

# 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

 @crichtson

chris@chrisrichardson.net

<http://microservices.io>

<http://eventuate.io>

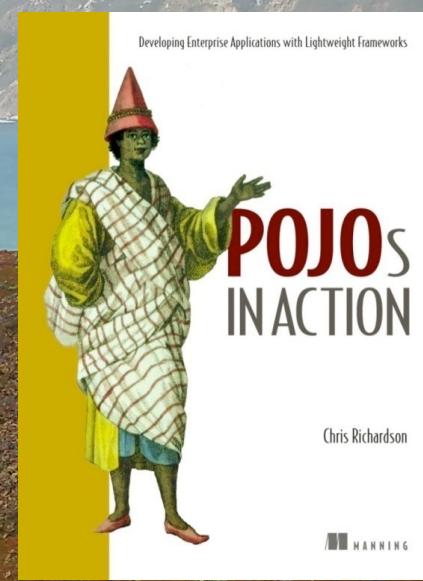
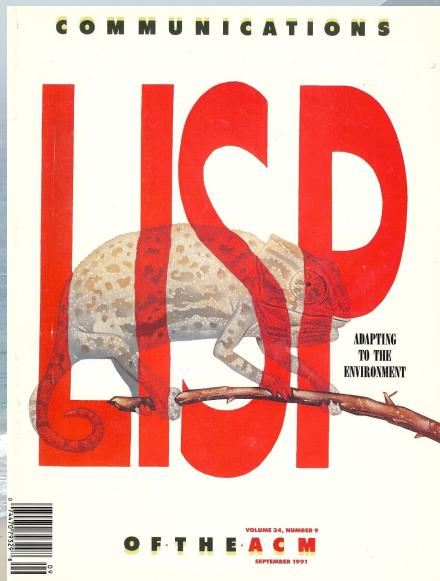
<http://plainoldobjects.com>

Presentation goal

Events play a key role in  
applications

Event sourcing enables  
the event-driven enterprise

# About Chris

A screenshot of the Cloud Foundry website. The header includes the Cloud Foundry logo and links for 'Email', 'Sign Up', 'Forgot password?', and 'SIGN IN'. Below the header, there are navigation links for 'HOW WE HELP', 'FEATURES', 'INFORMATION', 'BLOG', and 'CONTACT US'. A system alert message at the top states: 'SYSTEM ALERT. PLEASE READ: Cloud Foundry will be moving to a new URL. [More](#)'. The main content area features the heading 'The Enterprise Java Cloud' and a bulleted list: 'Real Java Applications Deployed in Minutes', 'Built for Spring and Grails Web Applications', and 'Most Widely Used Technologies Delivered as a Platform'. It also includes a 'SIGN UP' button with a 'NEW' badge and a 'LEARN MORE' button. On the right, there is a 'CLOUDFOUNDRY SPRING SOURCE' logo with a play button icon and the text 'APPLICATION DEMO Deploying Web Applications To Amazon EC2 with Cloud Foundry'.

@crichtson

# About Chris

Consultant and  
trainer focusing on  
microservices

<http://www.chrisrichardson.net/>

@crichtson

# About Chris

Founder of a startup that is  
creating  
a platform that makes it easy for  
application developers to write  
microservices

