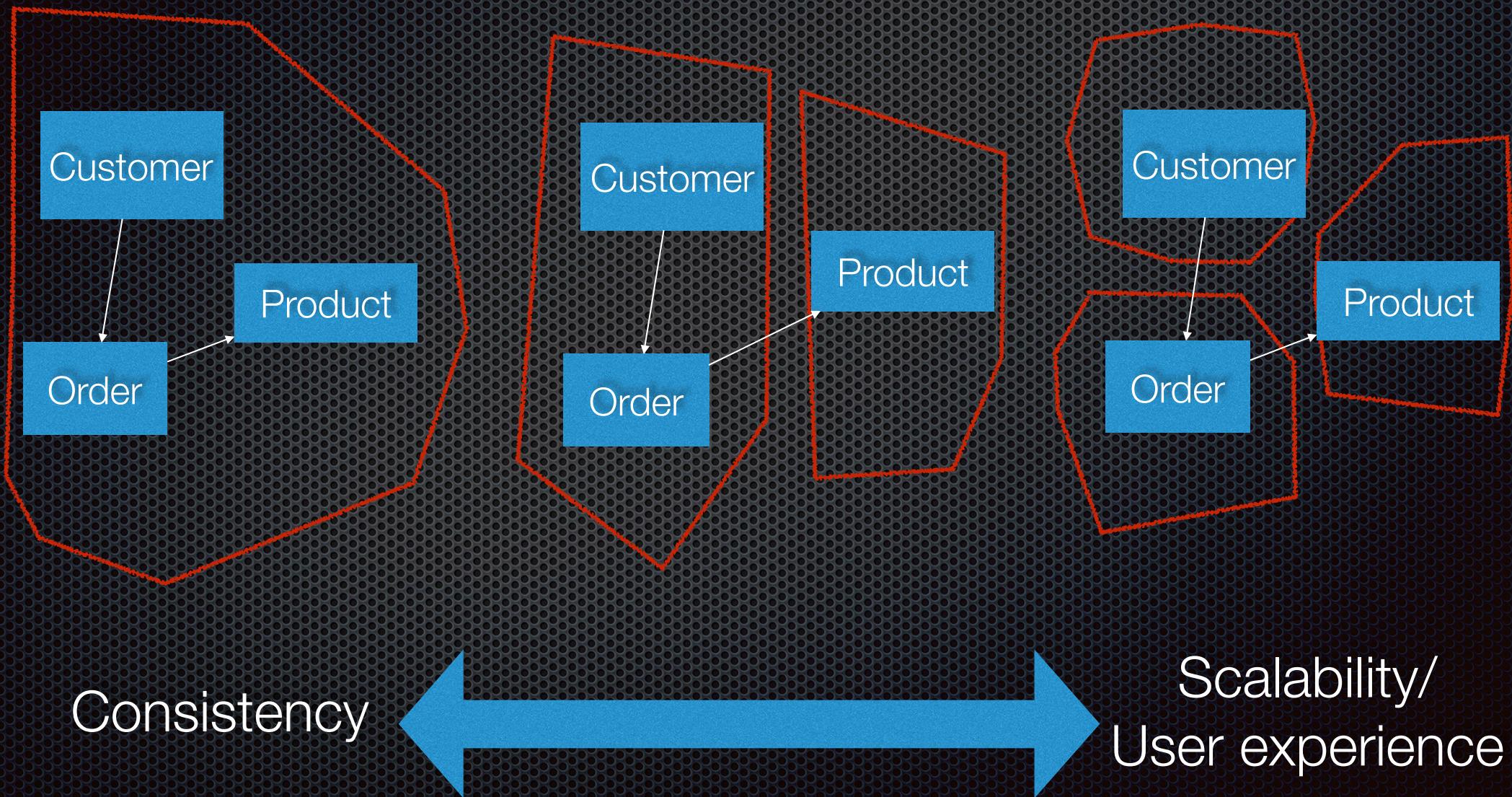
Partition the domain model into Aggregates



Transaction = processing one command by one aggregate

- No opportunity to update multiple aggregates within a transaction
- Aggregate granularity is important
- If an update must be atomic (i.e. no compensating transaction) then it must be handled by a single aggregate
 - e.g. scanning boarding pass at security checkpoint or when entering jetway

