Events on the outside, on the inside and at the core

Chris Richardson

Founder of Eventuate.io

Founder of the original CloudFoundry.com

Author of POJOs in Action

@crichardson

chris@chrisrichardson.net

http://microservices.io

http://eventuate.io

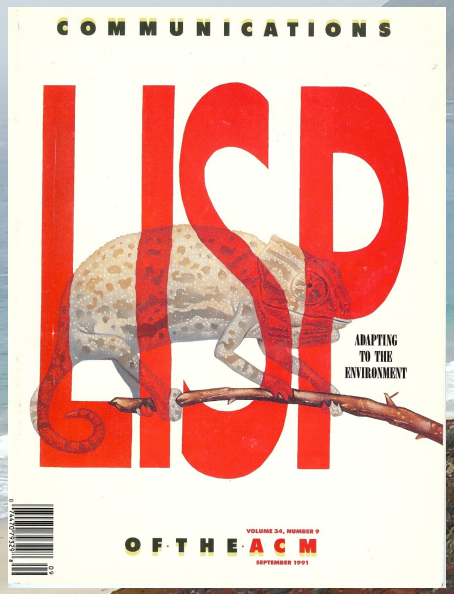
http://plainoldobjects.com

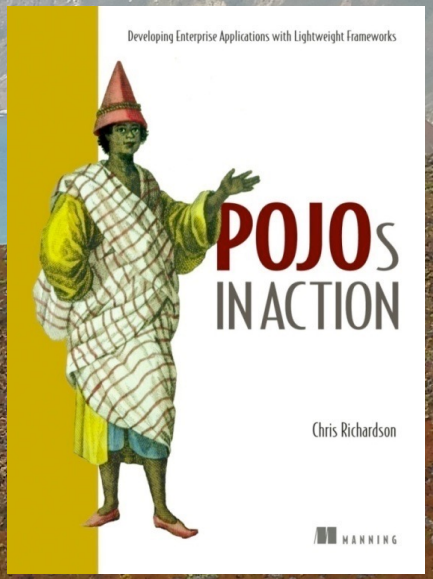
Presentation goal

Events play a key role in applications

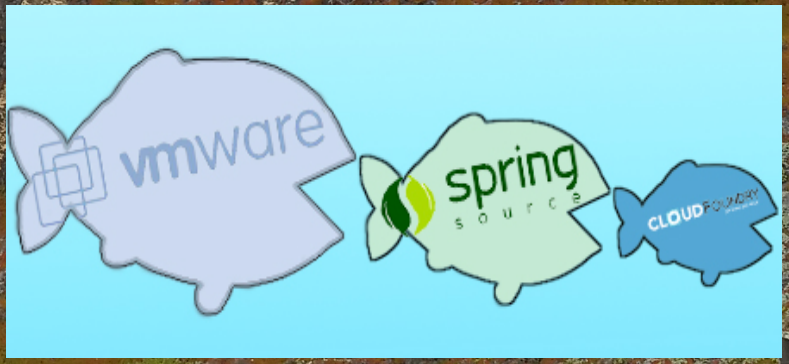
Event sourcing enables the event-driven enterprise

About Chris


COMMUNICATIONS
LISP
ADAPTING TO THE ENVIRONMENT
OF THE ACM


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


CLOUDFOUNDRY
SPRINGSOURCE
Email
SIGN IN
Sign Up
Forgot password?
HOW WE HELP
FEATURES
INFORMATION
BLOG
CONTACT US
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
CLOUDFOUNDRY
APPLICATION DEMO
Deploying Web Applications To Amazon EC2 with Cloud Foundry


vmware
spring
source
CLOUDFOUNDRY

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 to write microservices

http://eventuate.io

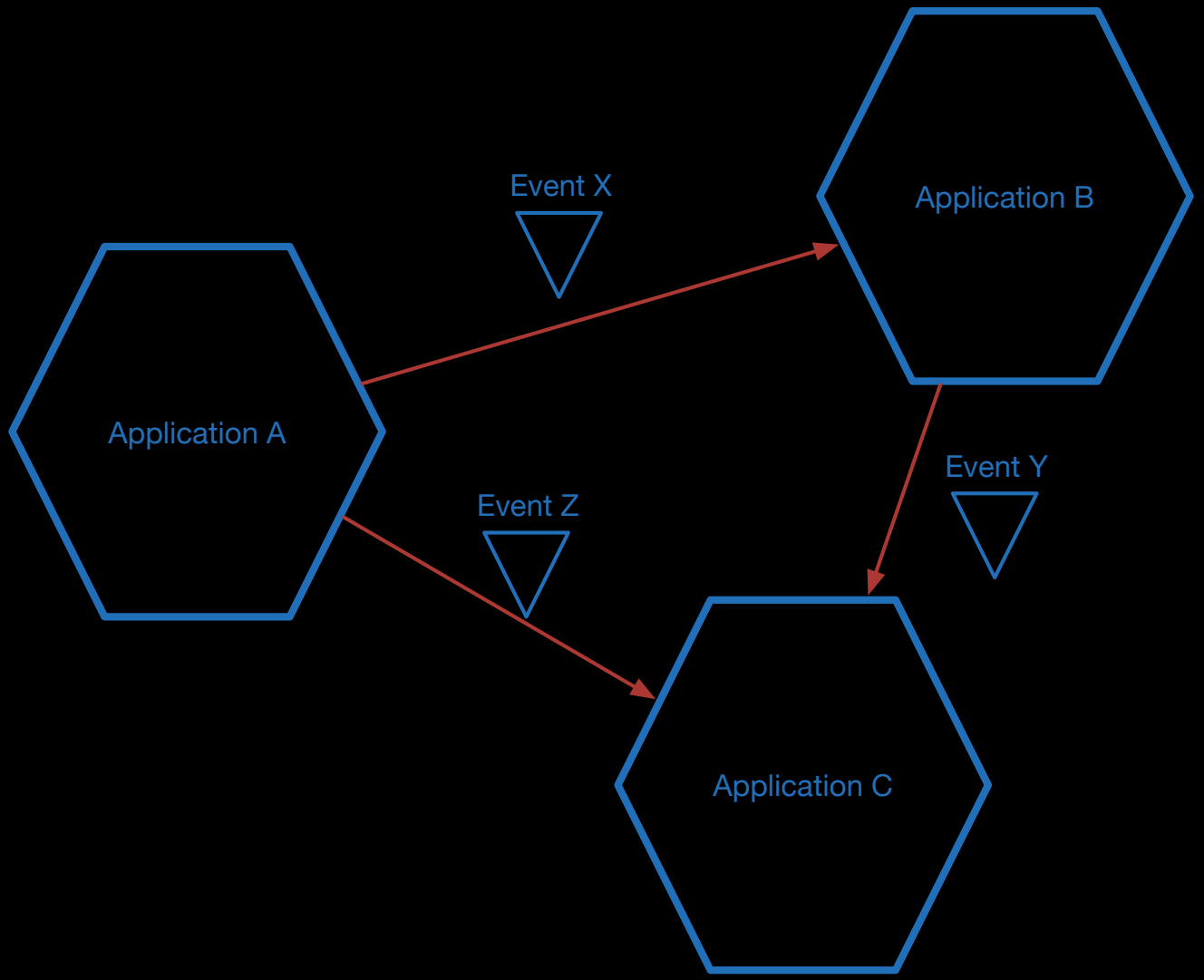
For more information

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

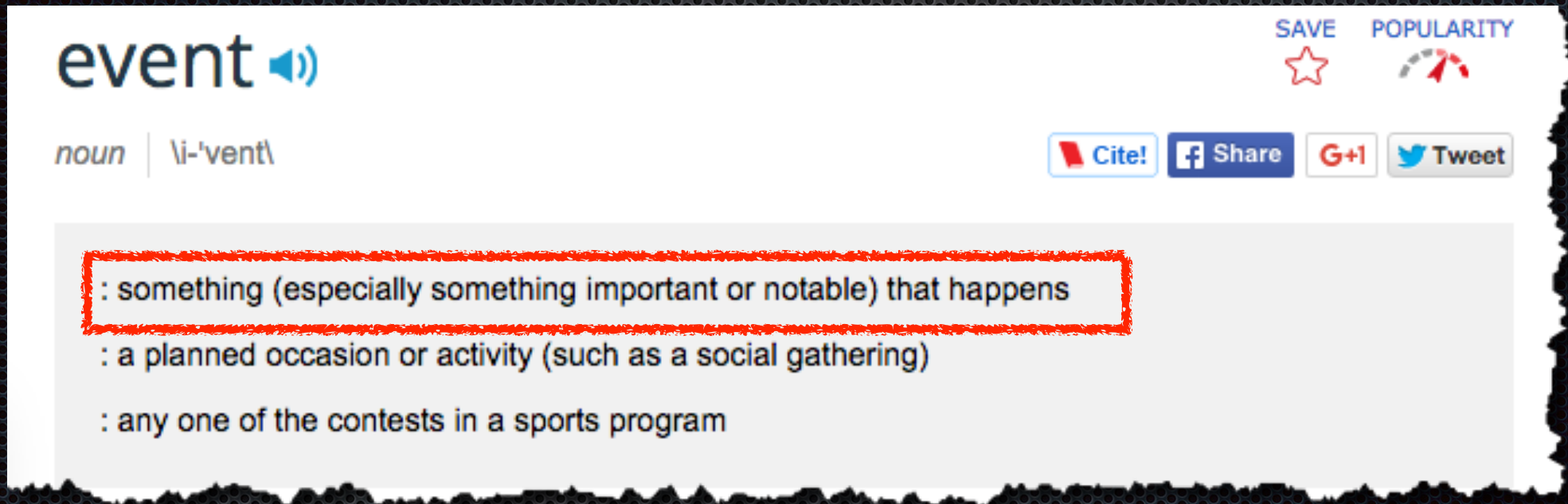
Agenda

- Events on the outside
- Events on the inside
- Events at the core with event sourcing
- Designing event-centric domain model

Events on the outside

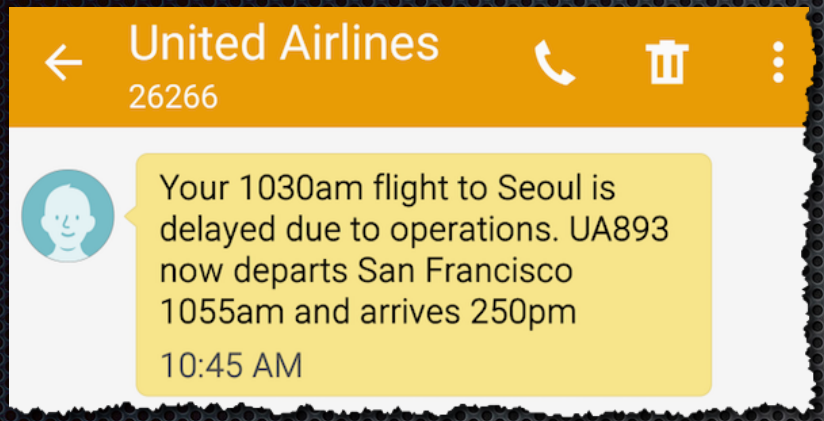


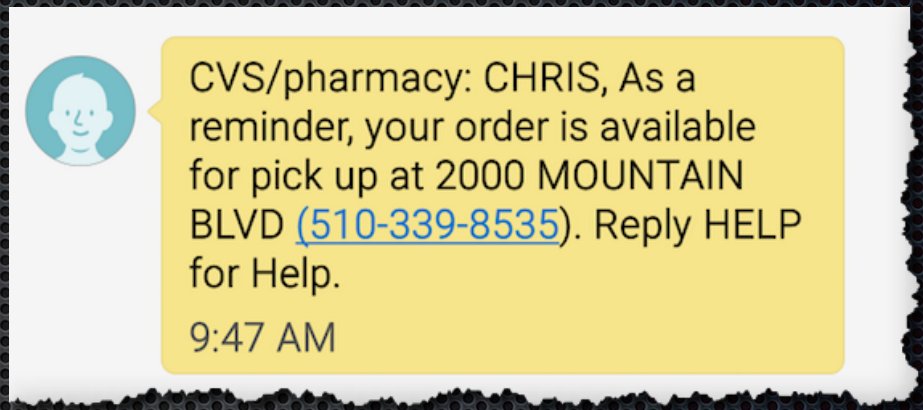
What's an event?

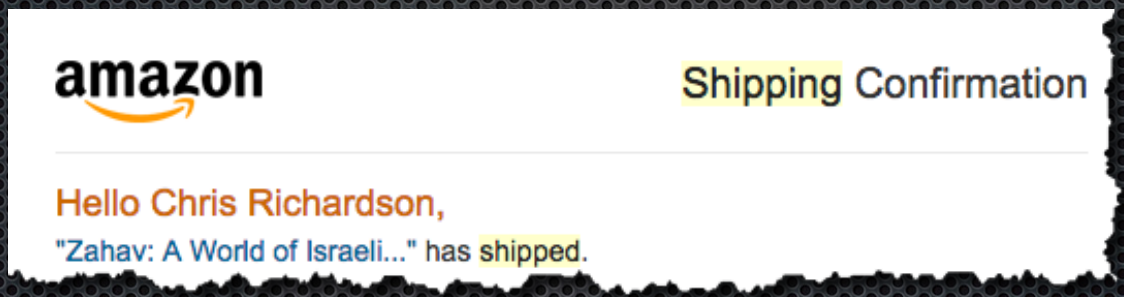


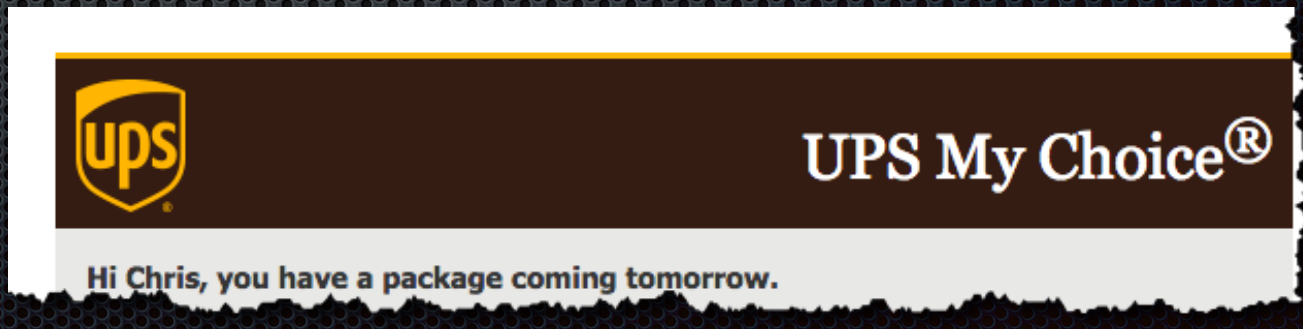
http://www.merriam-webster.com/dictionary/event