<http://eventuate.io>



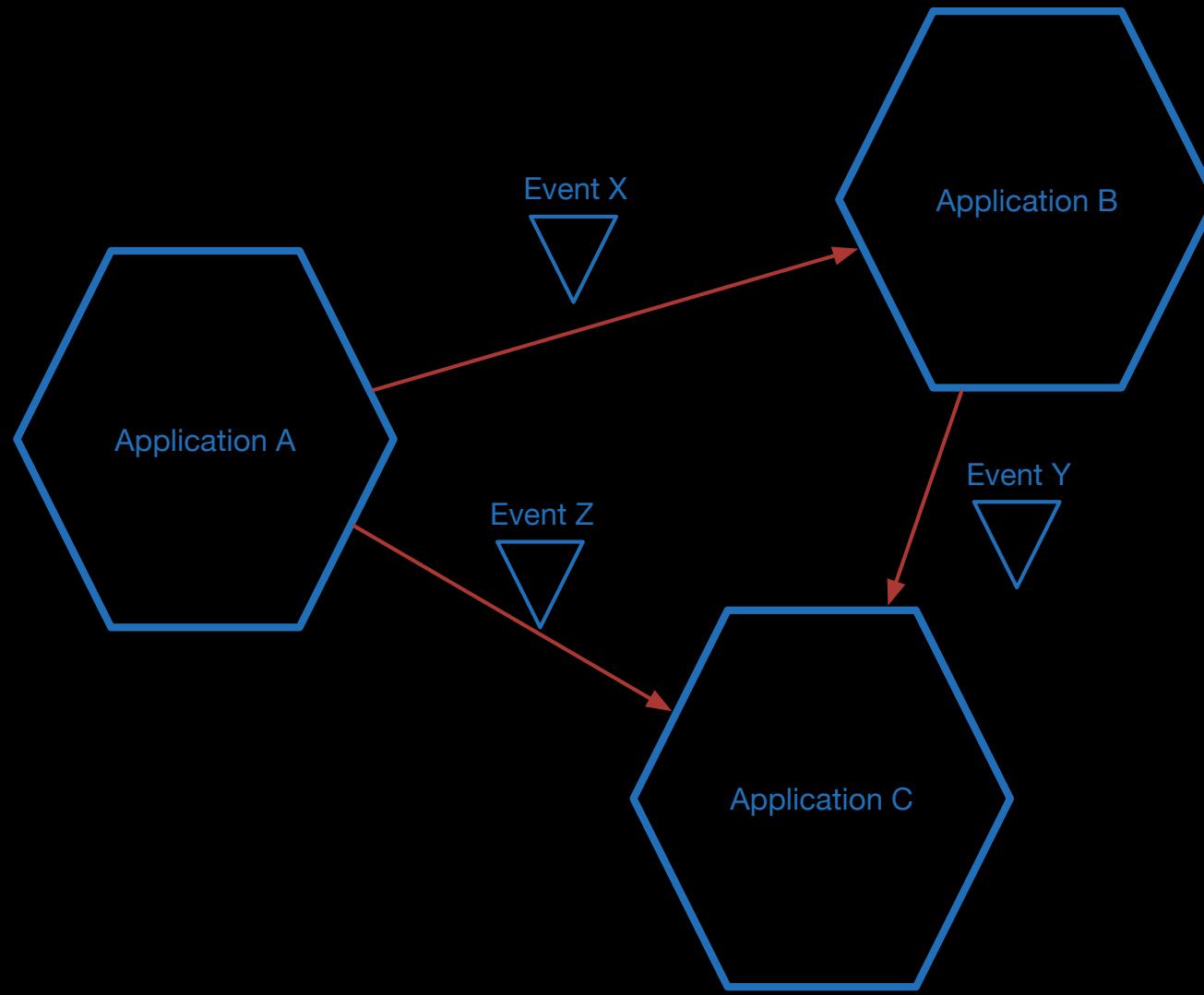
# For more information

- <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?