Aggregate granularity



Designing domain events

- Record state changes for an aggregate
- Part of the public API of the domain model

Required by aggregate

Enrichment:
Required by consumers

ProductAddedToCart

id : TimeUUID
productId
productName
productPrice
shoppingCartId

Example event

```
public class OrderCreatedEvent implements OrderEvent {  
    private Money orderTotal;  
    private EntityIdentifier customerId;  
  
    @Override  
    public boolean equals(Object obj) { return EqualsBuilder.reflectionEquals(this, obj); }  
  
    @Override  
    public int hashCode() { return HashCodeBuilder.reflectionHashCode(this); }  
  
    public OrderCreatedEvent(EntityIdentifier customerId, Money orderTotal) {...}  
  
    public Money getOrderTotal() { return orderTotal; }  
  
    public EntityIdentifier getCustomerId() { return customerId; }  
}
```

Designing commands

- Created by a service from incoming request
- Processed by an aggregate
- Immutable
- Contains value objects for
 - Validating request
 - Creating event
 - Auditing user activity

Example command

```
public class CreateCustomerCommand implements CustomerCommand {  
    private final String name;  
    private final Money creditLimit;  
  
    public CreateCustomerCommand(String name, Money creditLimit) {...}  
  
    public Money getCreditLimit() { return creditLimit; }  
  
    public String getName() { return name; }  
}
```

Hybrid OO/FP domain objects

OO = State + Behavior



Customer

```
creditLimit  
creditReservations : Map[OrderId, Money]
```

```
List<Event> processCommand (  
    Command aCommand) { ... }
```

```
void applyEvent (Event anEvent) { ... }
```

Aggregate traits

Used by
Event Store
to
reconstitute
aggregate

Apply event returning
updated Aggregate

```
trait Aggregate[T] { self : T =>
  def applyEvent(event : Event) : T
}

trait CommandProcessingAggregate[T, -CT] extends Aggregate[T] { self : T =>
  def processCommand(command : CT) : Seq[Event]
}
```

Map Command to Events

ReflectiveMutableCommand ProcessingAggregate

```
abstract class ReflectiveMutableCommandProcessingAggregate[T, -CT <: Command]
  extends CommandProcessingAggregate[T, CT] { self : T =>

  def applyEvent(event : Event) : T = {
    getClass.getMethod("apply", event.getClass).invoke(this, event)
    this
  }

  def processCommand(command : CT) : Seq[Event] =
    getClass.getMethod("process", command.getClass).invoke(this, command).asInstanceOf[java.util.List[Event]]
}
```

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().isGreaterThanOrEqualTo(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
    }

}
```

Event Store API

T is a subclass of Aggregate[T]

```
trait EventStore {  
  
  def save[T <: Aggregate[T]](entityClass: Class[T], events: java.util.List[Event]):  
    Observable[EntityIdAndVersion]  
  
  def update[T <: Aggregate[T]](entityClass: Class[T], entityIdAndVersion: EntityIdAndVersion, events: java.util.List[Event]):  
    Observable[EntityIdAndVersion]  
  
  def find[T <: Aggregate[T]](entityClass: Class[T], entityId: EntityIdentifier):  
    Observable[EntityWithMetadata[T]]  
  
  def find[T <: Aggregate[T]](entityClass: Class[T], entityId: EntityIdentifier, triggeringEvent: PublishedEvent):  
    Observable[EntityWithMetadata[T]]
```

Rx Observable = Future++

Creating an order

```
public class OrderServiceImpl implements OrderService {  
  
    private final AggregateRepository<Order, OrderCommand> orderRepository;  
  
    public OrderServiceImpl(AggregateRepository<Order, OrderCommand> orderRepository) {  
        this.orderRepository = orderRepository;  
    }  
  
    @Override  
    public Observable<EntityWithIdAndVersion<Order>>  
        createOrder(EntityIdentifier customerId, Money orderTotal) {  
        return orderRepository.save(new CreateOrderCommand(customerId, orderTotal));  
    }  
}
```

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 {

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

        return ctx.update(Customer.class, customerId, new ReserveCreditCommand(orderTotal, orderId));
    }
}
```