Examples of events



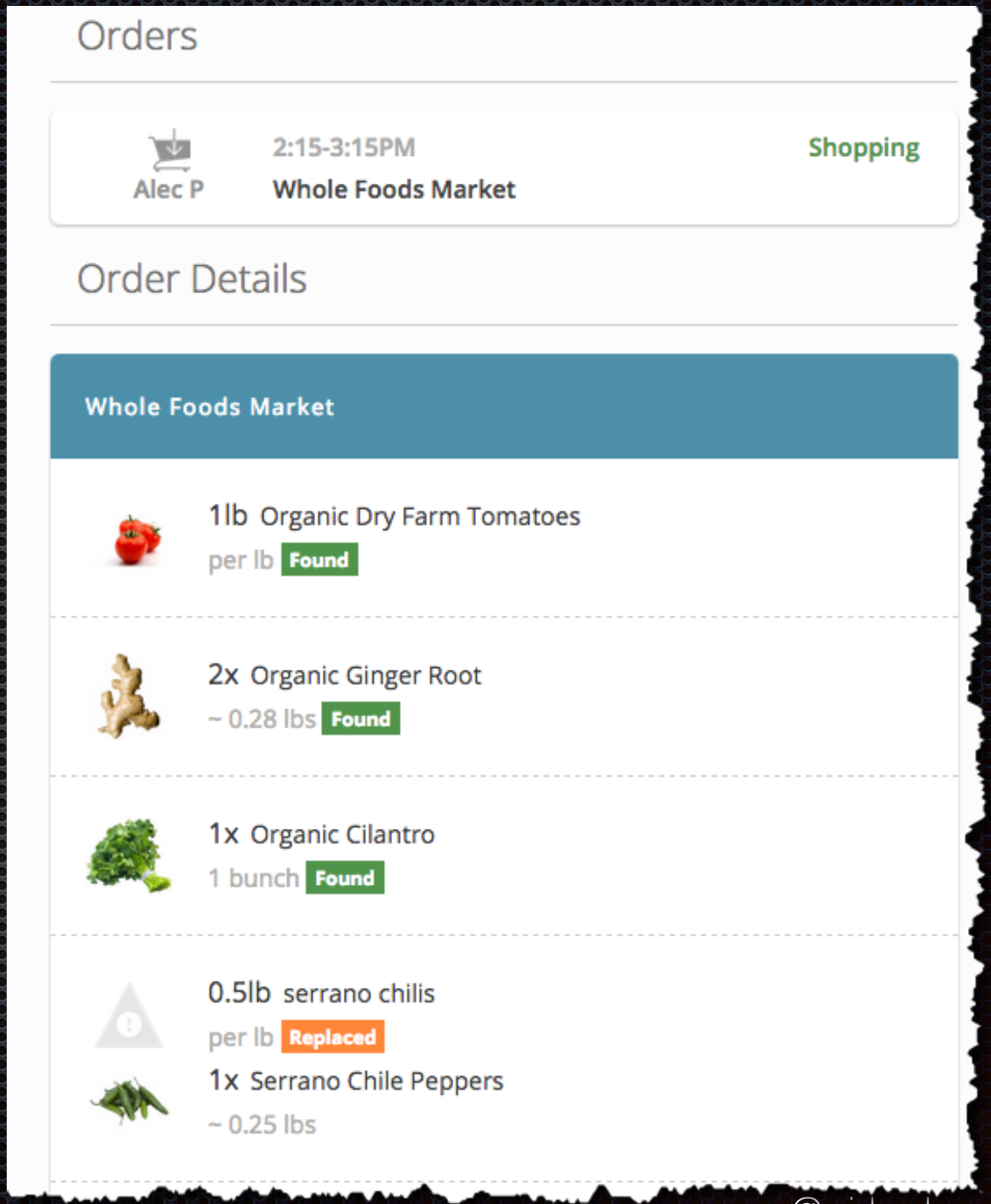






@crichardson

Instacart: event-driven grocery shopping



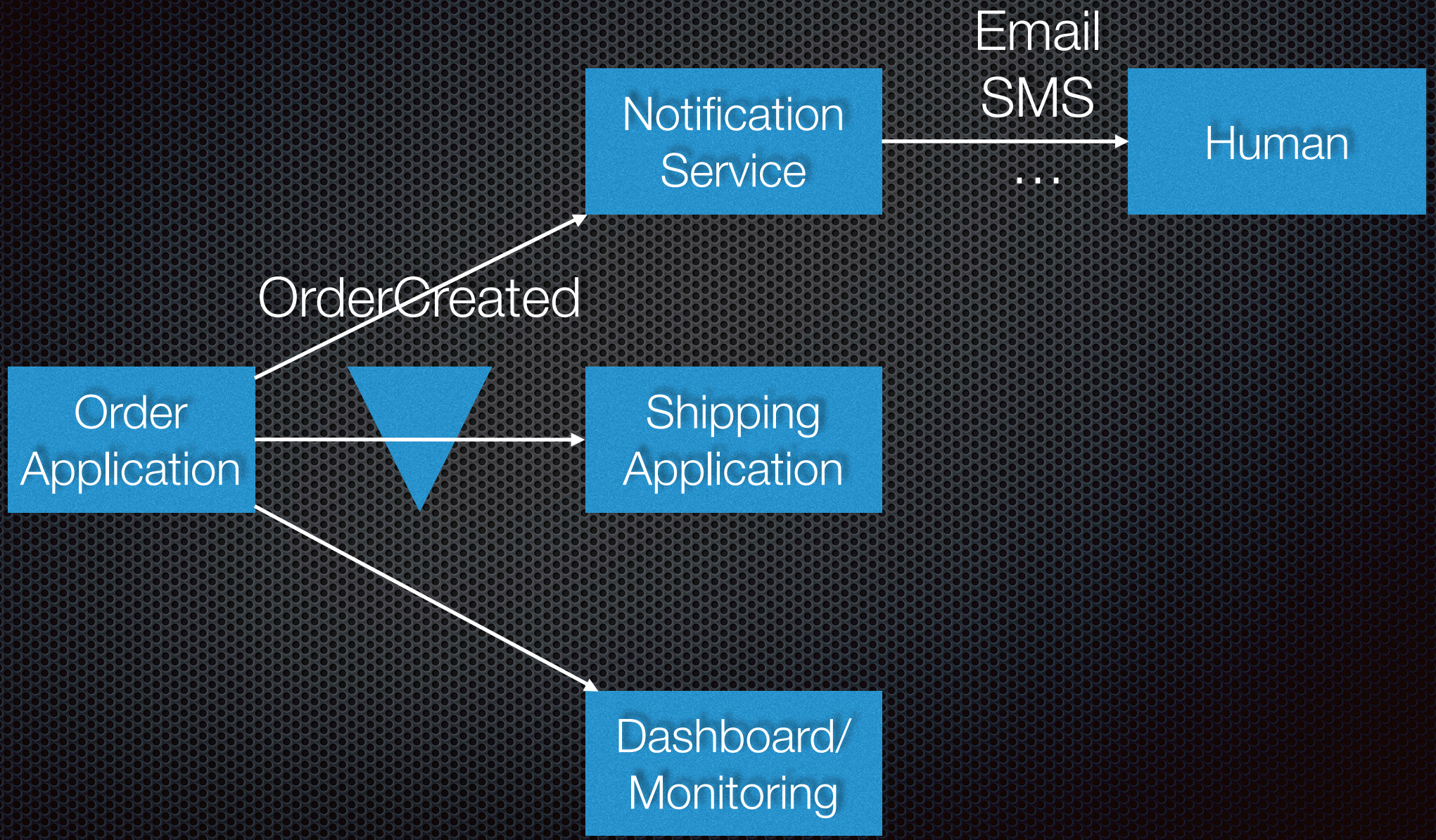
An event is produced when....

- Creation or update of a business object
- Attempt to violate a business rule

How to reliably generate events?

More on that later...

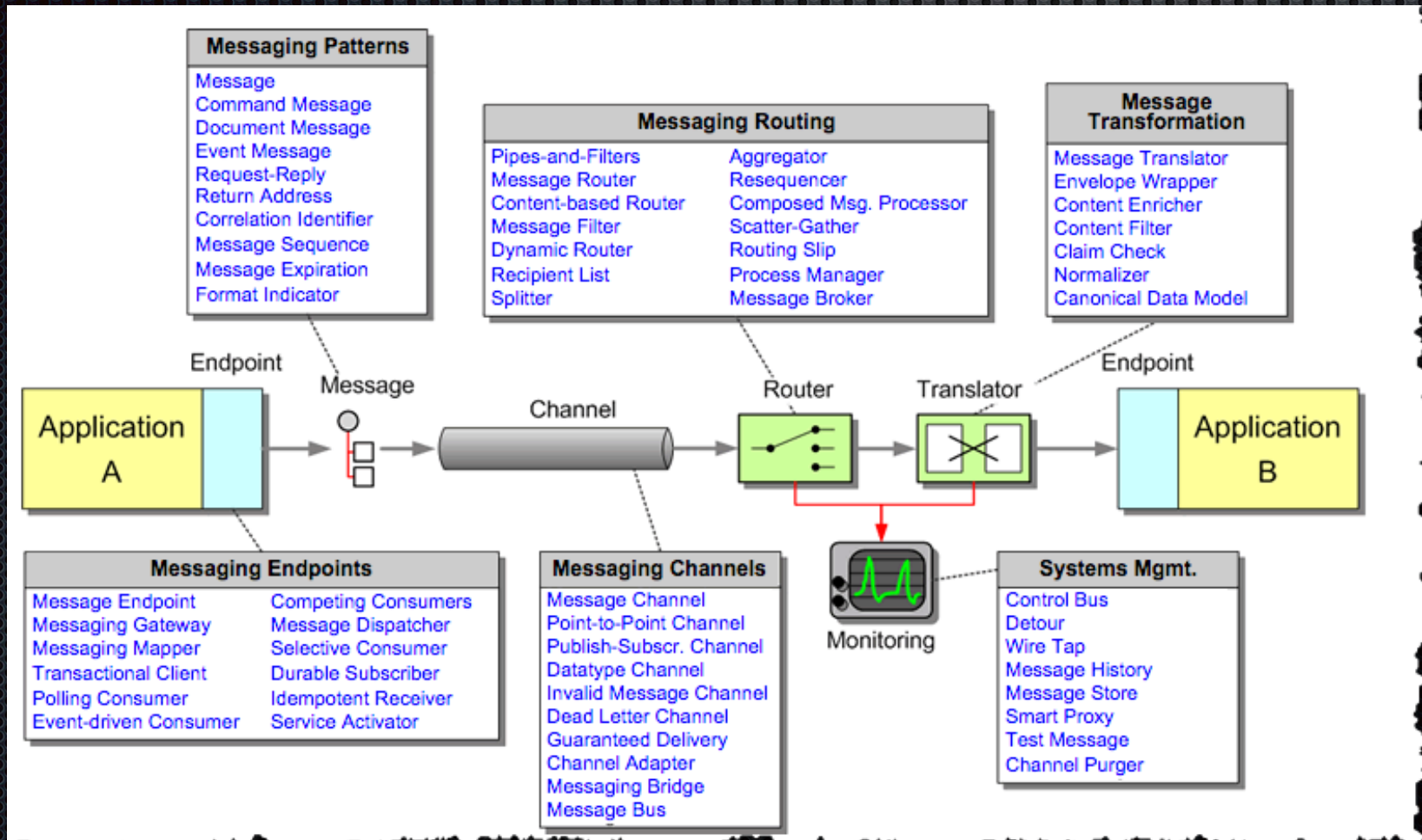
Who consumes an event?

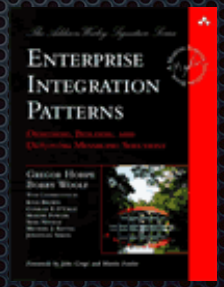


How to deliver events?

Inside the firewall

Enterprise integration patterns





- http://www.enterpriseintegrationpatterns.com/patterns/messaging/

Messaging-based IPC

Example messaging systems

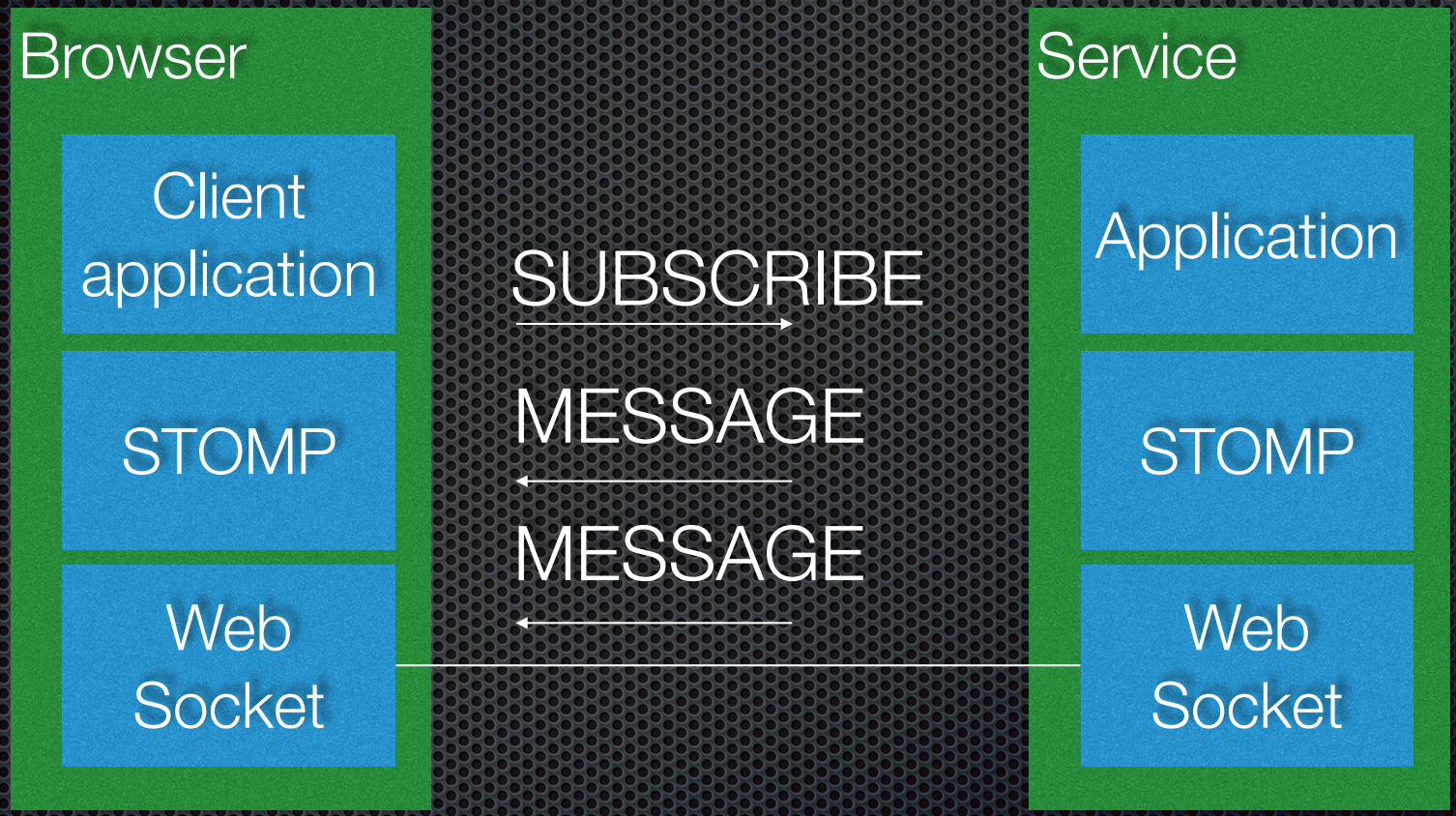
Outside the firewall

Polling for events

- HTTP
  - Periodically poll for events
- Atom Publishing Protocol (AtomPub)
  - Based on HTTP
  - Head is constantly changing
  - Tail is immutable and can be efficiently cached

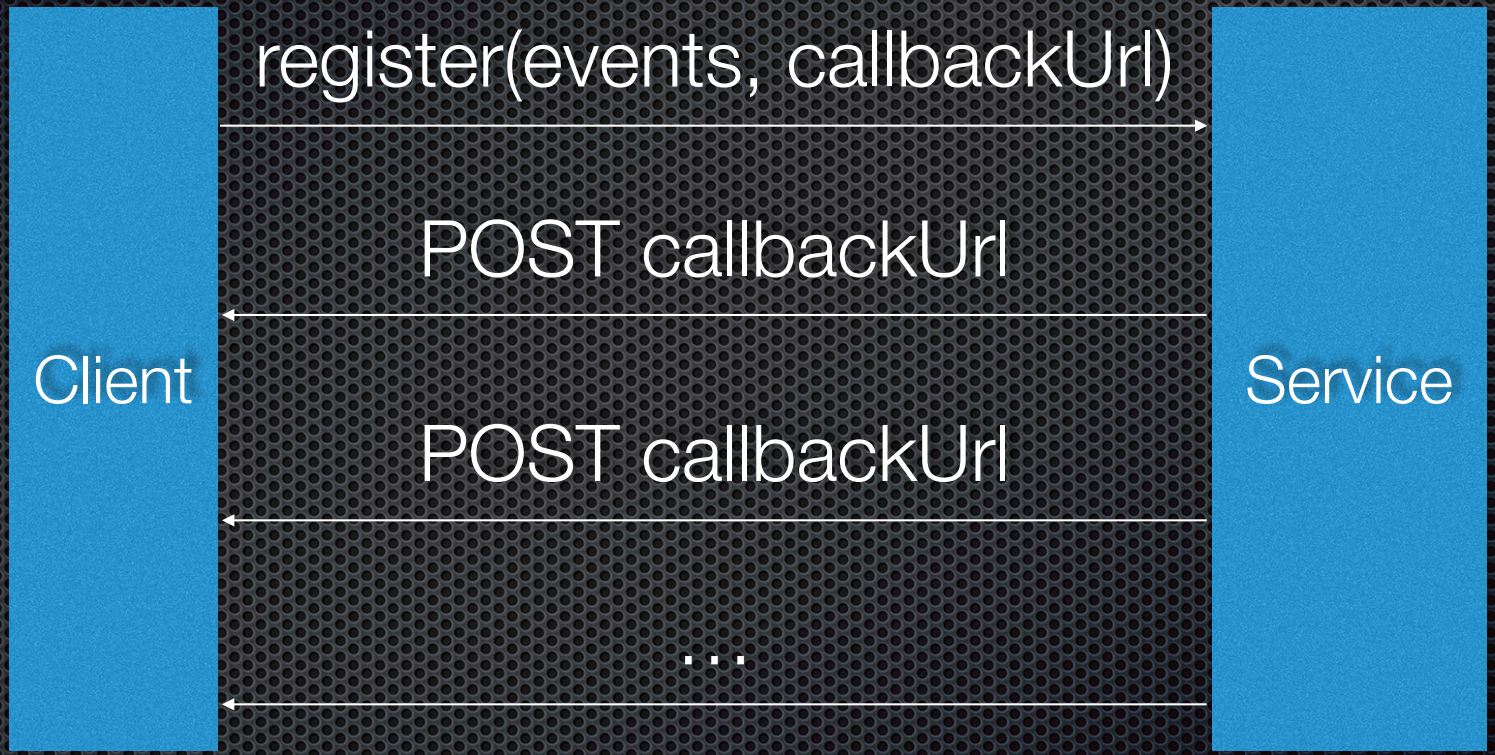
High-latency, inefficient

Using WebSockets



Low latency, more efficient, but what about past events?

Webhooks = user-defined HTTP callback



https://en.wikipedia.org/wiki/Webhook

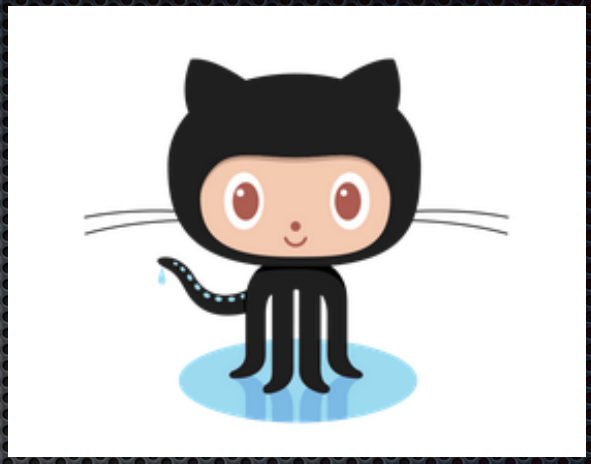
Low latency, more efficient, but what about past events?