**event** 

*noun* | \i-'vent\

SAVE  POPULARITY 

Cite! Share G+1 Tweet

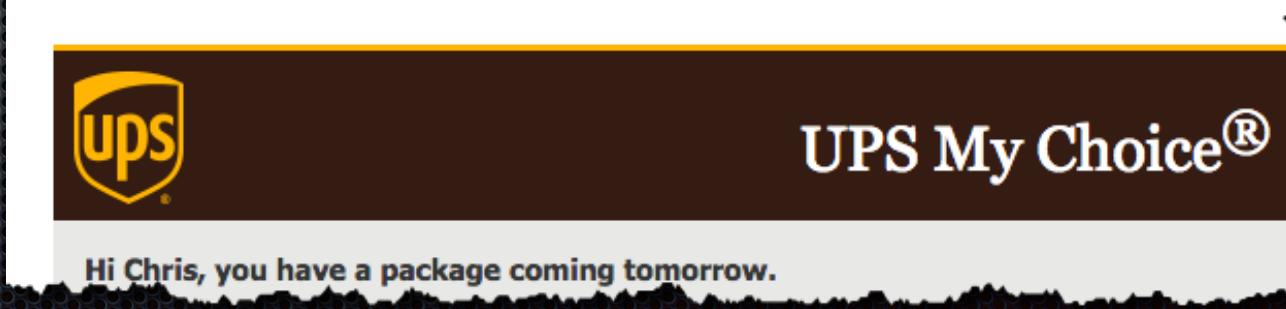
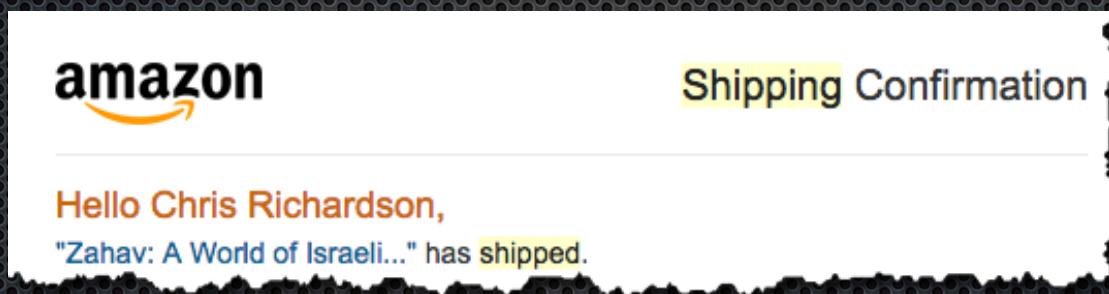
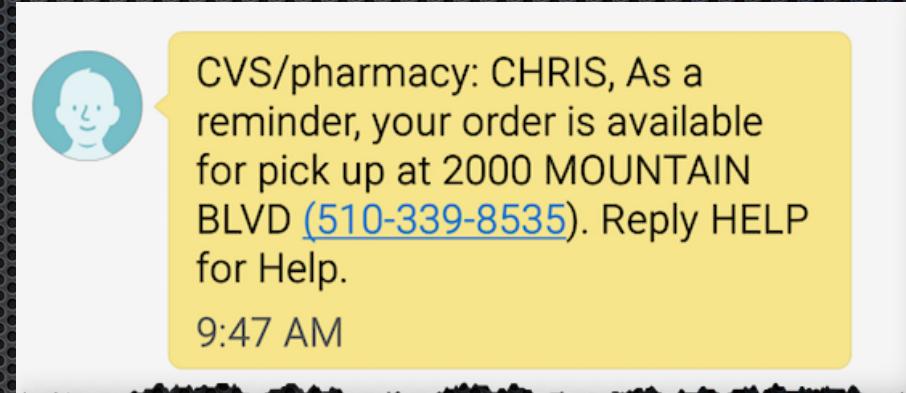
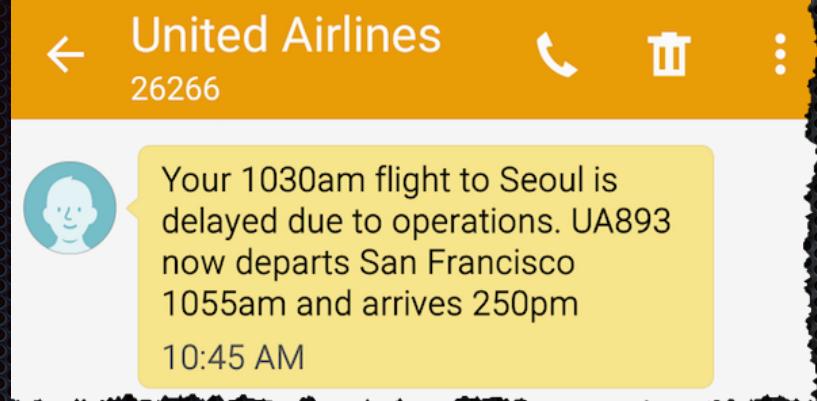
: something (especially something important or notable) that happens