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

Agenda

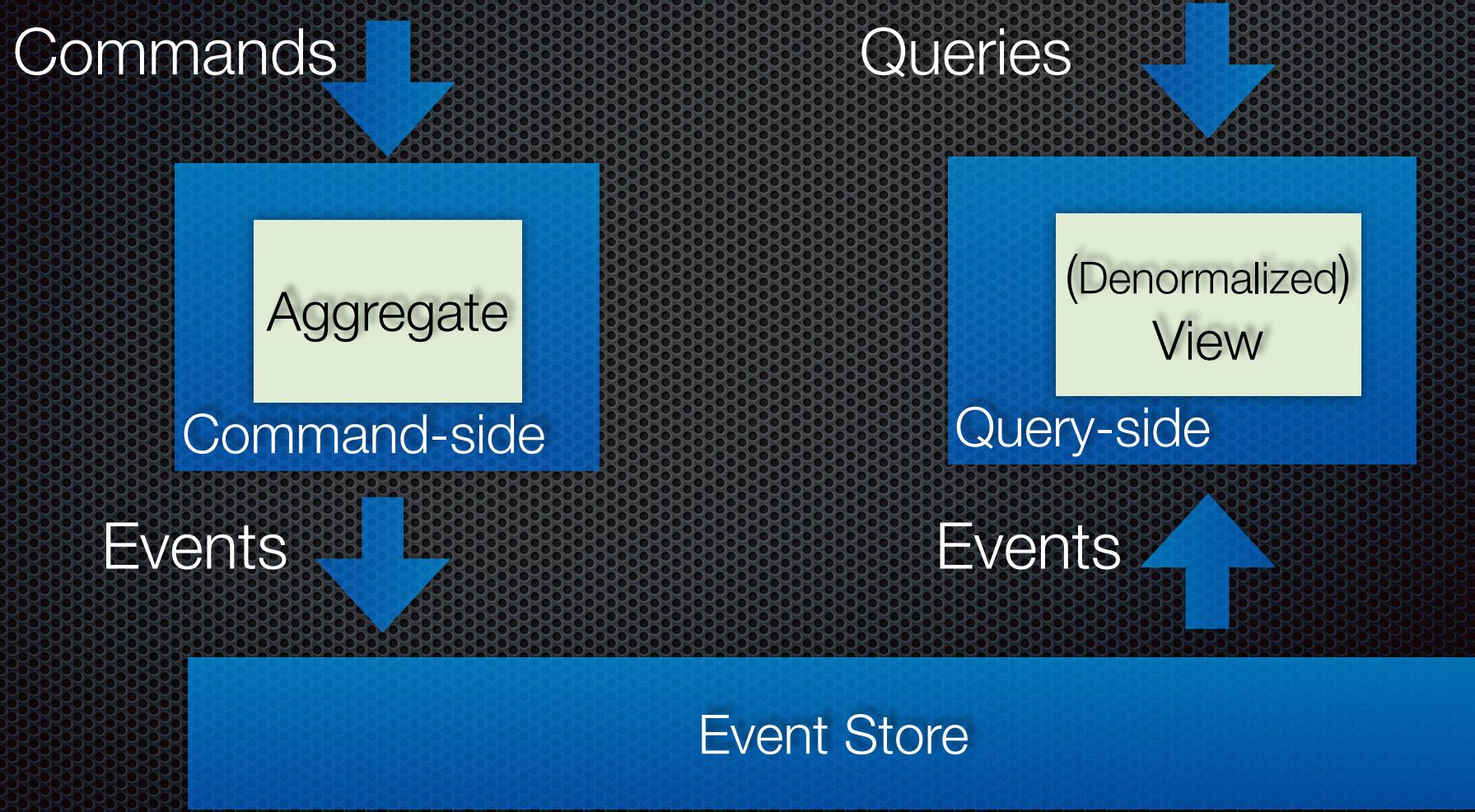
- Why event sourcing?
- Overview of event sourcing
- ACID-free design
- Designing a domain model based on event sourcing
- Implementing queries in an event sourced application
- Event sourcing and microservices