WebHooks

=

web friendly publish/subscribe

Github webhooks



- https://developer.github.com/webhooks/
- Installed on an organization or repository
  - e.g. POST /repos/:owner/:repo/hooks
- Available events:
  - push - push to a repository
  - fork - repository is forked
  - pull_request - assigned, unassigned, …
  - push - push to a repository
  - …

Twilio - Telephony and SMS as a service

Manage resources
Send SMS
Initiate voice calls

REST API

Voice
SMS

Twilio

HTTP GET/
POST

Your
Application

TwiML doc

Phone number ⇒

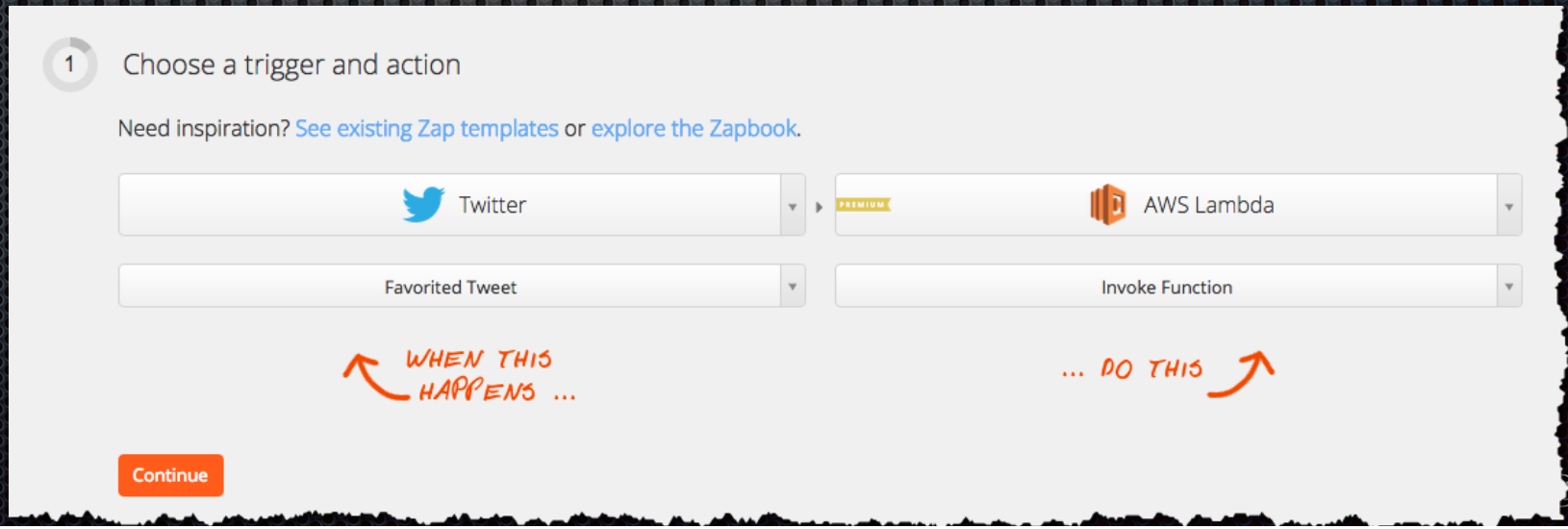
SMS URL + VOICE URL

Webhooks handle incoming SMS and voice calls

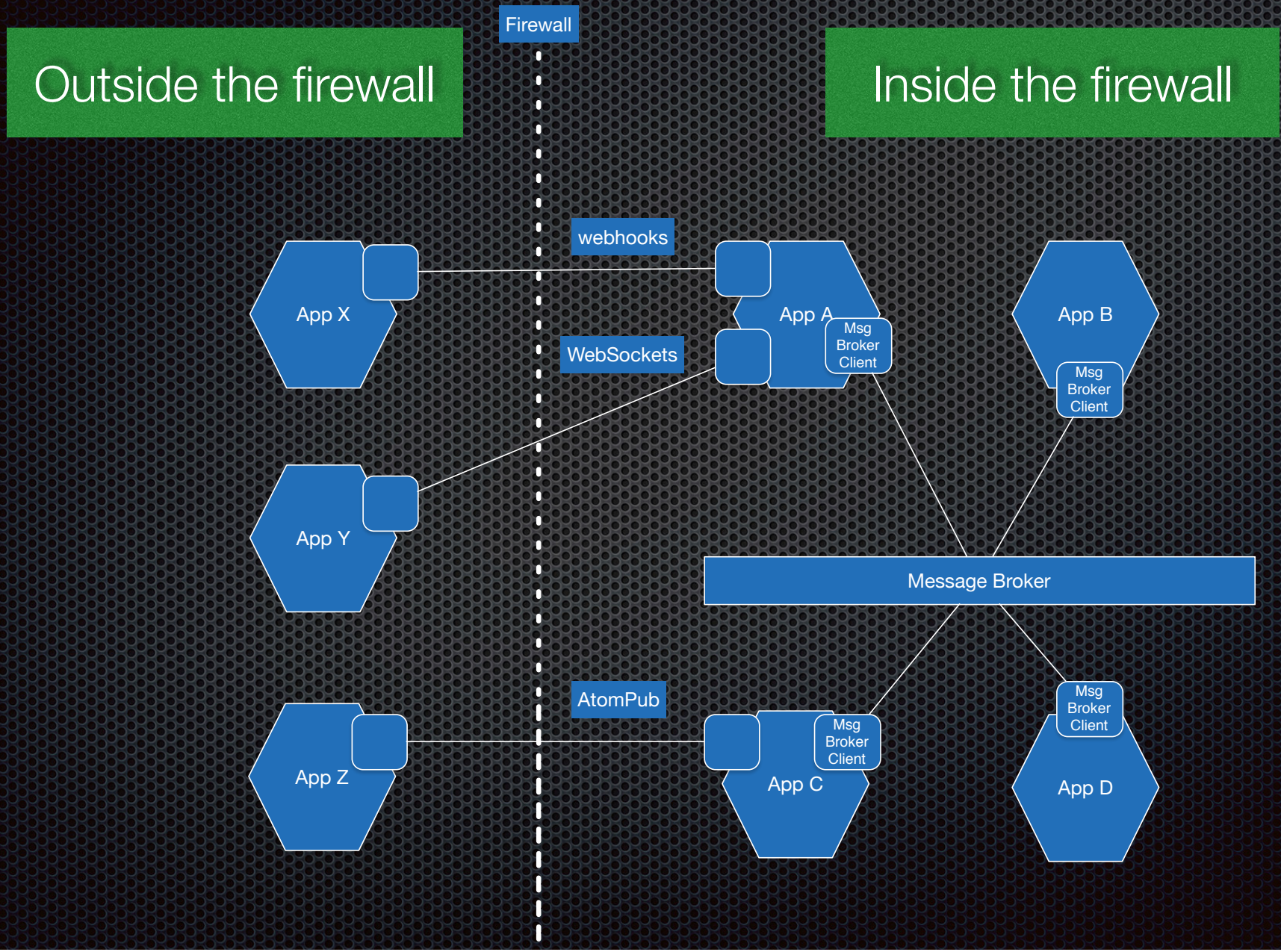
@crichardson

Integration hubs - Zapier, IFTTT

- Application abstraction:
  - Triggers - events published by application: polling or Webhooks
  - Action - operation supported by application, e.g. REST API end points



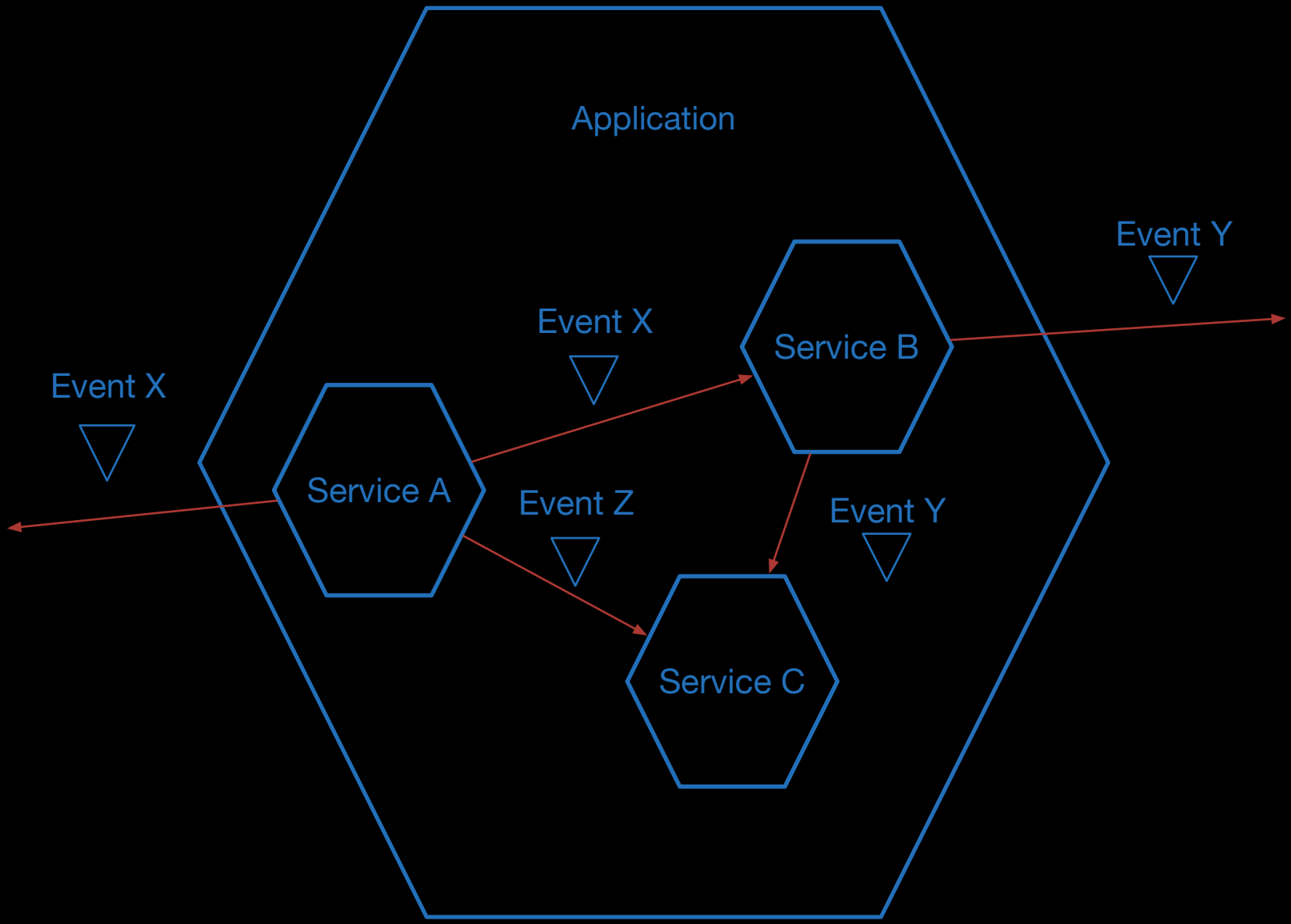
The event-driven enterprise



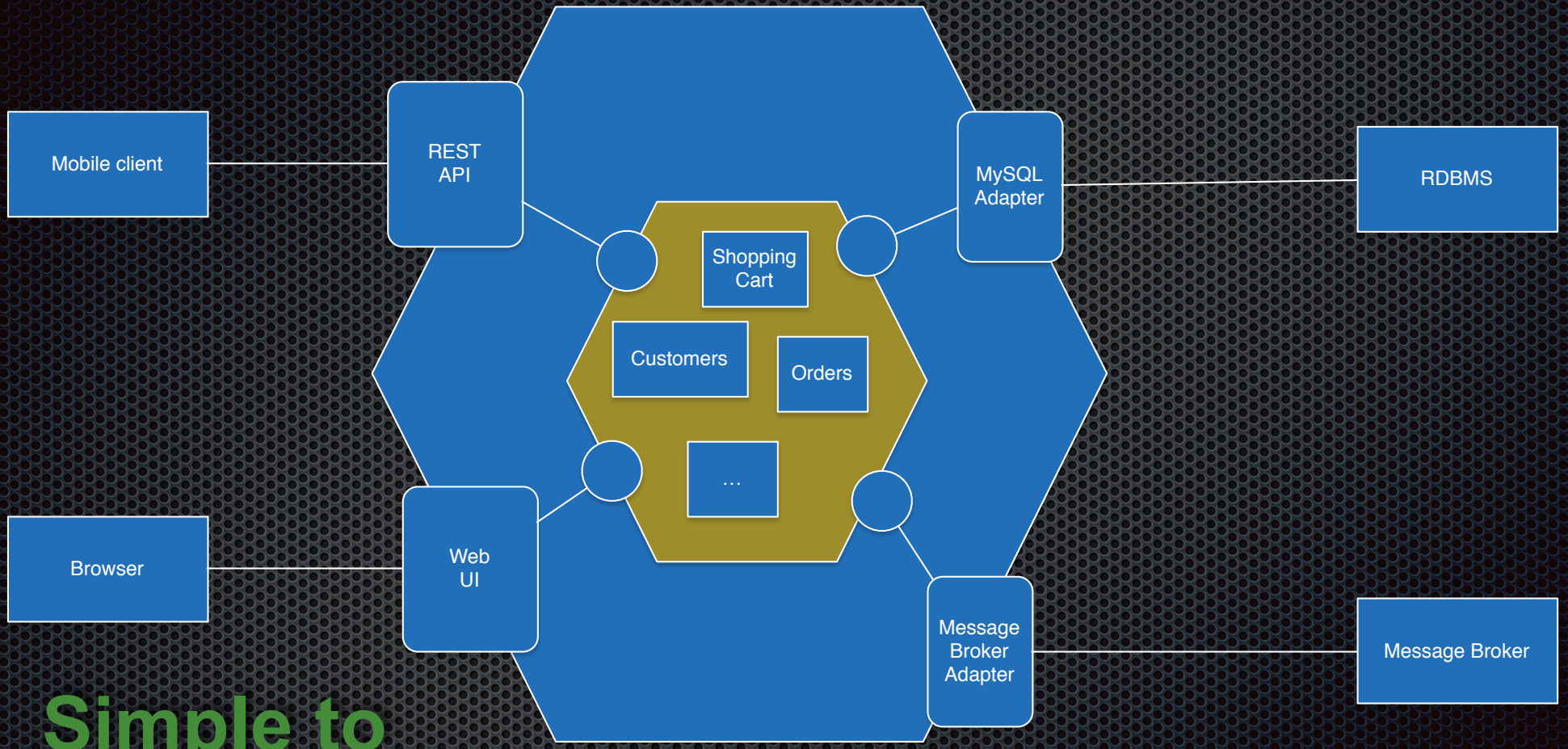
Agenda

- Events on the outside
- Events on the inside
- Events at the core with event sourcing
- Designing event-centric domain model

Events on the inside



Traditional monolithic architecture



Simple to

develop

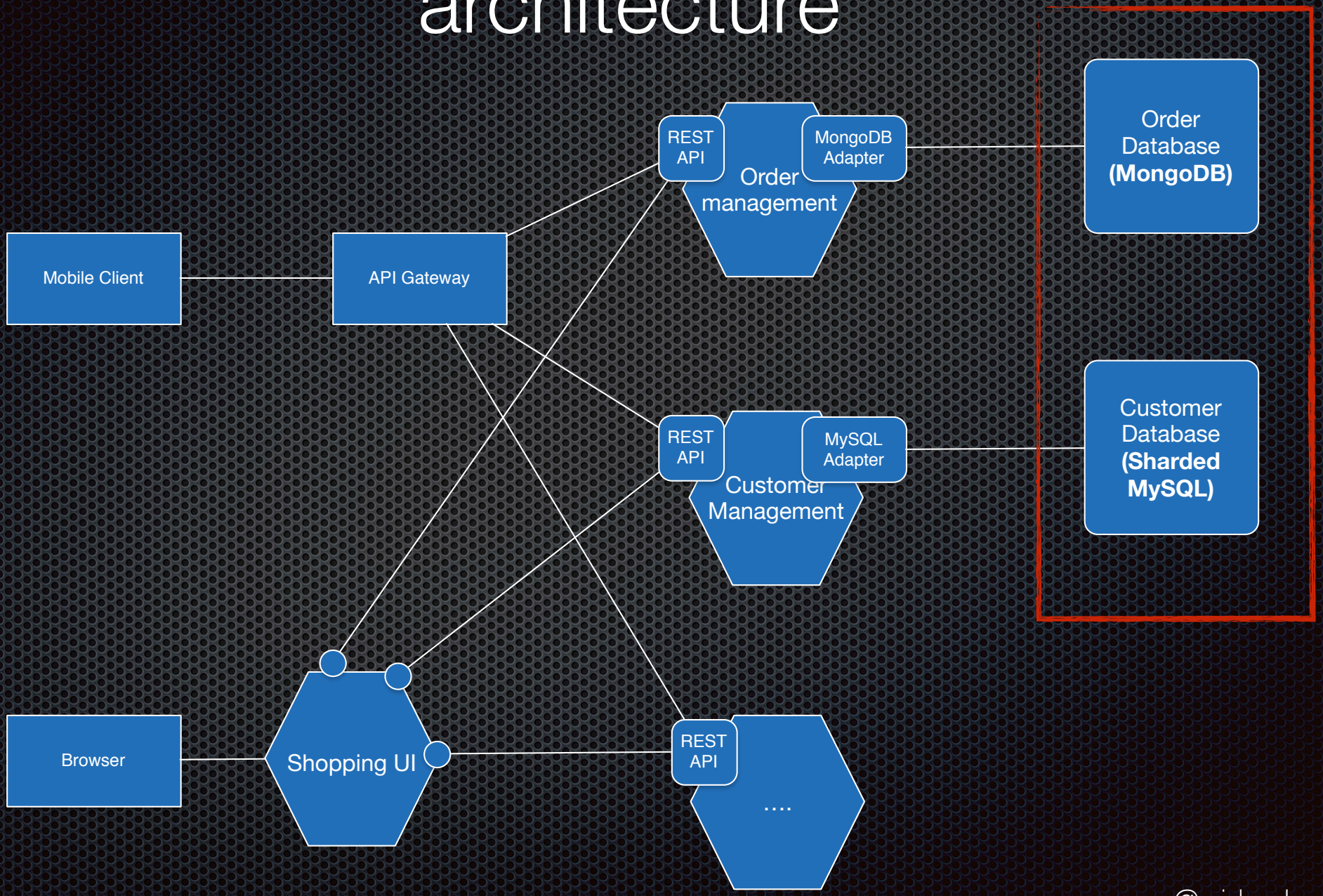
test

deploy

scale

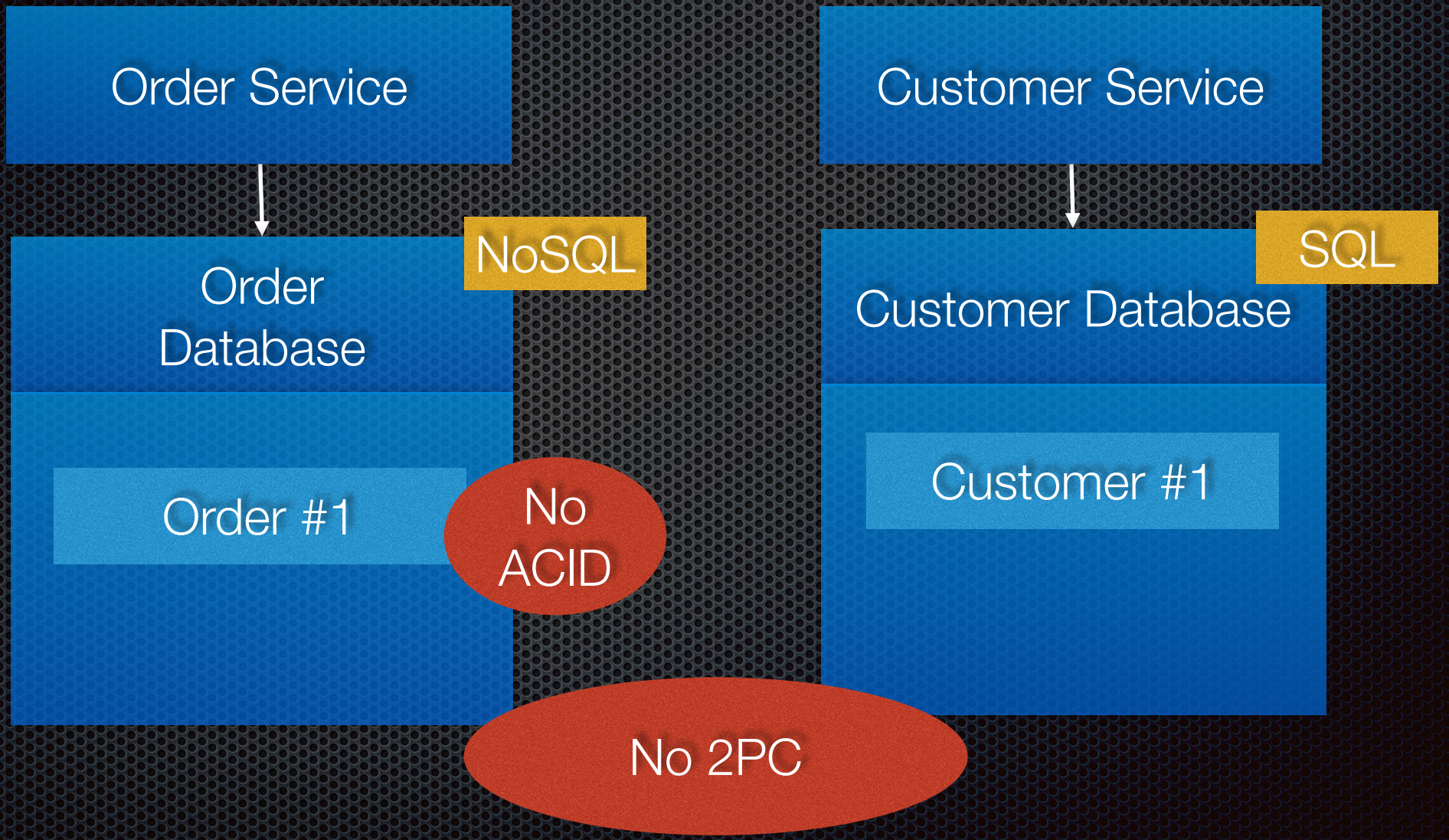

But that leads* to
monolithic hell
For large and/or complex applications...