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

: any one of the contests in a sports program

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

# Examples of events



@crichardson

# Instacart: event- driven grocery shopping

Orders

Alec P 2:15-3:15PM Whole Foods Market Shopping

Order Details

Whole Foods Market

1lb Organic Dry Farm Tomatoes per lb Found

2x Organic Ginger Root ~ 0.28 lbs Found

1x Organic Cilantro 1 bunch Found

0.5lb serrano chilis per lb Replaced

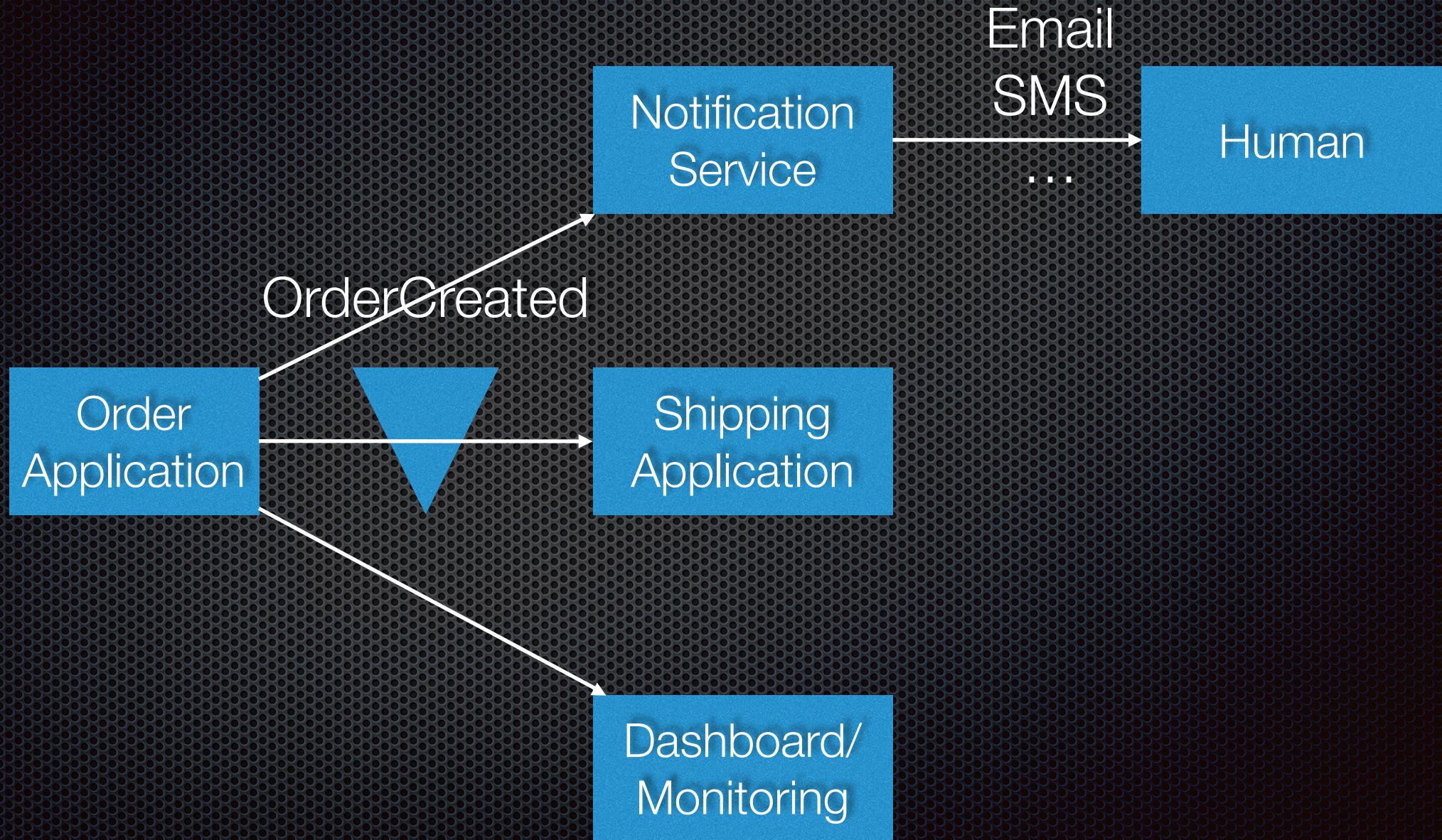
1x Serrano Chile Peppers ~ 0.25 lbs

@crichtson

# An event is produced when...

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

# Who consumes an event?



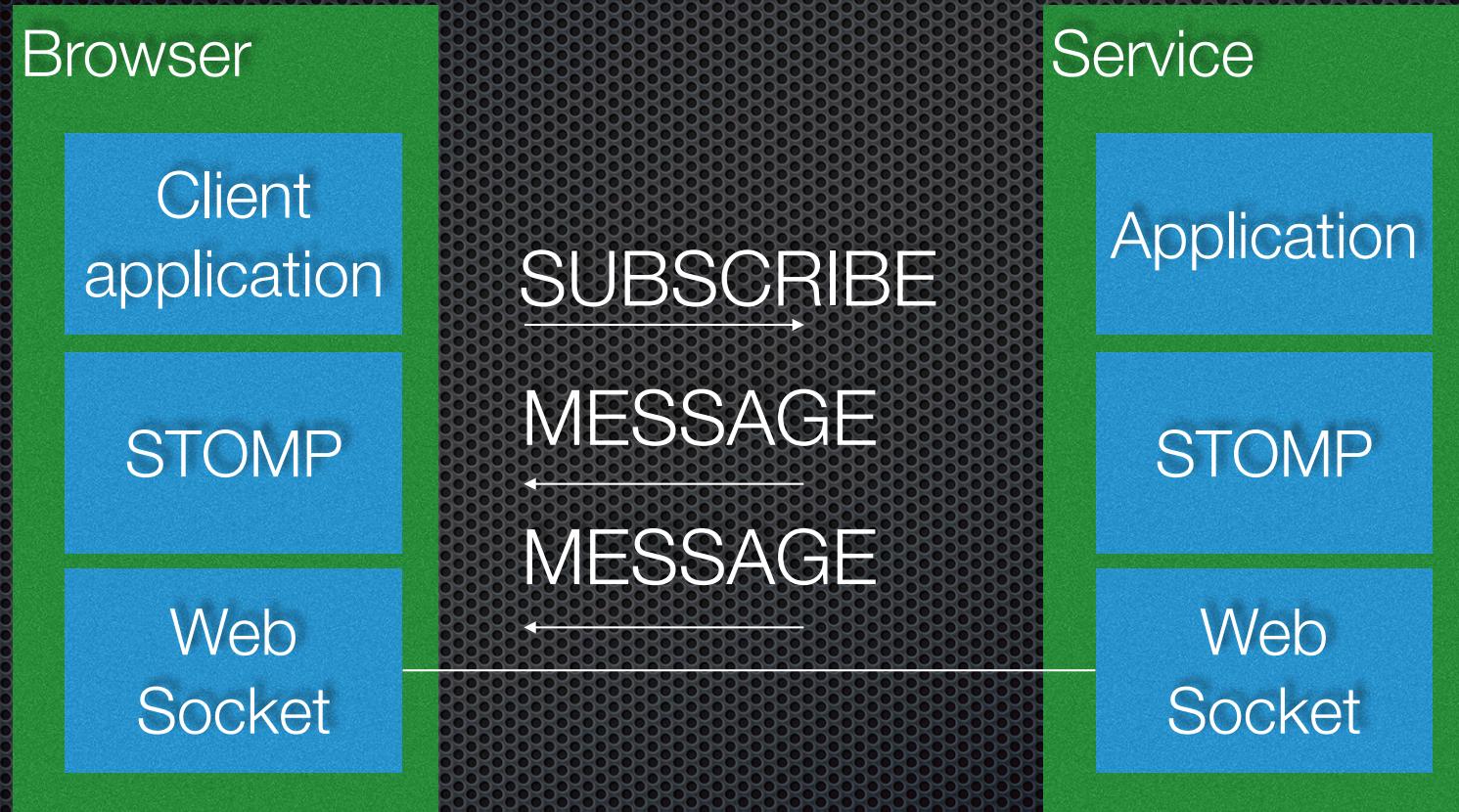
# How to deliver events?