Let's imagine you want to
display a customer and their
recent orders

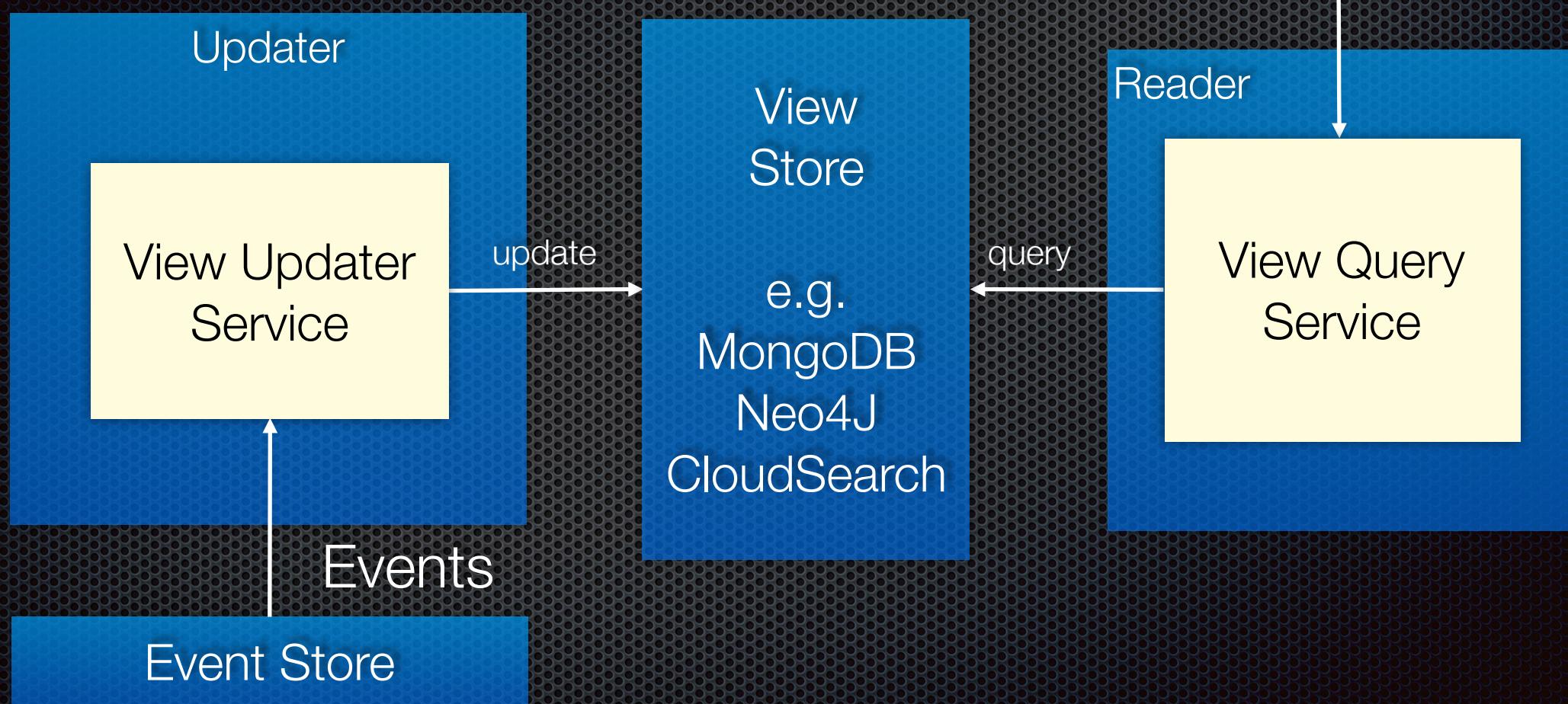
Need to ‘join’ customers and orders
BUT
Event store only supports primary
key lookup

Use Command Query
Responsibility Segregation
(CQRS)

Command Query Responsibility Segregation (CQRS)



Query-side design



Persisting a customer and order history in MongoDB

```
{  
    "_id" : "0000014f9a45004b-0a00270000000000",  
    "_class" : "net.chrisrichardson....views.orderhistory.CustomerView",  
    "version" : NumberLong(5),  
    "orders" : {  
        "0000014f9a450063-0a00270000000000" : {  
            "state" : "APPROVED",  
            "orderId" : "0000014f9a450063-0a00270000000000",  
            "orderTotal" : {  
                "amount" : "1234"  
            }  
        },  
        "0000014f9a450063-0a00270000000001" : {  
            "state" : "REJECTED",  
            "orderId" : "0000014f9a450063-0a00270000000001",  
            "orderTotal" : {  
                "amount" : "3000"  
            }  
        }  
    },  
    "name" : "Fred",  
    "creditLimit" : {  
        "amount" : "2000"  
    }  
}
```