Today: use a microservice, polyglot architecture

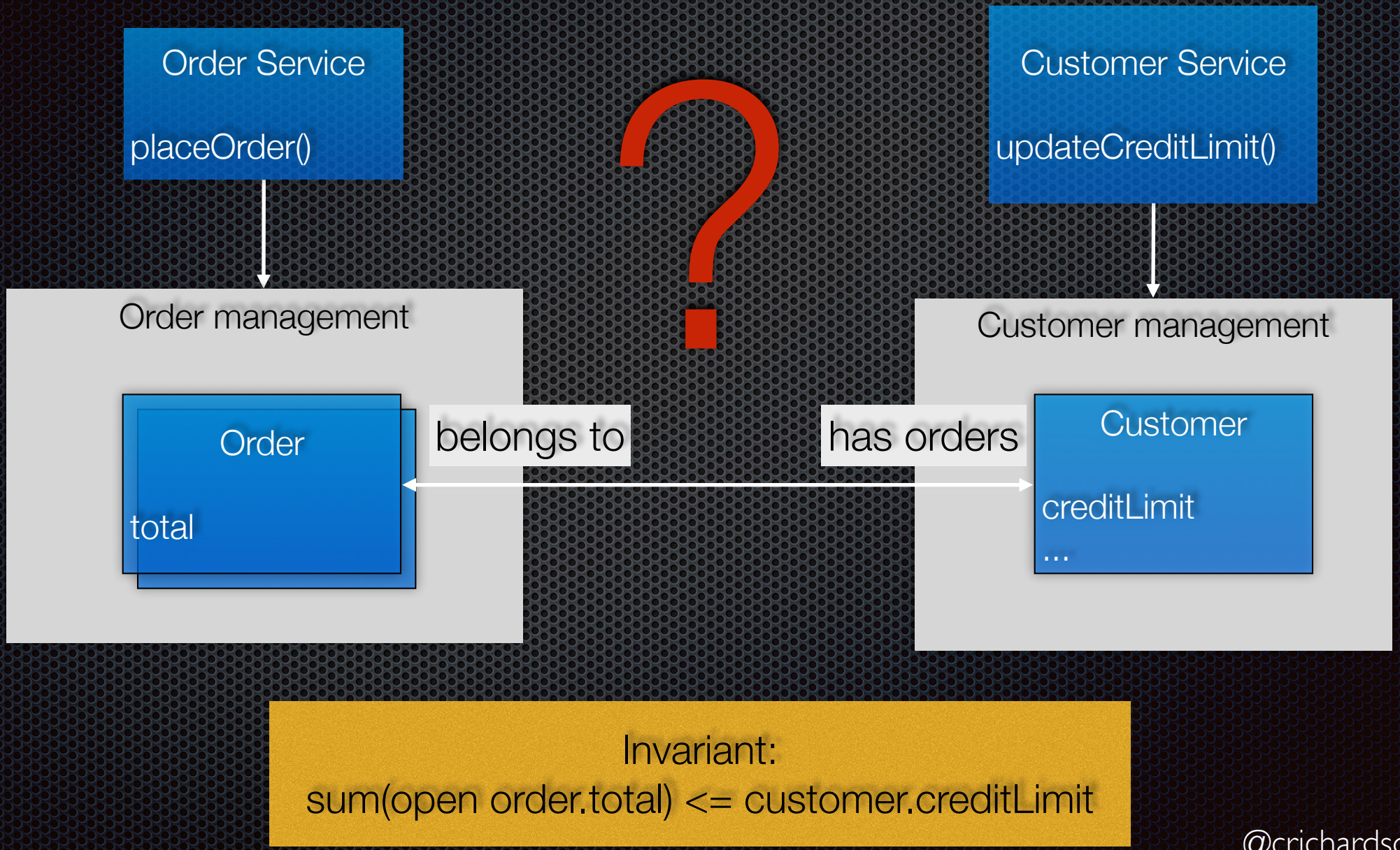


But now we have distributed data management problems

Example: placing an order



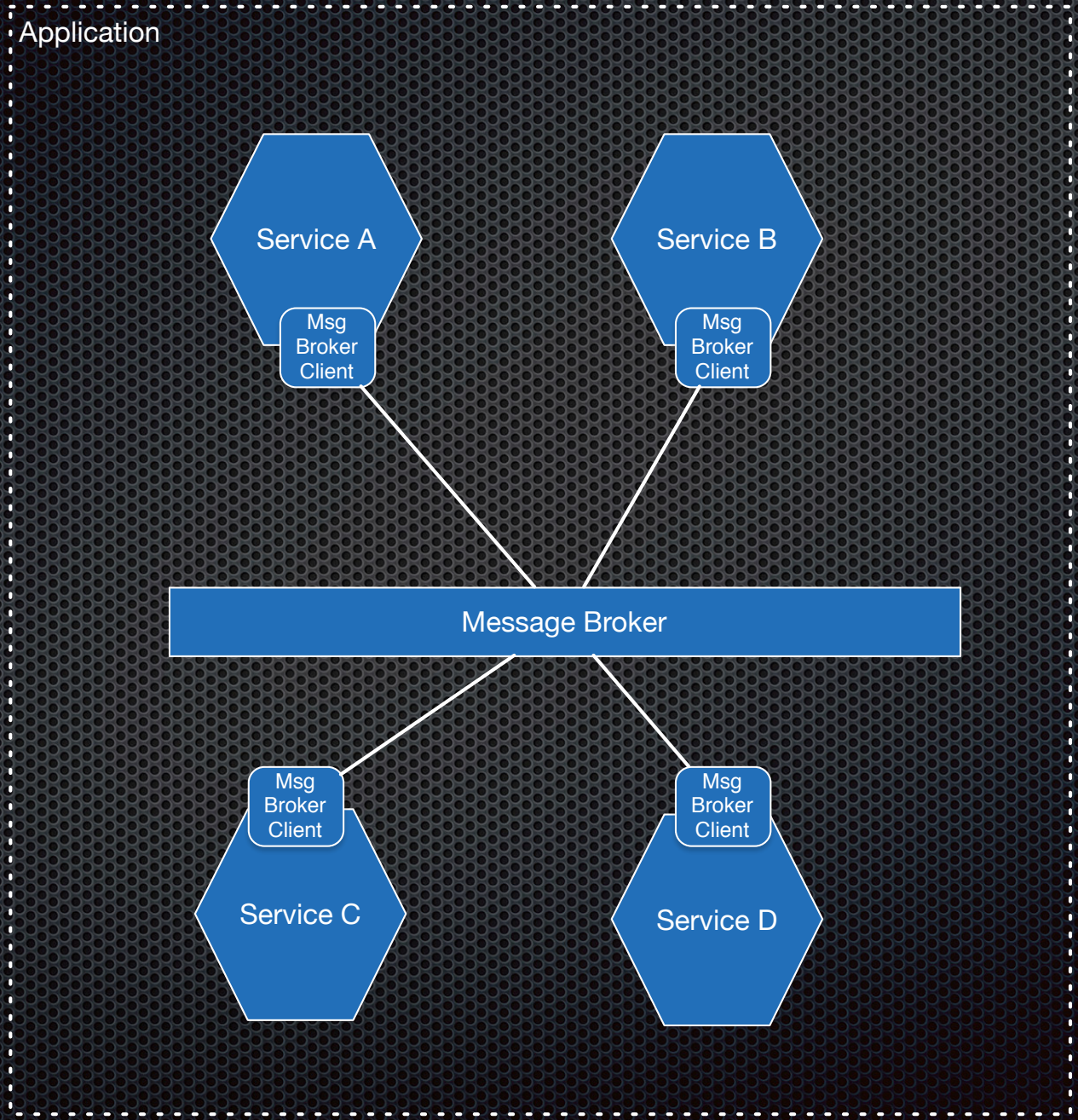
How to maintain invariants?



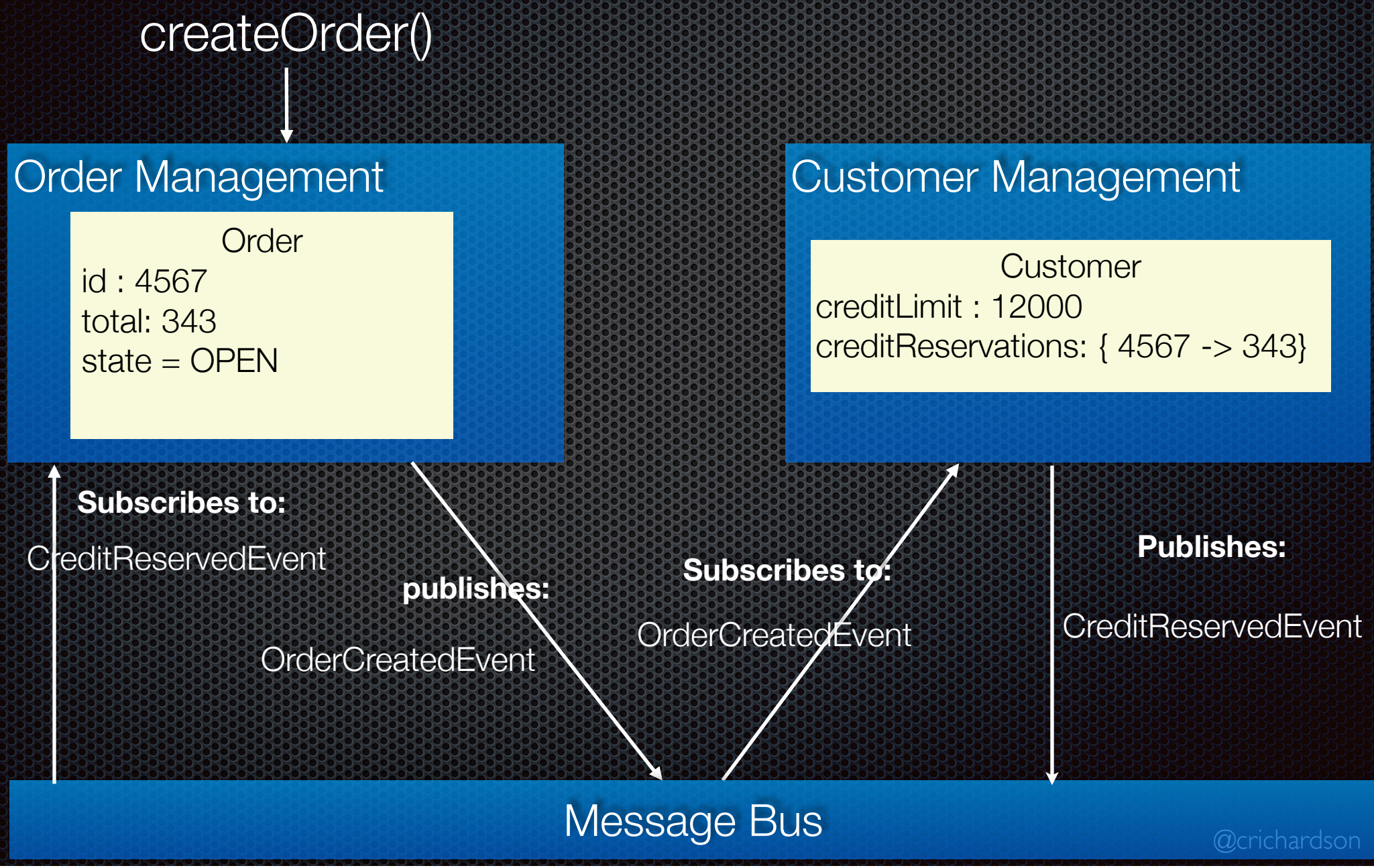
Use an event-driven architecture

- 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

Event-driven application architecture



Eventually consistent credit checking

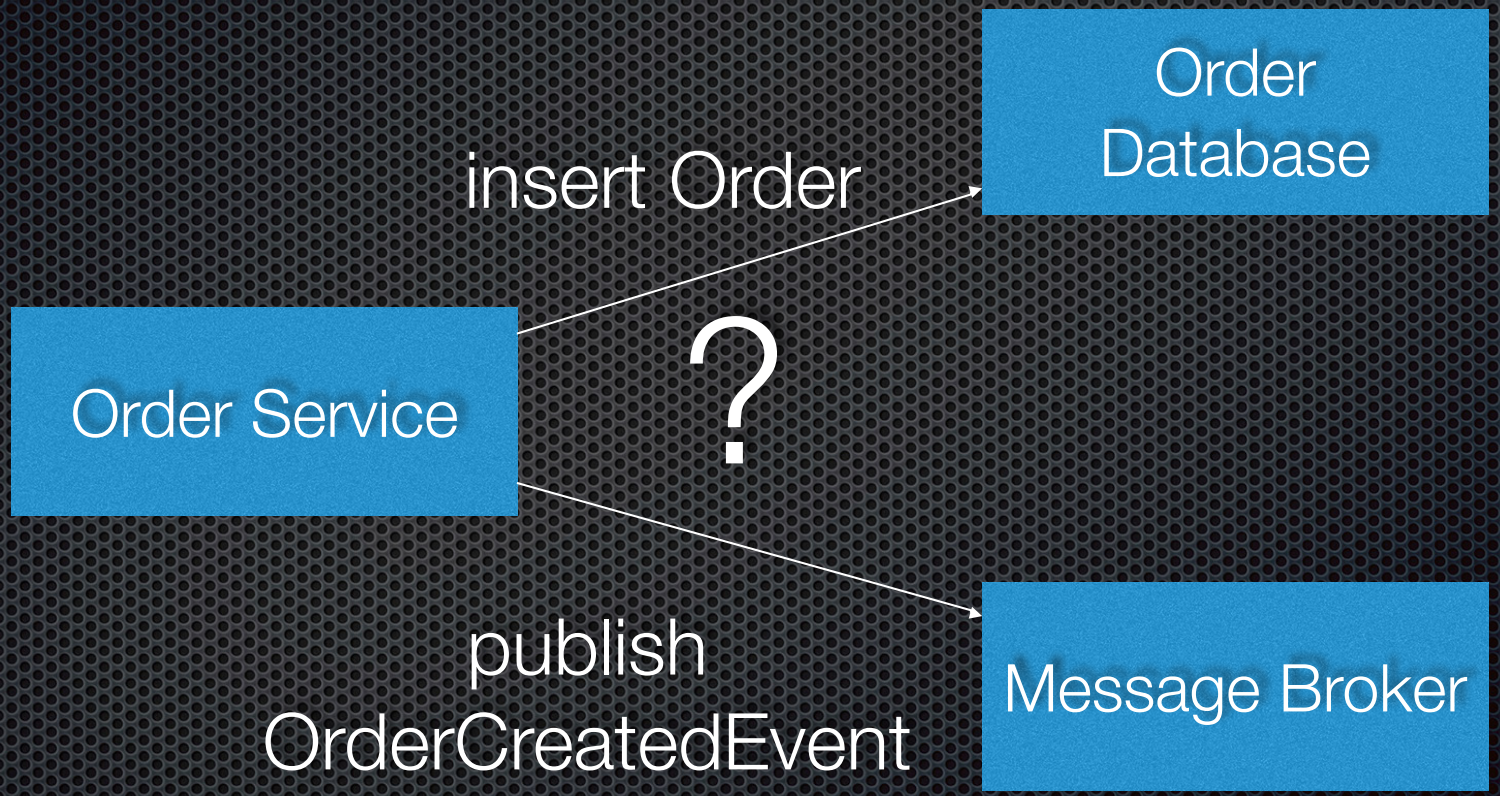


Now there are two problems to solve....