# 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

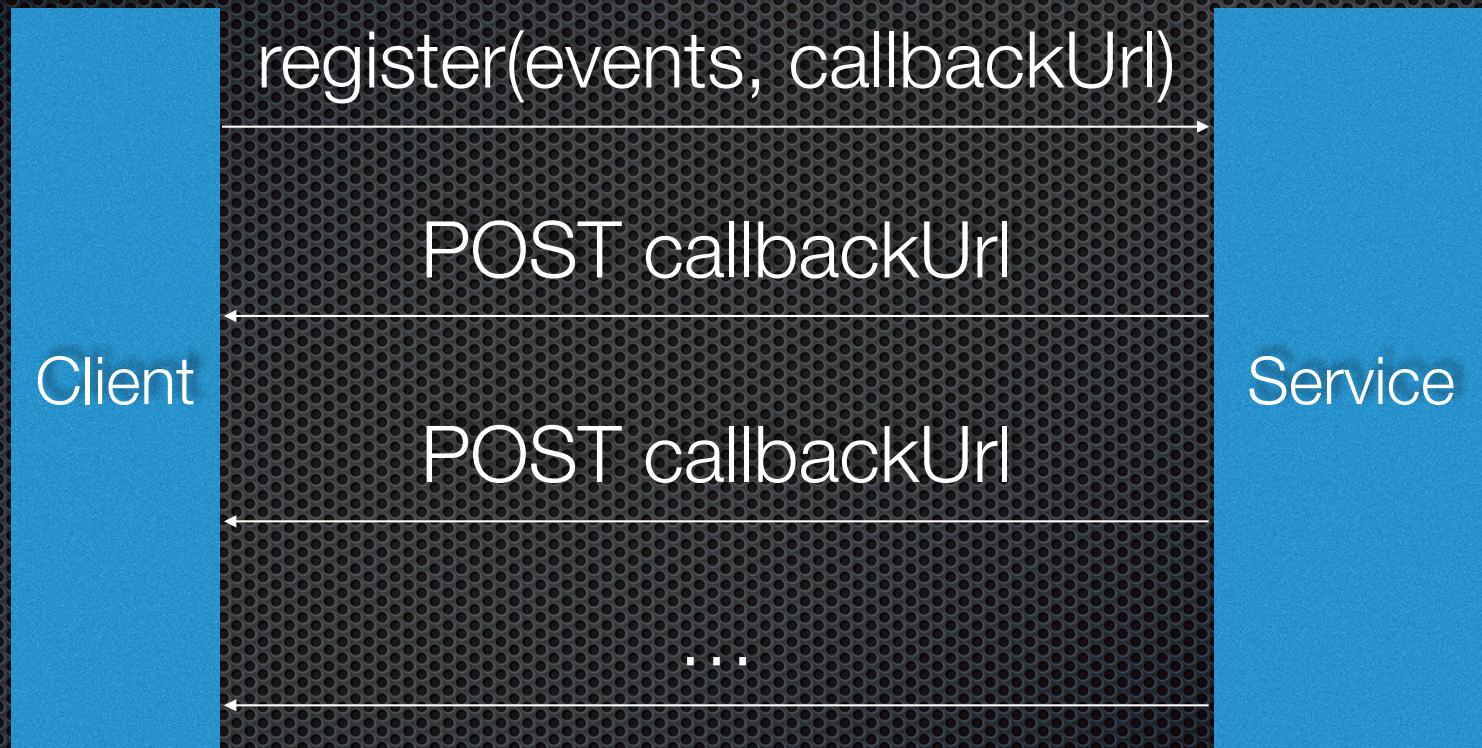
*High-latency, inefficient*

# 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?

@crichton

# 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



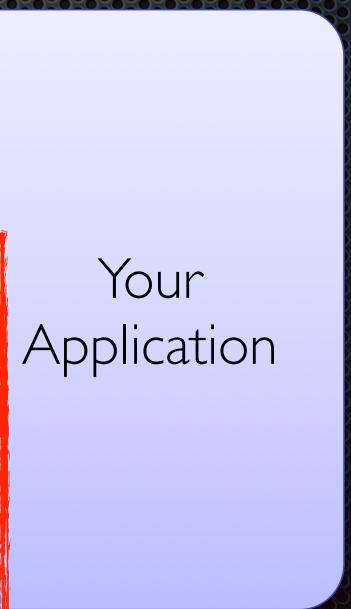
Manage resources  
Send SMS  
Initiate voice calls