Denormalized = efficient lookup

@crichardson

Persisting customers and order info using Spring Data for MongoDB...

```
@EventSubscriber(id="orderHistoryWorkflow")
public class OrderHistoryViewWorkflow {

    private OrderHistoryViewService orderHistoryViewService;

    @Autowired
    public OrderHistoryViewWorkflow(OrderHistoryViewService orderHistoryViewService) {
        this.orderHistoryViewService = orderHistoryViewService;
    }

    @EventHandler
    public Observable<Object> createCustomer(DispatchedEvent<CustomerCreatedEvent> de) {
        String customerId = de.getEntityIdentifier().getId();
        orderHistoryViewService.createCustomer(customerId, de.event().getName(), de.event().getCreditLimit());
        return Observable.just(null);
    }

    @EventHandler
    public Observable<Object> createOrder(DispatchedEvent<OrderCreatedEvent> de) {...}

    @EventHandler
    public Observable<Object> orderApproved(DispatchedEvent<OrderApprovedEvent> de) {...}

    @EventHandler
    public Observable<Object> orderRejected(DispatchedEvent<OrderRejectedEvent> de) {...}

}
```

Persisting customers and order using Spring Data for MongoDB...

```
public class OrderHistoryViewService {  
  
    private CustomerViewRepository customerViewRepository;  
  
    @Autowired  
    public OrderHistoryViewService(CustomerViewRepository customerViewRepository) {  
        this.customerViewRepository = customerViewRepository;  
    }  
  
    void createCustomer(String customerId, String customerName, Money creditLimit) {  
        CustomerView customerView = findOrCreateCustomerView(customerId);  
        customerView.setName(customerName);  
        customerView.setCreditLimit(creditLimit);  
        customerViewRepository.save(customerView);  
    }  
  
    private CustomerView findOrCreateCustomerView(String customerId) {  
        CustomerView customerView = customerViewRepository.findOne(customerId);  
        if (customerView == null) {  
            customerView = new CustomerView();  
            customerView.setId(customerId);  
            customerView = customerViewRepository.insert(customerView);  
        }  
        return customerView;  
    }  
  
}  
  
public interface CustomerViewRepository extends MongoRepository<CustomerView, String> {}
```

Other kinds of views

- AWS Cloud Search
 - Text search as-a-Service
 - View updaters batches aggregates to index
 - View query service does text search
- AWS DynamoDB
 - NoSQL as-a-Service
 - On-demand scalable - specify desired read/write capacity
 - Document and key-value data models
 - Useful for denormalized, UI oriented views

Benefits and drawbacks of CQRS

Benefits

- Necessary in an event-sourced architecture
- Separation of concerns = simpler command and query models
- Supports multiple denormalized views
- Improved scalability and performance

Drawbacks

- Complexity
- Potential code duplication
- Replication lag/eventually consistent views

Agenda

- Why event sourcing?
- Overview of event sourcing
- ACID-free design
- Designing a domain model based on event sourcing
- Implementing queries in an event sourced application
- Event sourcing and microservices