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

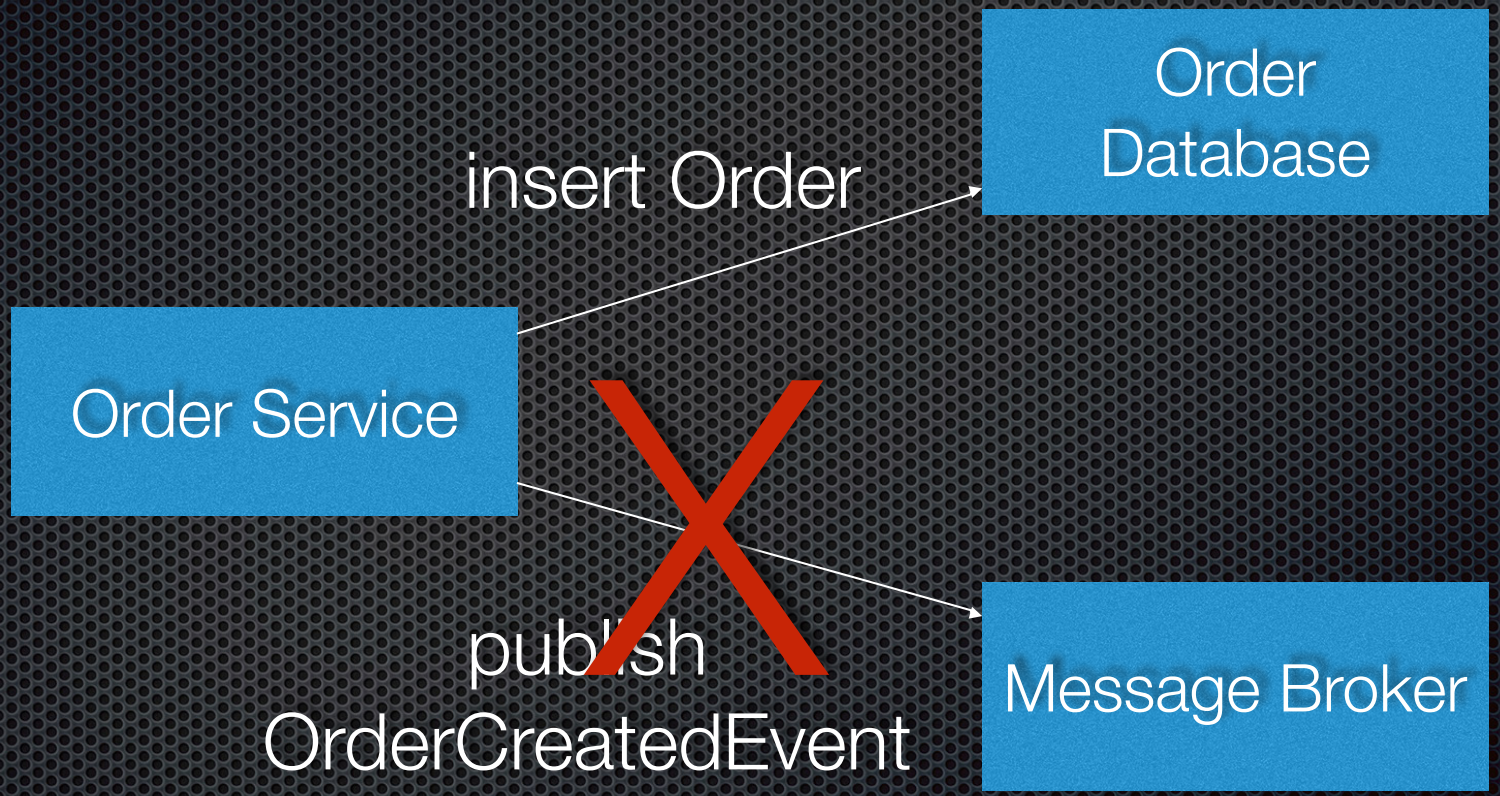
More on that later....

Problem #2: How to atomically update database and publish an event



dual write problem

Failure = inconsistent system

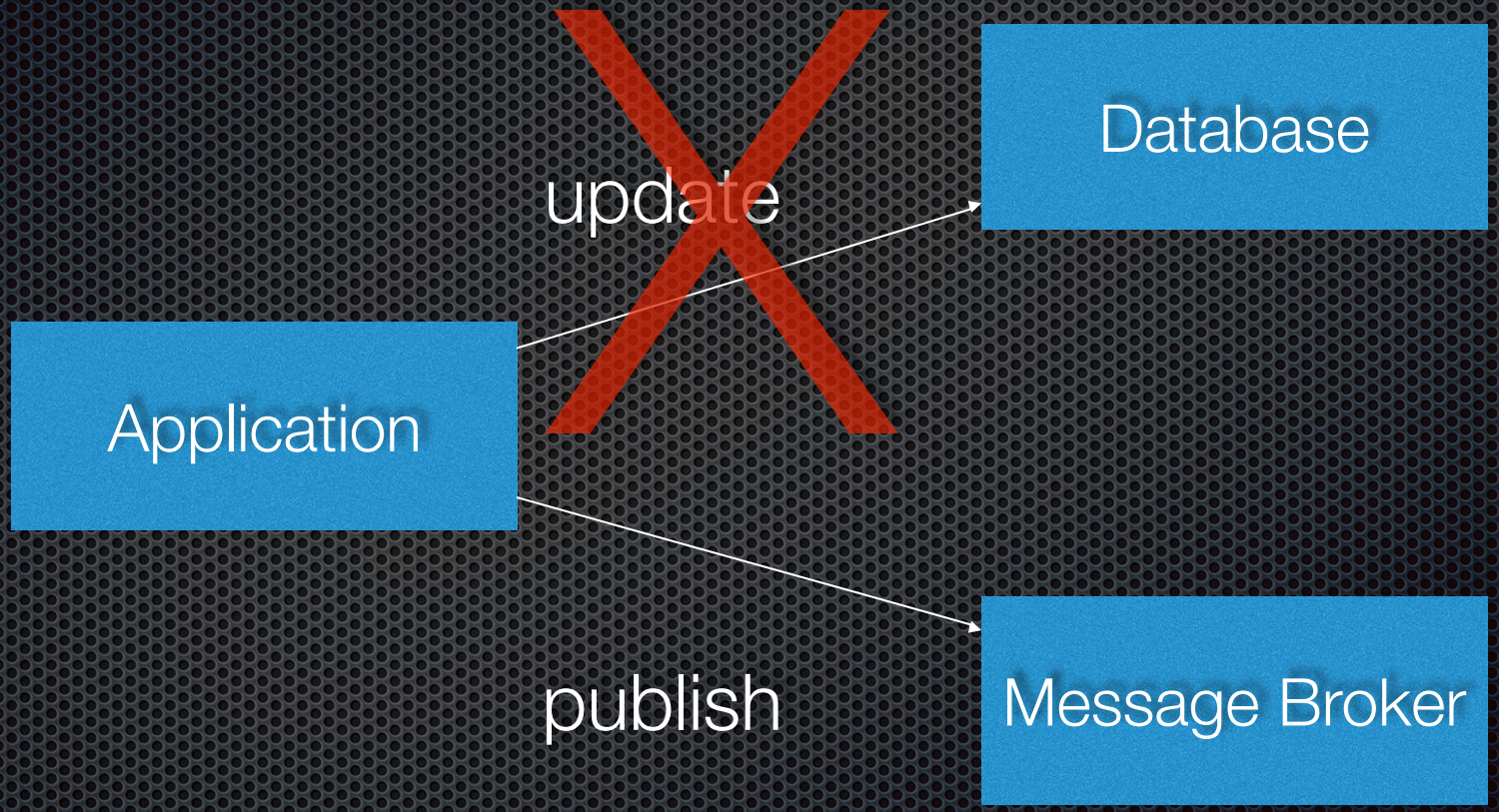



Two-phase commit
@crichardson

Agenda

- Events on the outside
- Events on the inside
- Events at the core with event sourcing
- Designing event-centric domain model

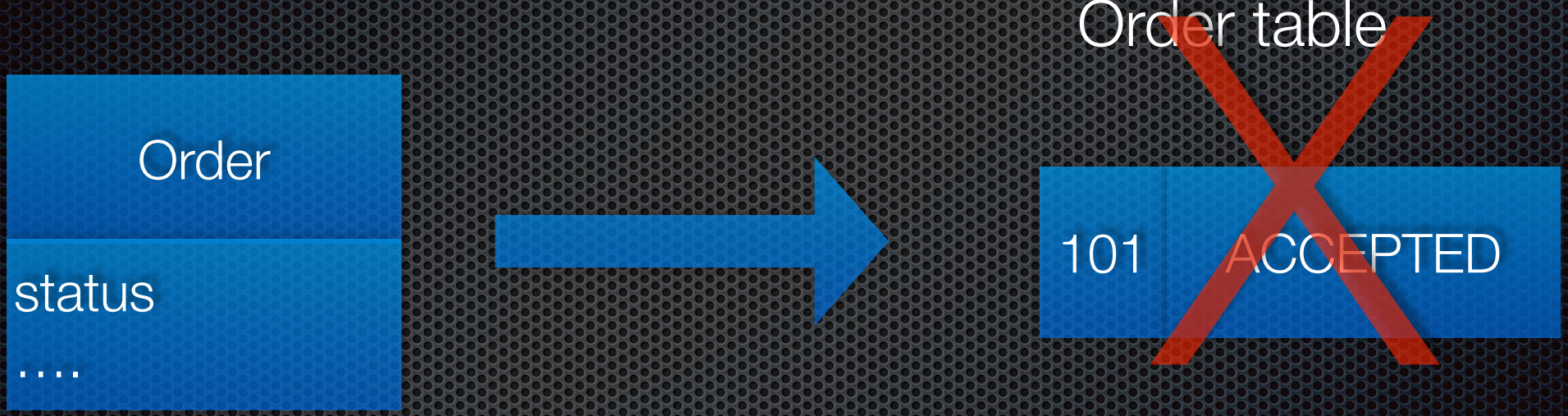
Just publish events



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

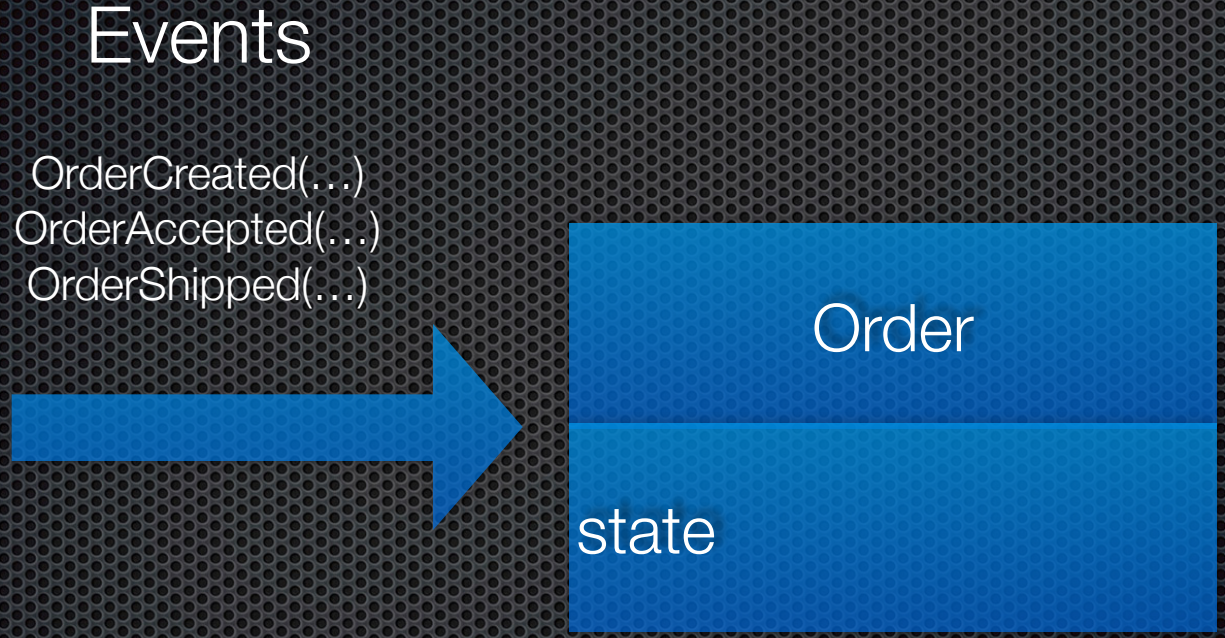


Persists events NOT current state

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

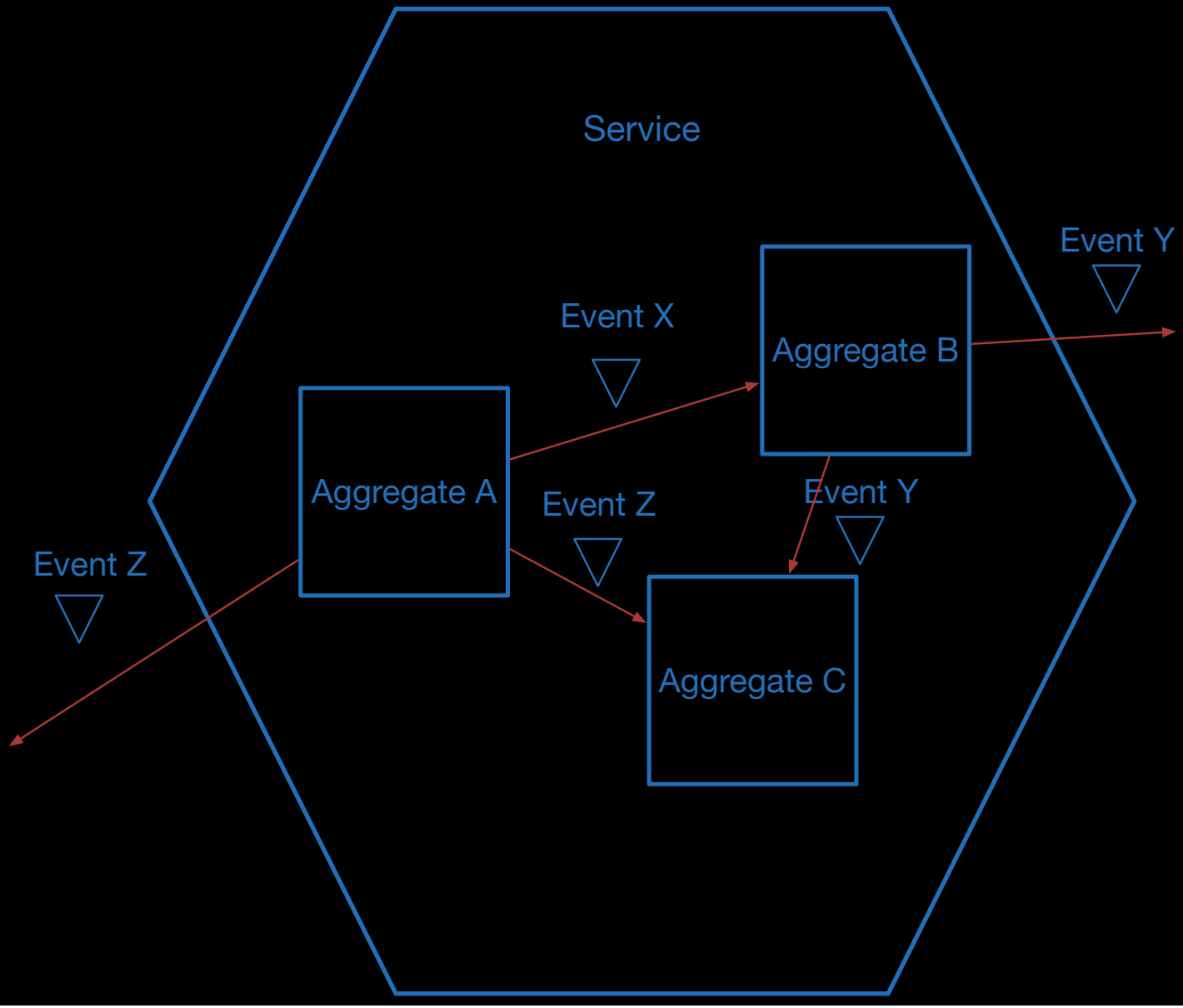


Periodically snapshot to avoid loading all events

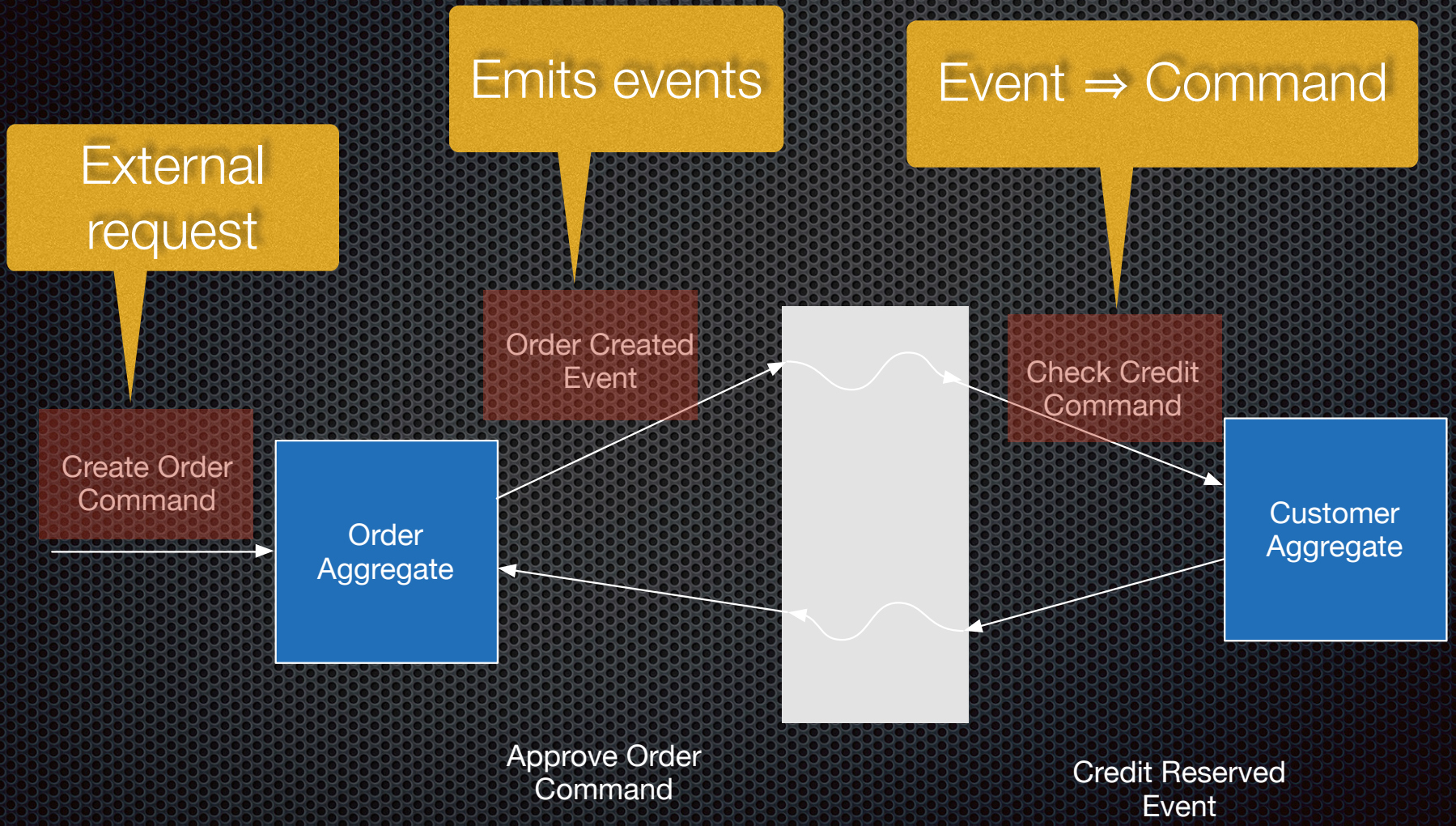
The present is a fold over history

currentState = foldl(applyEvent, initialState, events)

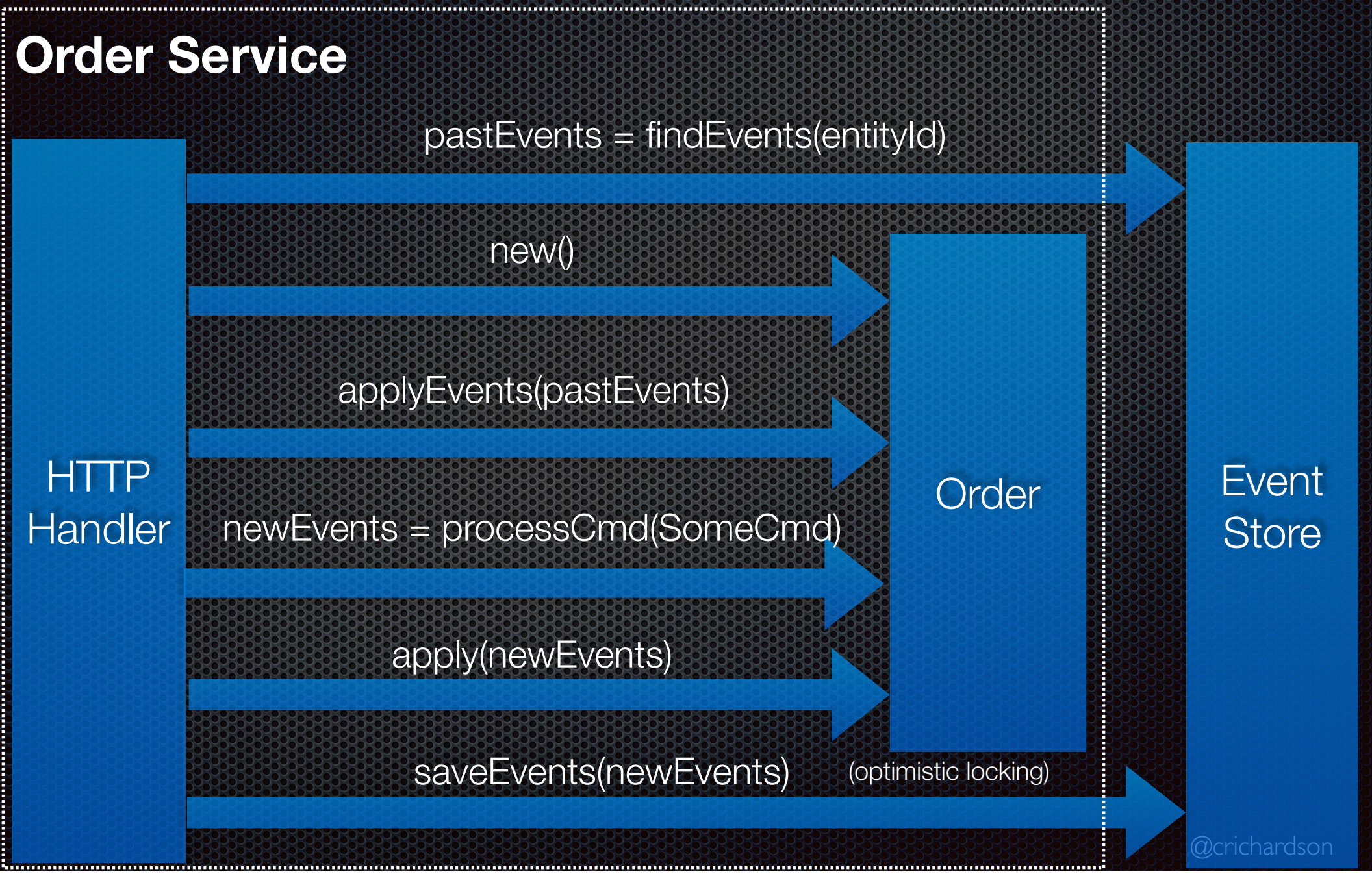
Events at the core



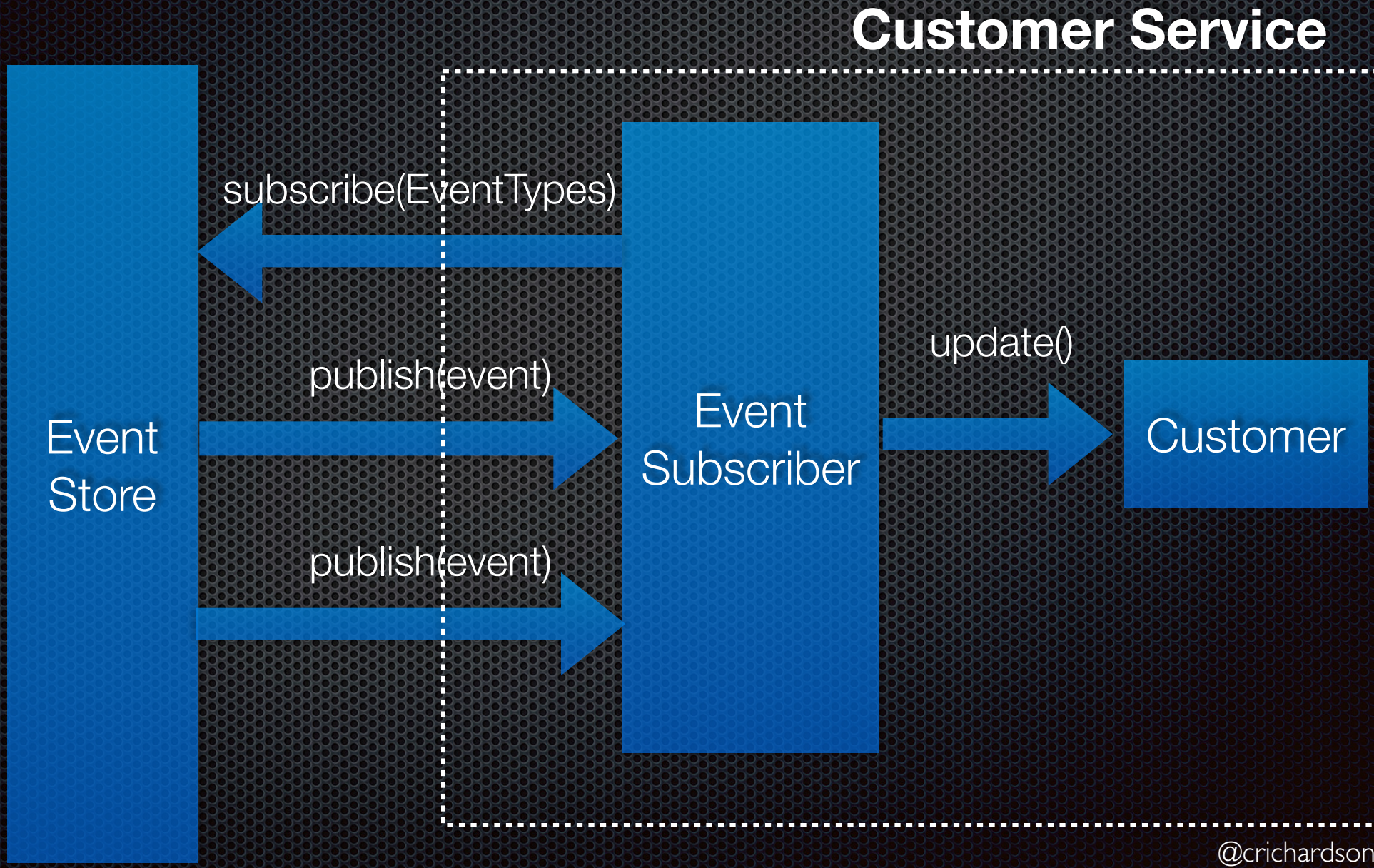
Domain logic = event-driven aggregates



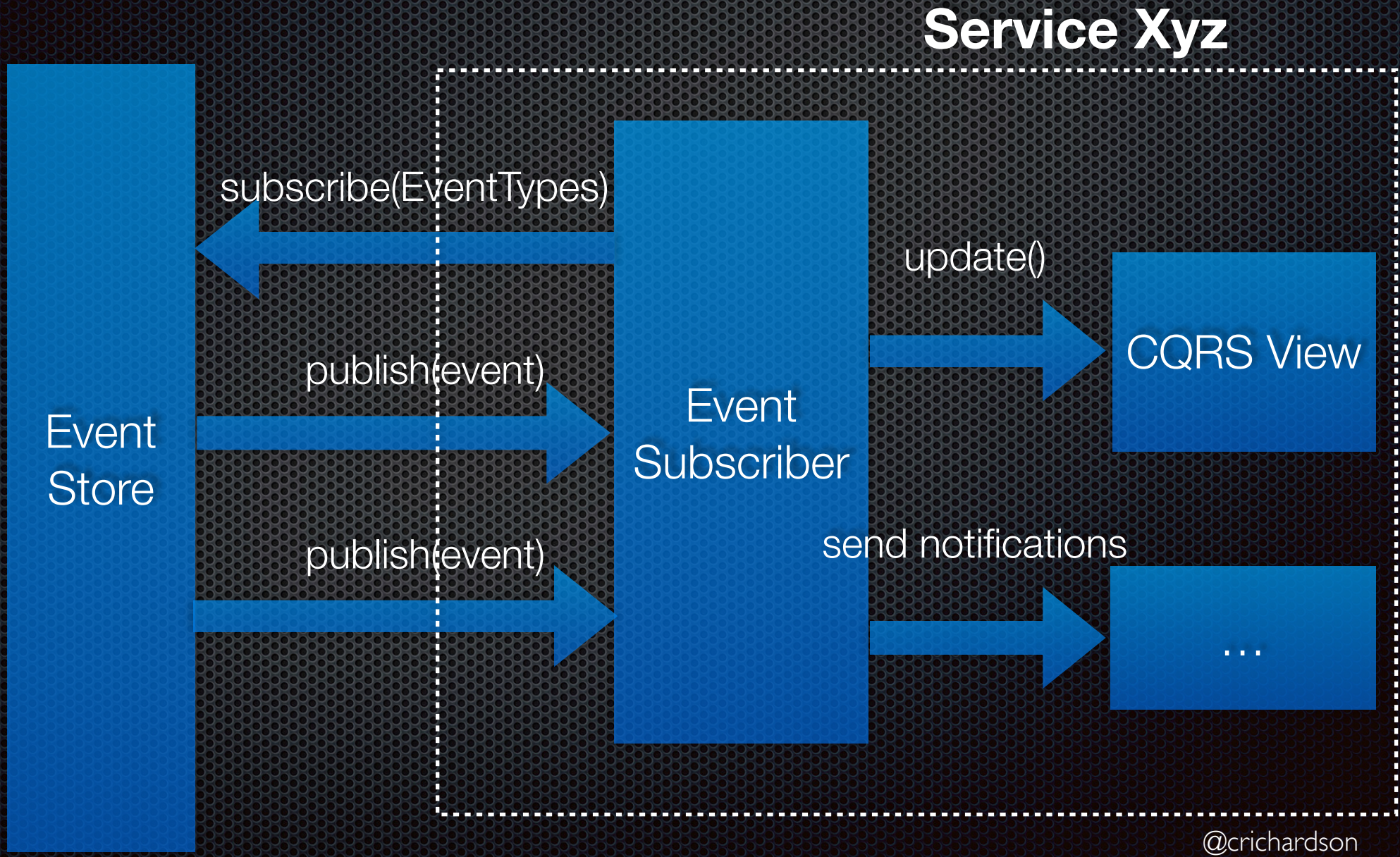
Request handling in an event sourced application



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

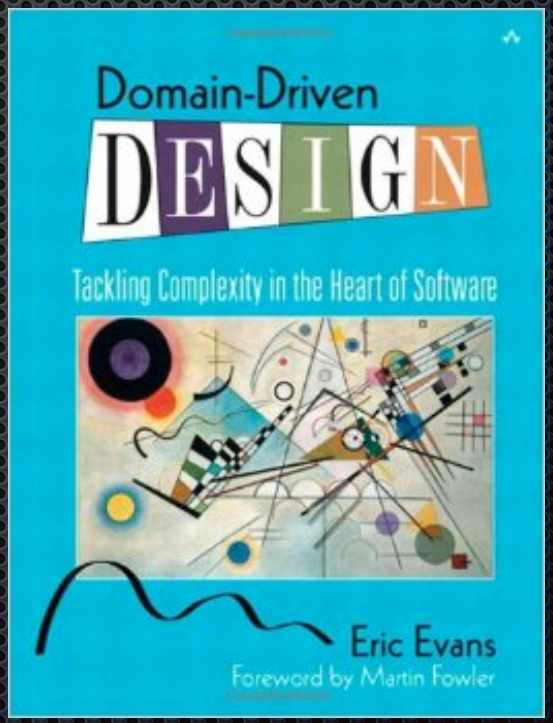
- Querying the event store can be challenging
- Some queries might be complex/inefficient, e.g. accounts with a balance > X
- Event store might only support lookup of events by entity id
- Must use Command Query Responsibility Segregation (CQRS) to handle queries ⇒ application must handle eventually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

Agenda

- Events on the outside
- Events on the inside
- Events at the core with event sourcing
- Designing event-centric domain model