Partitioning into microservices: Use Strategic DDD

Strategic design: identify sub-domains and bounded contexts



Online store domain

Strategic design: identify sub-domains and bounded contexts



Bounded context = microservices

Command side

Customer
management

domain
model

Order management

domain
model

Query side

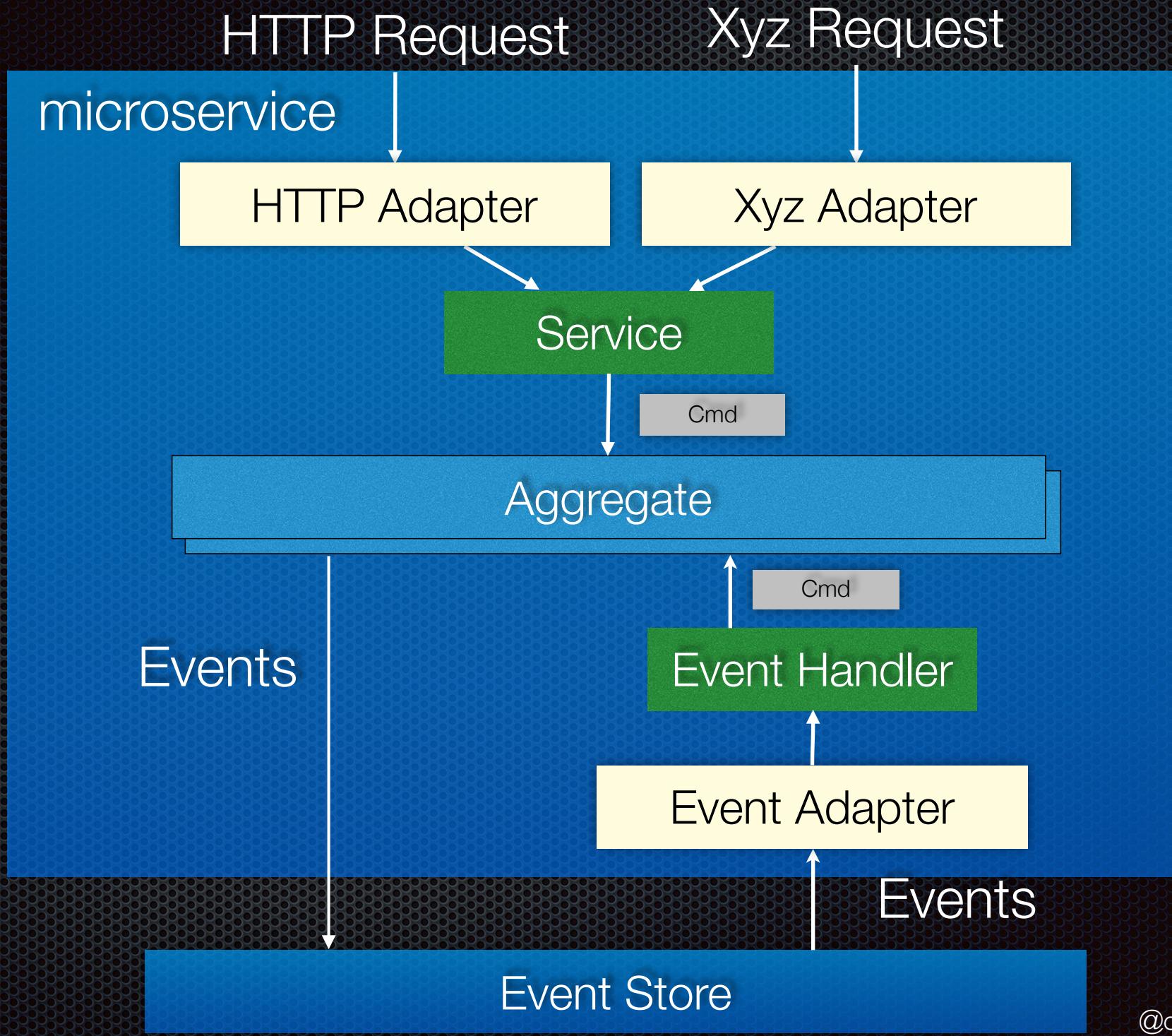
Customer view

Catalog
management

domain
model

...

domain
model



Order Controller

```
@RestController
public class OrderController {

    private OrderService orderService;

    @Autowired
    public OrderController(OrderService orderService) { this.orderService = orderService; }

    @RequestMapping(value="/orders", method= RequestMethod.POST)
    public Observable<CreateOrderResponse> createOrder(@RequestBody CreateOrderRequest createOrderRequest) {

        Observable<EntityWithIdAndVersion<Order>> order =
            orderService.createOrder(new EntityIdentifier(createOrderRequest.getCustomerId()),
                createOrderRequest.getOrderTotal());

        return order.map(ewidv -> new CreateOrderResponse(ewidv.getEntityIdentifier()));
    }
}
```

When to use microservices?