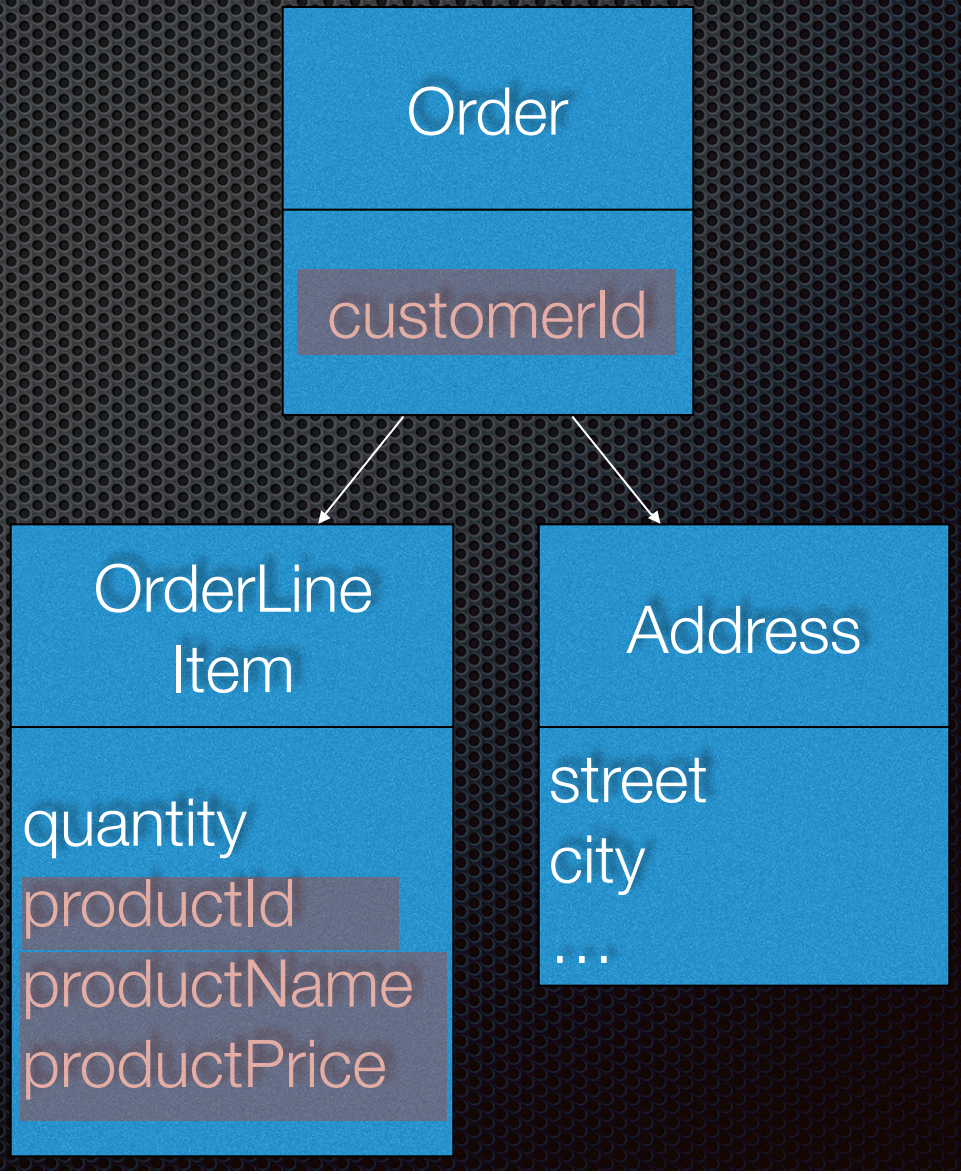
Use the familiar building blocks of DDD

- Entity
- Value object
- Services
- Repositories
- Aggregates ⟸ **essential**

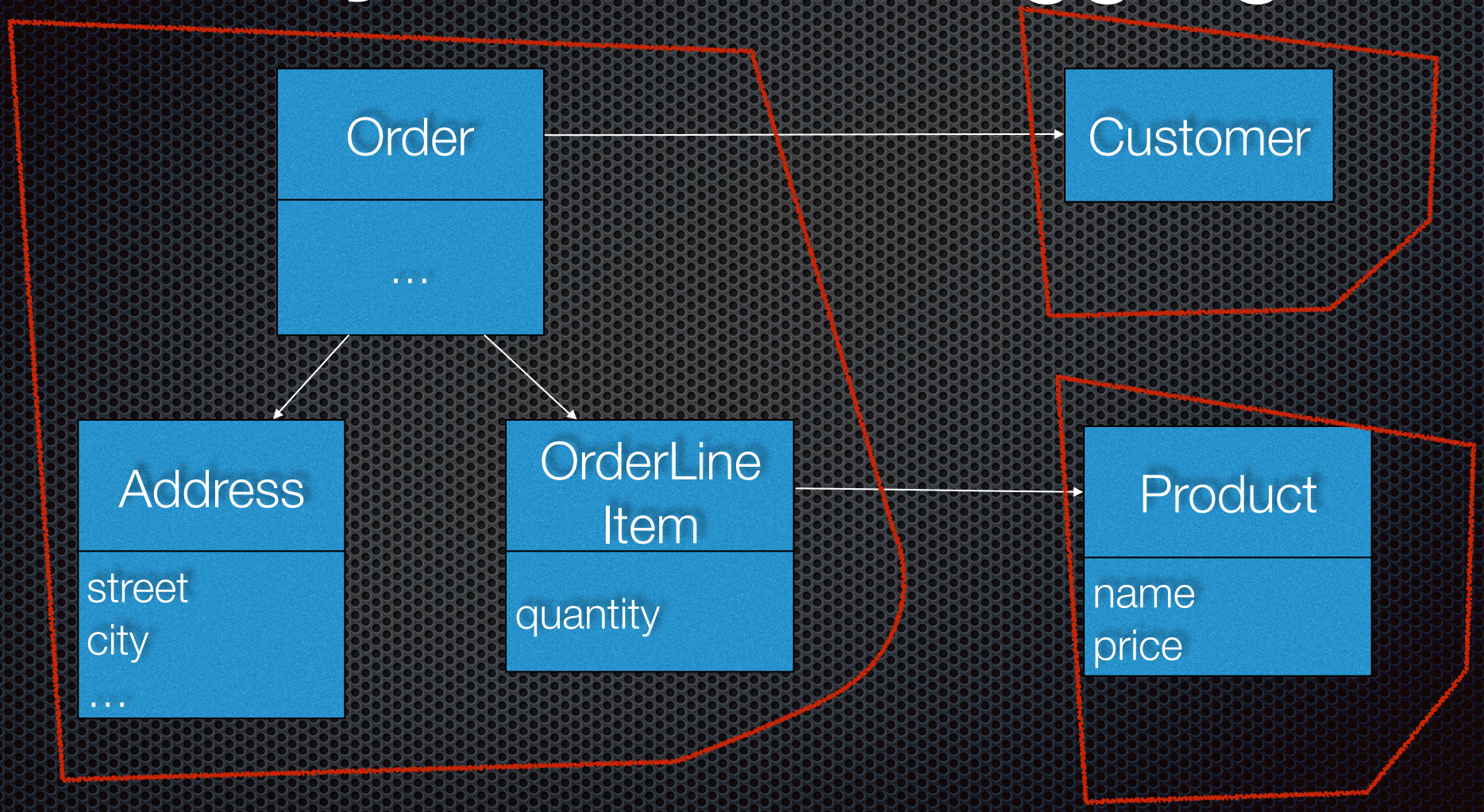


About Aggregates

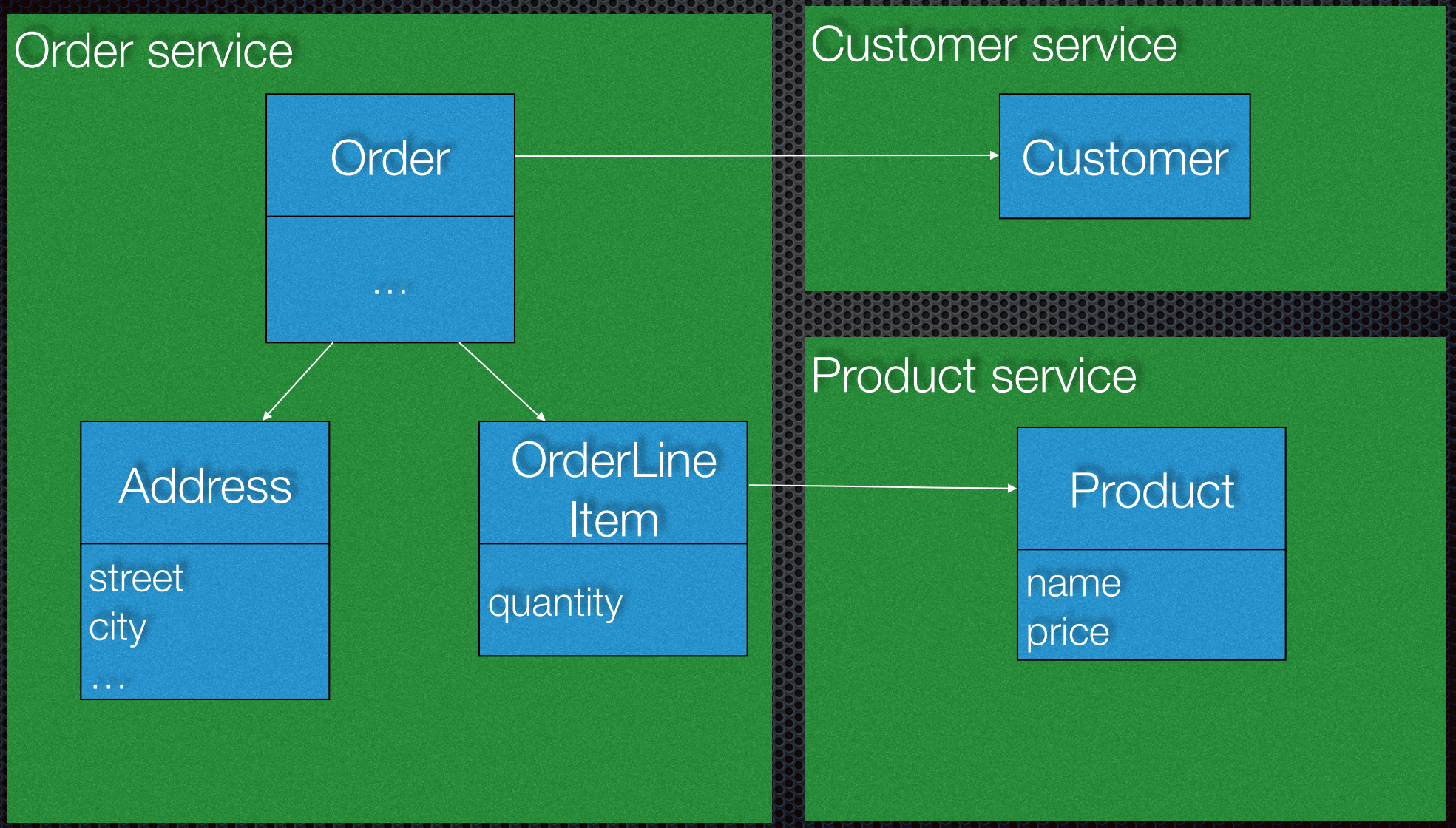
- Graph consisting of a root entity and one or more other entities and value objects
- Each core business entity = Aggregate: e.g. customer, Account, Order, Product, ….
- Reference other aggregate roots via primary key
- Often contains partial copy of other aggregates' data



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



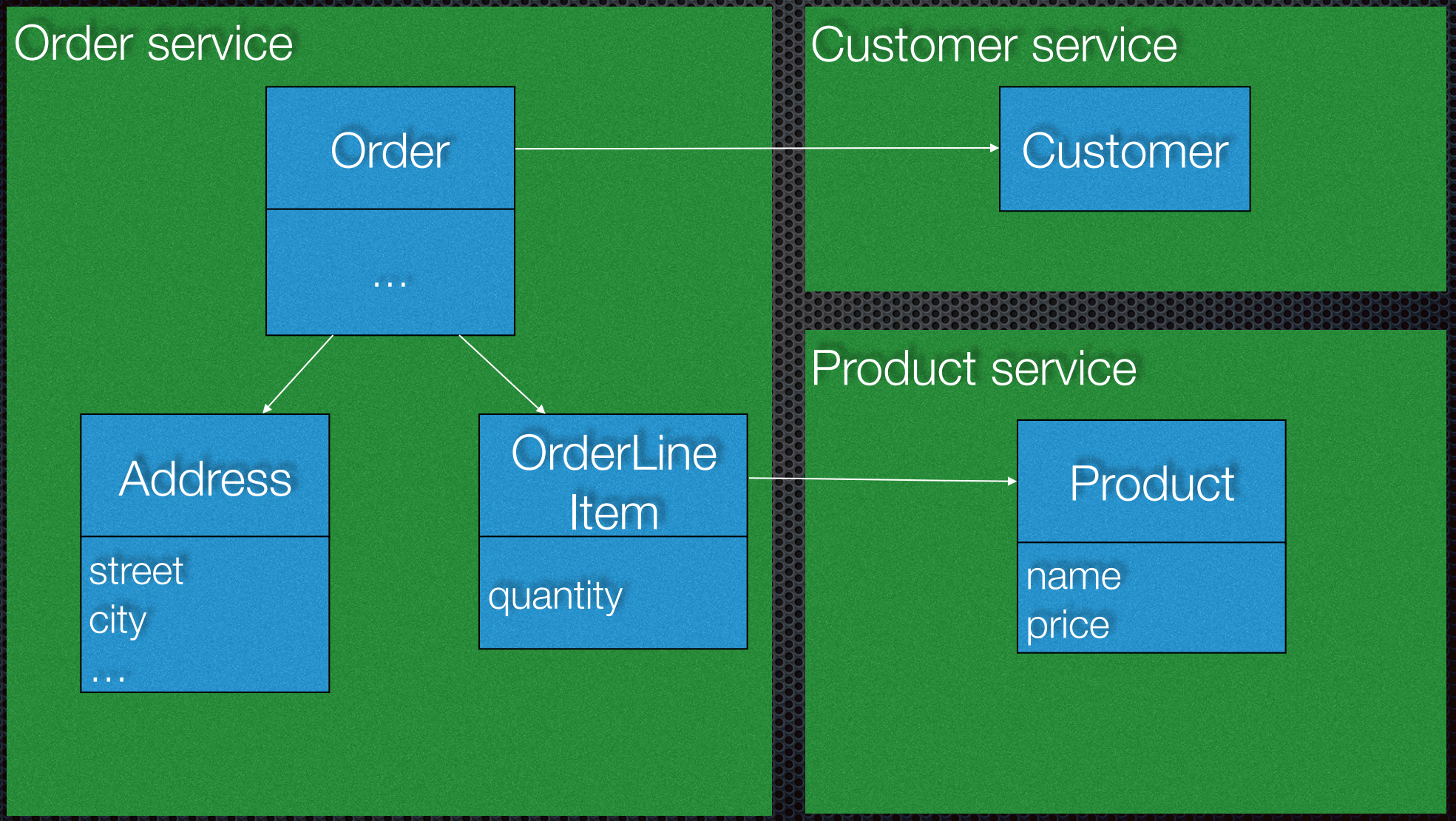
Easily partition into microservices



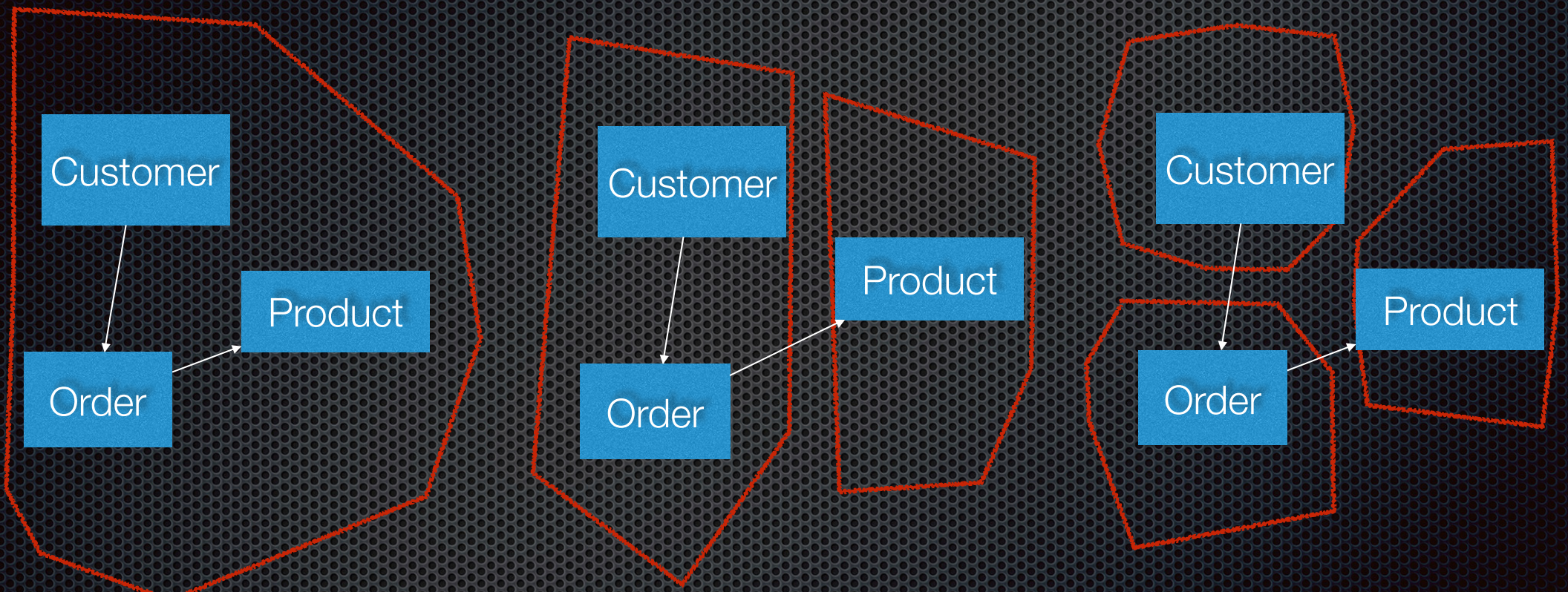
Transaction = processing one command by one aggregate

- No opportunity to update multiple aggregates within a transaction ⇒ event driven eventual consistency between aggregates
- If an update must be atomic (i.e. no compensating transaction) then it must be handled by a single aggregate
- **Therefore,** aggregate granularity is important

Transaction scope = service



Aggregate granularity

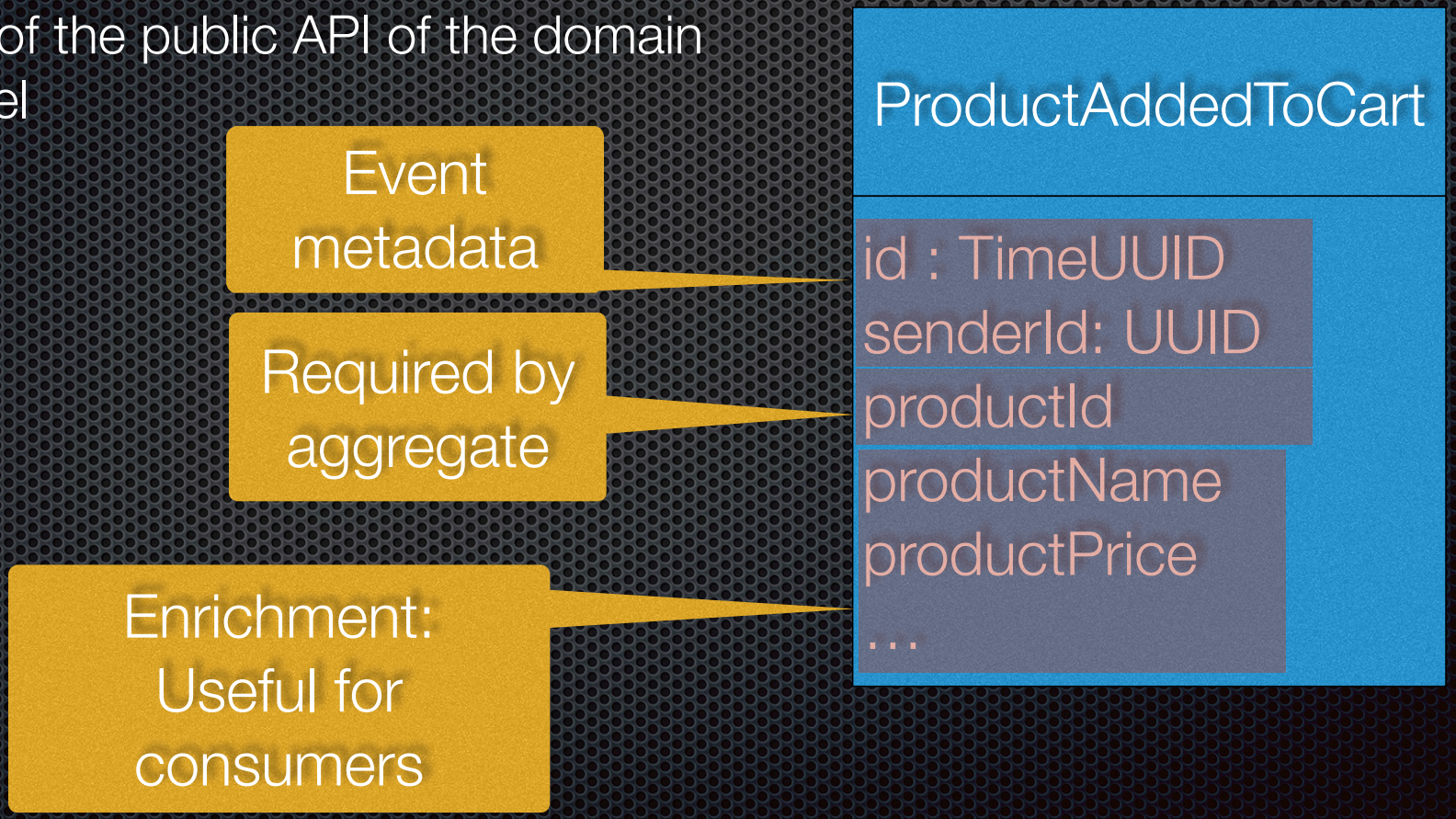


Consistency

Scalability/
User experience

Designing domain events

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



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

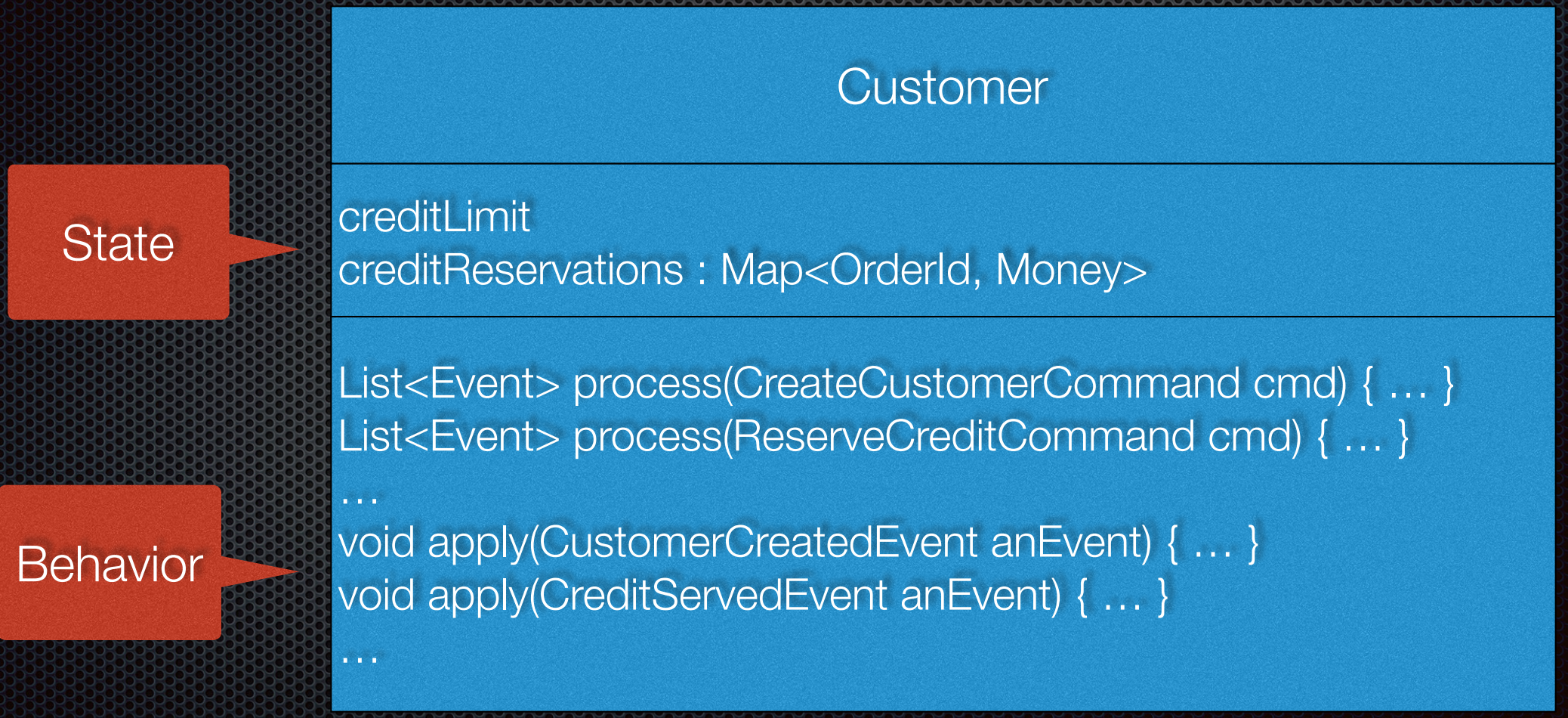
```
public class CreateCustomerCommand implements CustomerCommand {
  private final String name;
  private final Money creditLimit;

  public CreateCustomerCommand(String name, Money creditLimit) {...}

  public Money getCreditLimit() { return creditLimit; }

  public String getName() { return name; }
}
```

Various programming models

- “Traditional Java” mutable object-oriented domain objects
  - https://github.com/cer/event-sourcing-examples/tree/master/java-spring
- Functional Scala with immutable domain objects
  - https://github.com/cer/event-sourcing-using-scala-typeclasses
- Hybrid OO/Functional Scala with immutable domain objects
  - https://github.com/cer/event-sourcing-examples/tree/master/scala-spring

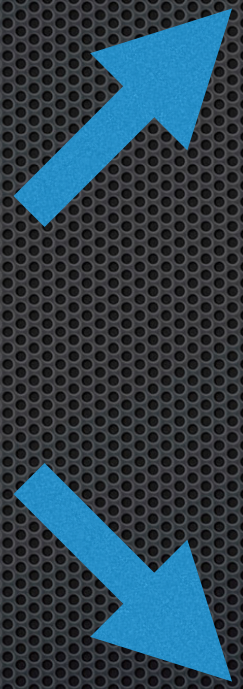
Hybrid OO/FP domain objects

OO = State + Behavior



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 … new CreditReservedCredit();
  }
```

```
 public void apply(
       CreditReservedCredit 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
  }

}
```

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 Order aggregate
2. Processes command
3. Applies events
4. Persists events

Event handling in Customers

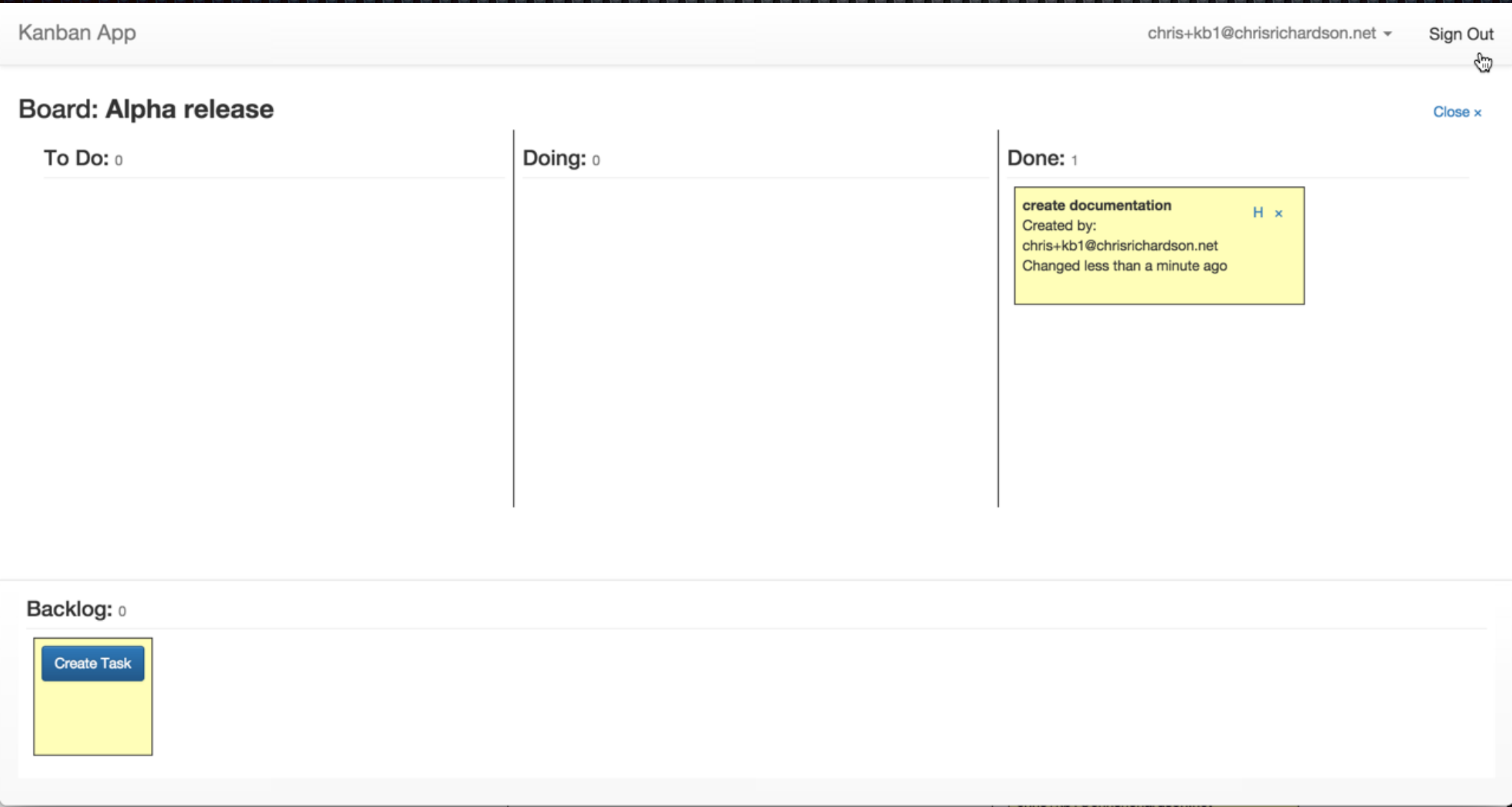
Triggers BeanPostProcessor

Durable subscription name

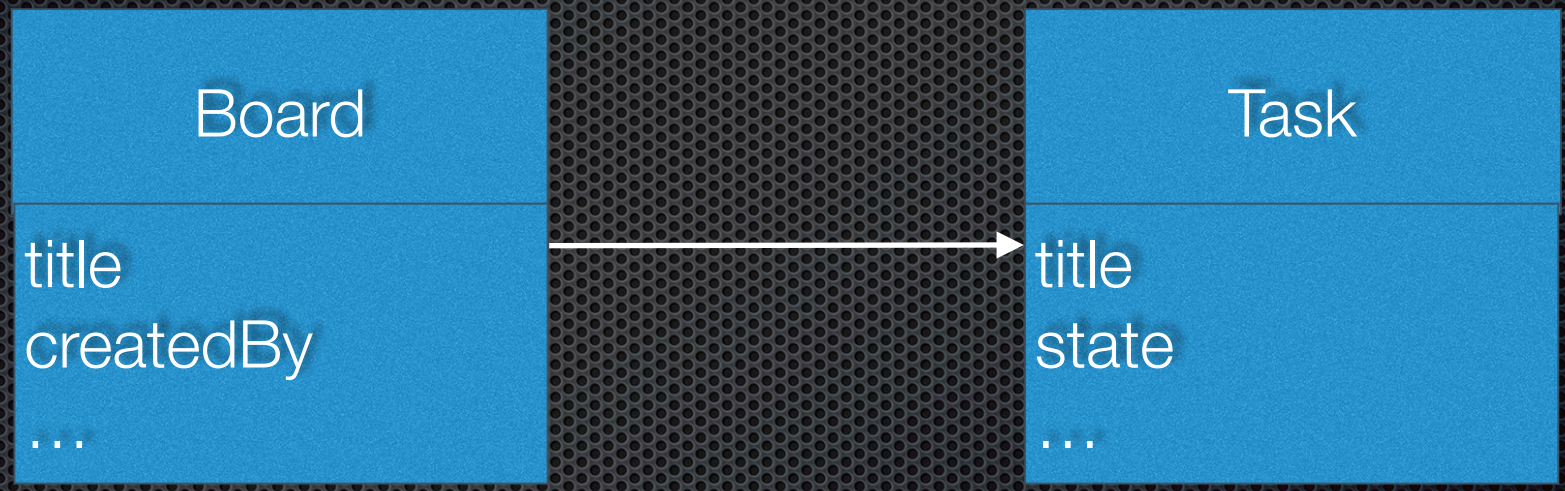
```
@EventSubscriber(id="customerWorkflow")
public class CustomerWorkflow {

  @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

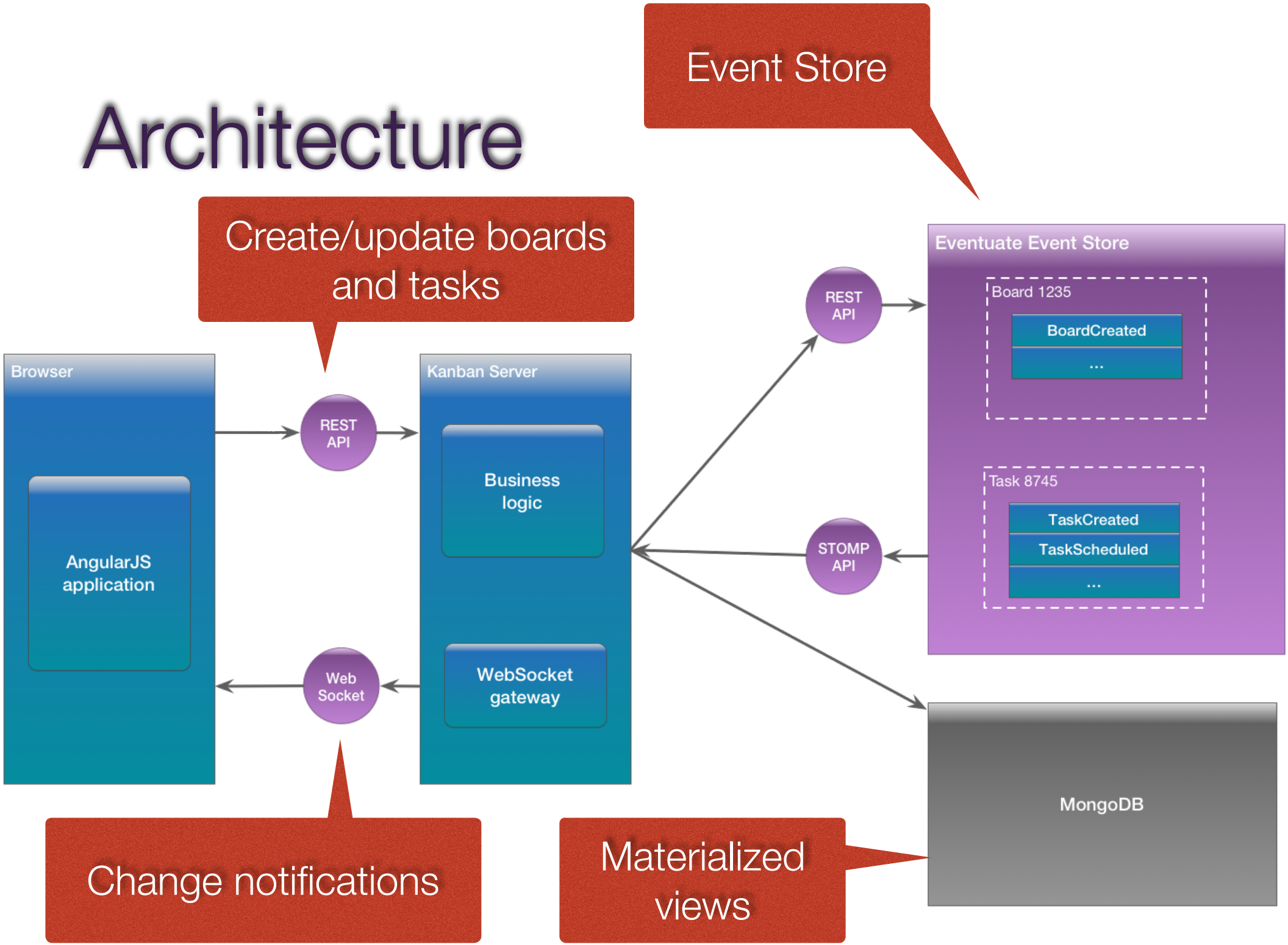
Kanban board example

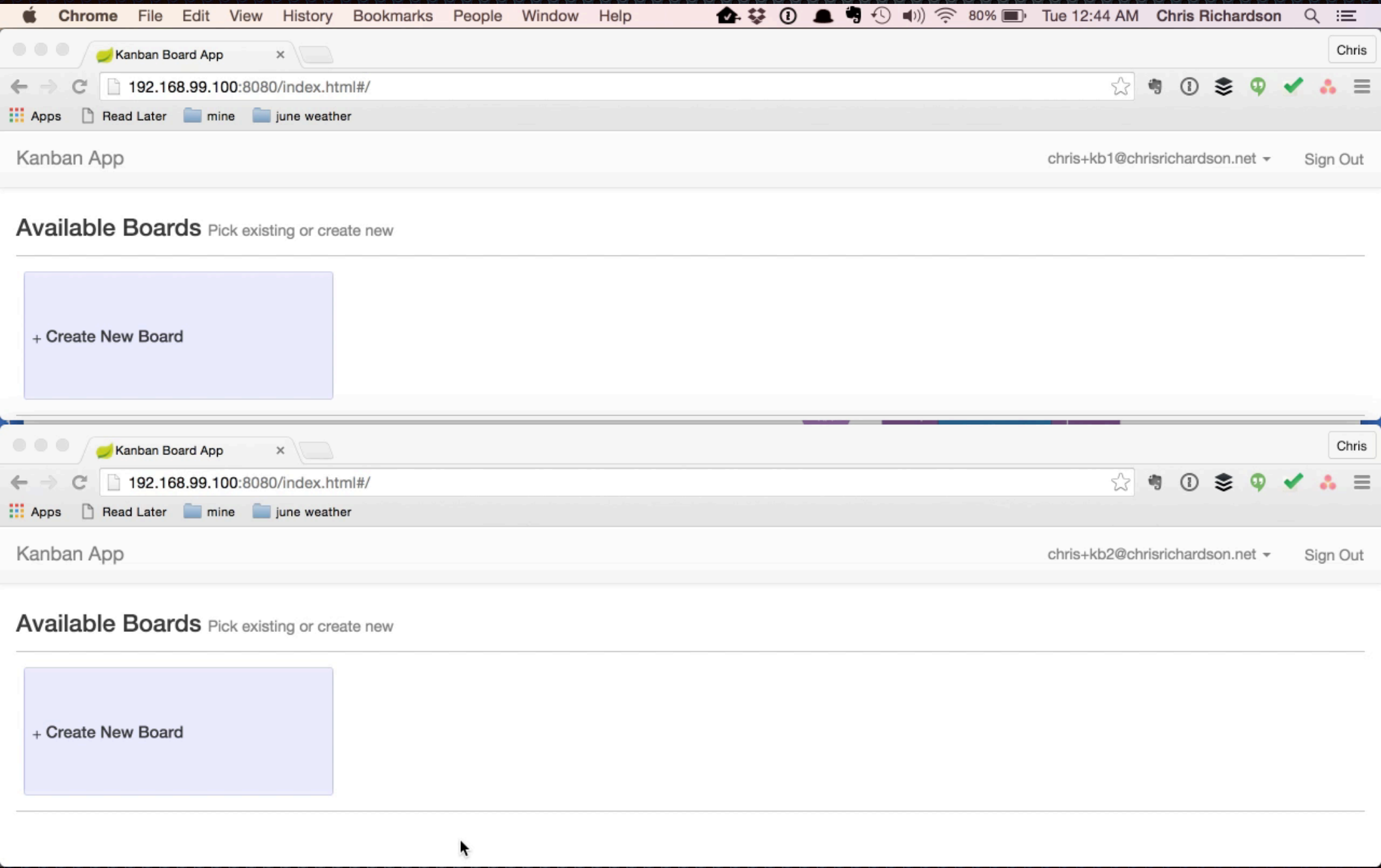


Kanban domain model



Architecture




Chrome File Edit View History Bookmarks People Window Help 80% Tue 12:44 AM Chris Richardson
Kanban Board App
Chris
192.168.99.100:8080/index.html#/
Apps Read Later mine june weather
Kanban App
chris+kb1@chrisrichardson.net
Sign Out
Available Boards Pick existing or create new
+ Create New Board
Kanban Board App
Chris
192.168.99.100:8080/index.html#/
Apps Read Later mine june weather
Kanban App
chris+kb2@chrisrichardson.net
Sign Out
Available Boards Pick existing or create new
+ Create New Board

Summary

- Events are a central to modern applications
- Events integrate applications
- Events maintain data consistency in a microservices architecture

⇒

- Build events into the core of your application using event sourcing


@crichardson chris@chrisrichardson.net
Questions?
http://plainoldobjects.com
http://microservices.io
http://eventuate.io