In the beginning:

- You don't need it
- It will slow you down

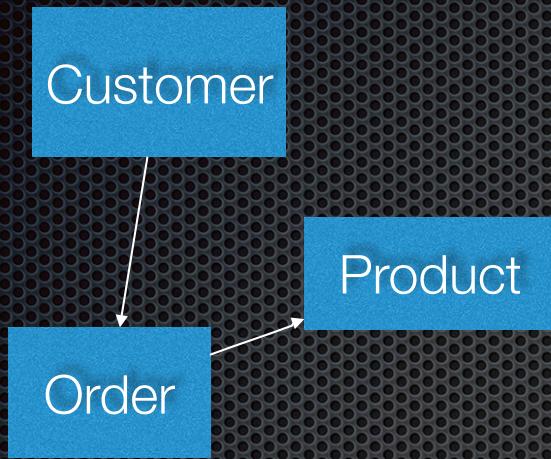


Later on:

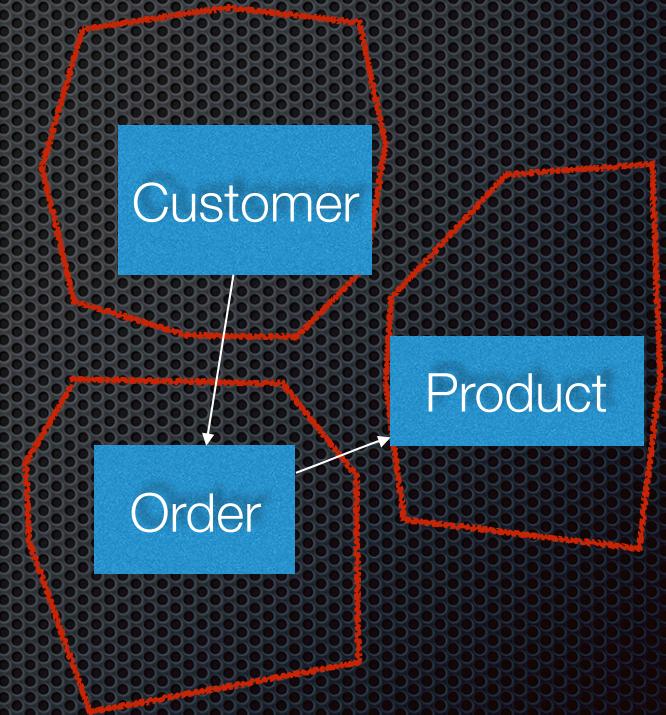
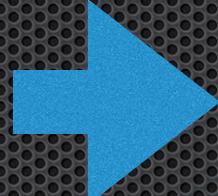
- You need it
- Refactoring dependencies is painful

Aggregates + Event Sourcing
=
Modular domain model

Modular domain model

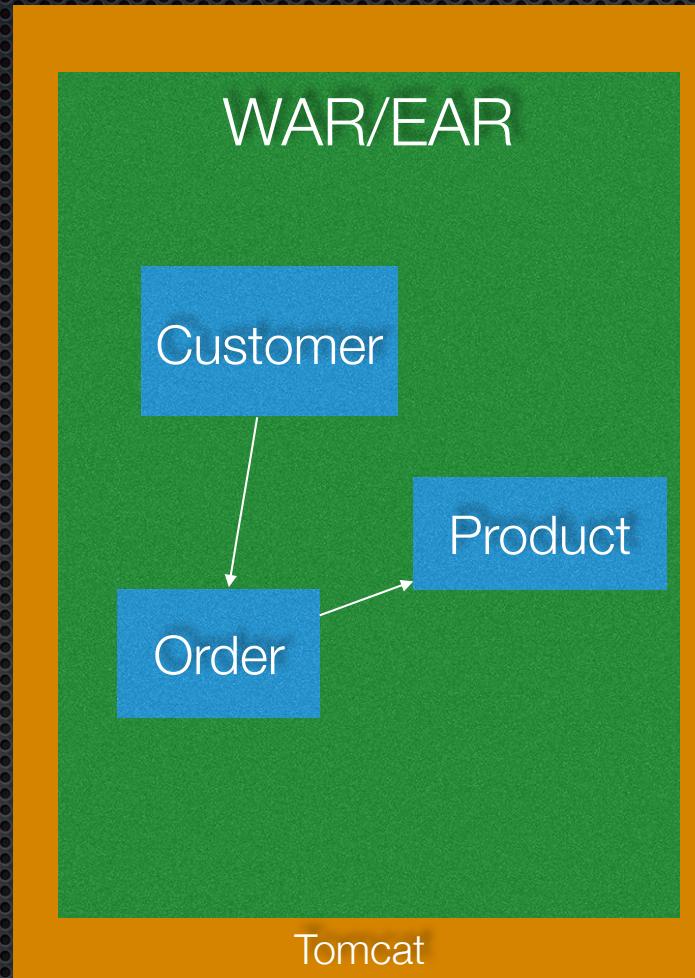


Tightly coupled
ACID



Loosely coupled aggregates
Eventually consistent

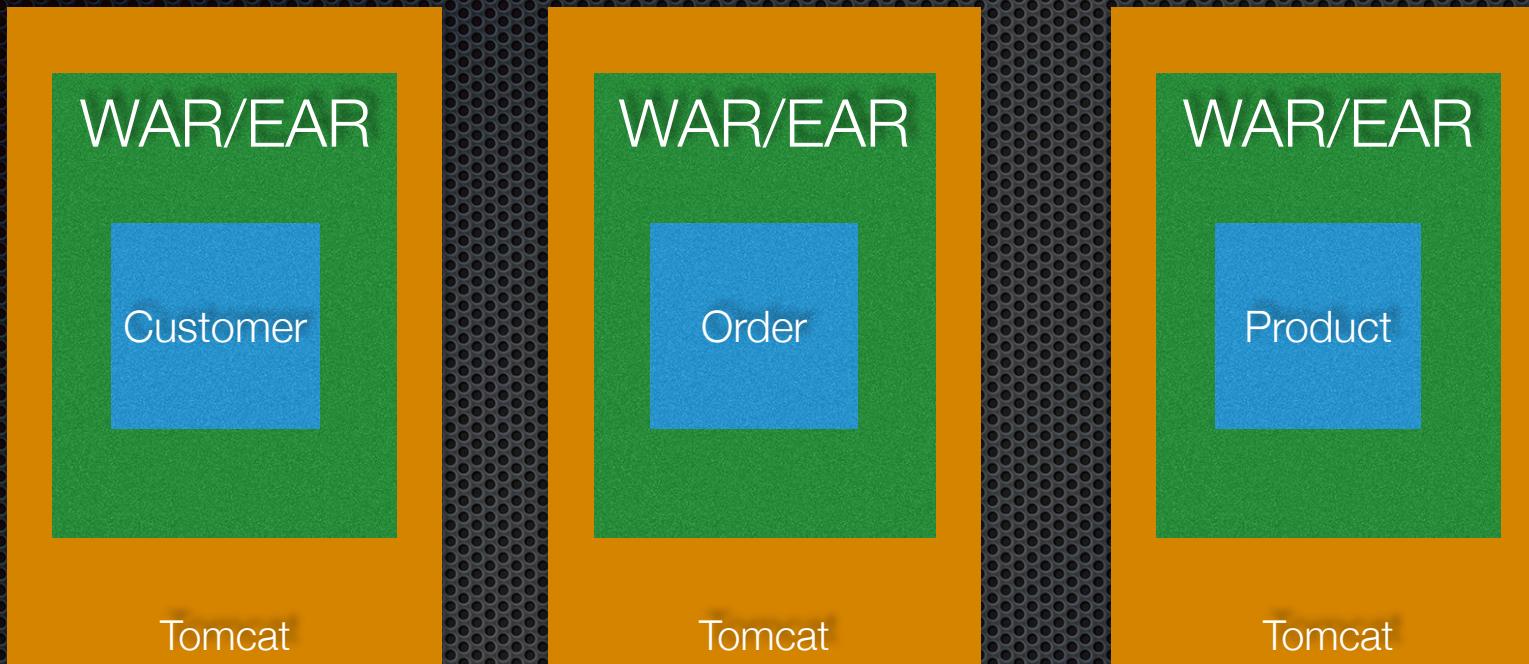
MonolithicFirst approach



Not entirely free though -
Event Sourcing premium

But no Big Ball of Mud to
untangle

Microservices deployment



Much higher –
microservices premium

Summary

- Event sourcing solves key data consistency issues with:
 - Microservices
 - Partitioned SQL/NoSQL databases
- Apply strategic DDD to identify microservices
- Apply tactical DDD to design individual services
- Use CQRS to implement materialized views for queries

• @crichton chris@chrisrichardson.net



<http://plainoldobjects.com>

<http://microservices.io>