Voice  
SMS



REST API

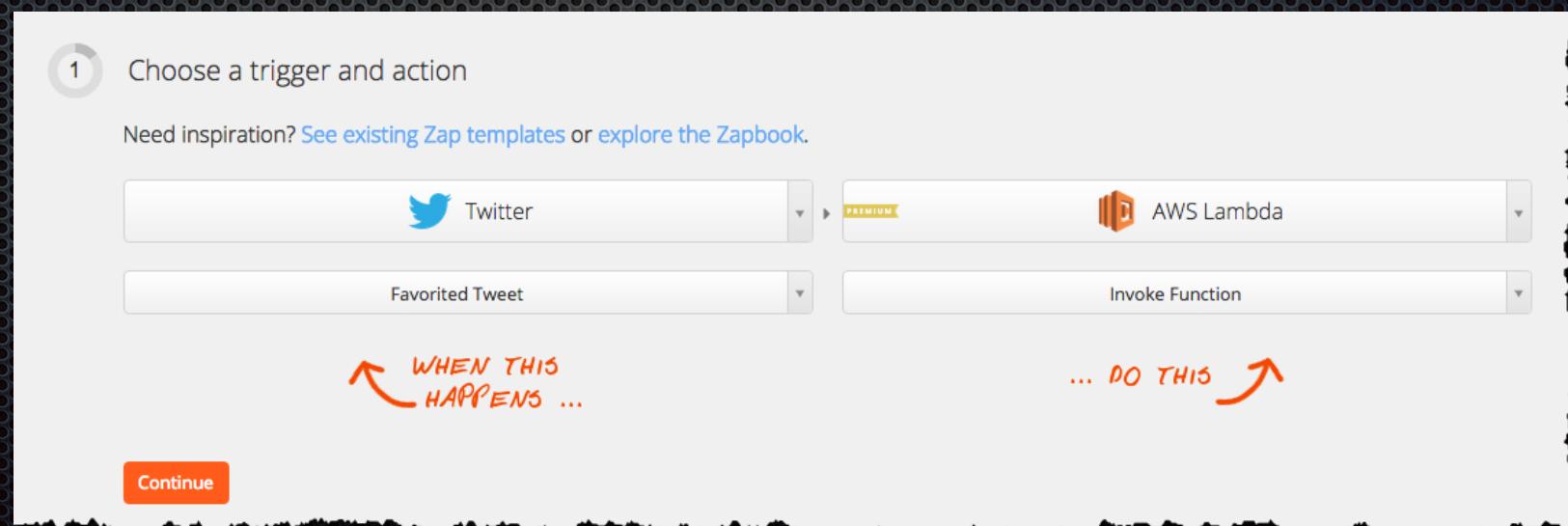


Phone number ⇒  
SMS URL + VOICE URL

Webhooks handle incoming SMS and  
voice calls

# Integration hubs - Zapier, IFTTT

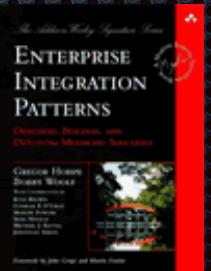
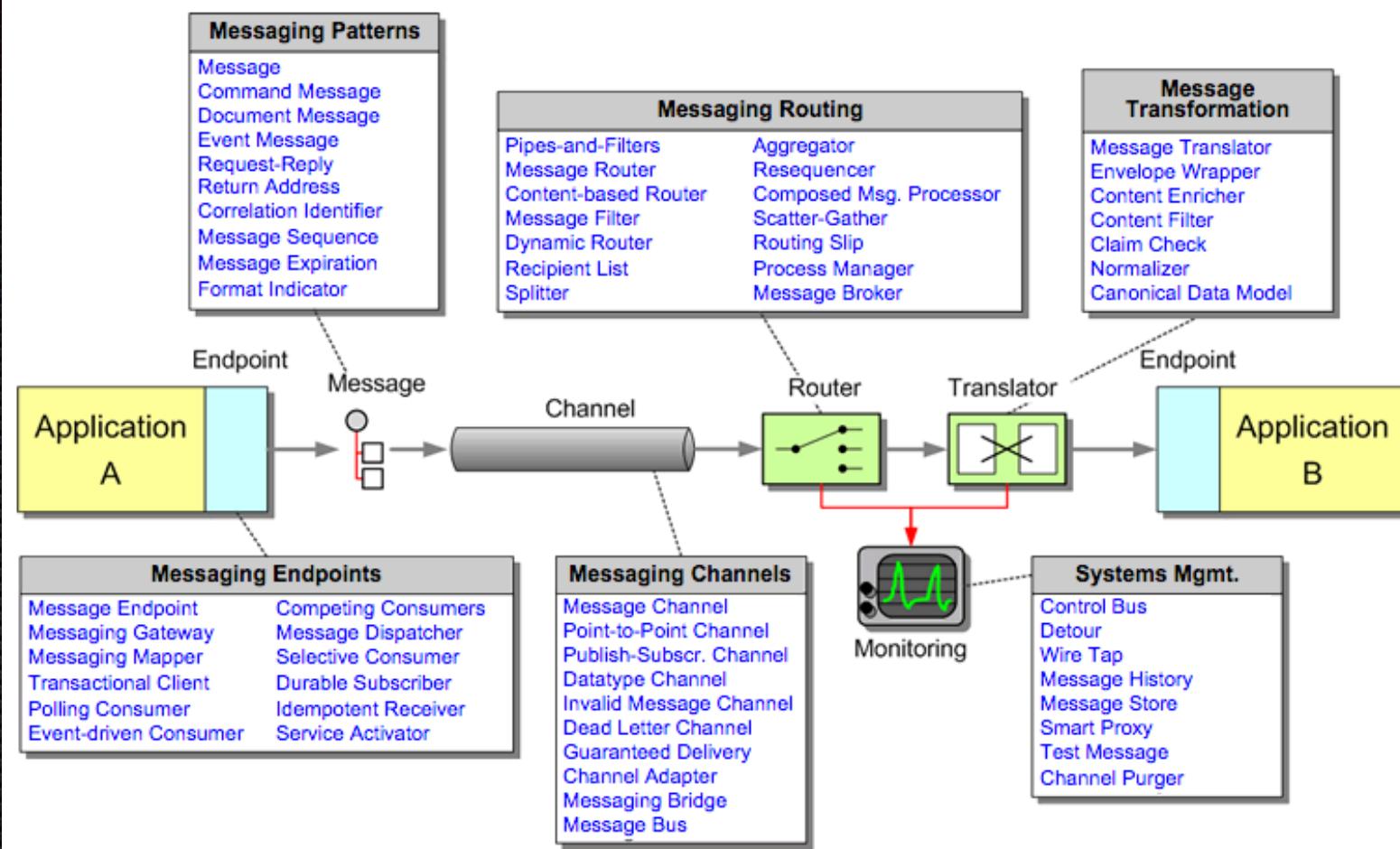
- Application abstraction:
  - Triggers - events published by application: polling or Webhooks
  - Action - operation supported by application, e.g. REST API end points



WebHooks  
=

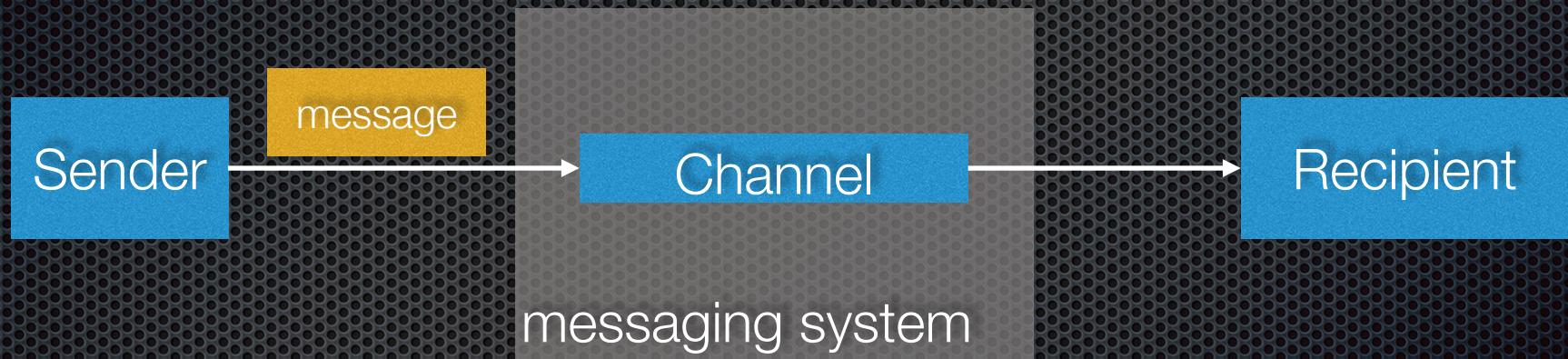
web friendly publish/subscribe

# Enterprise integration patterns



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

# Messaging-based IPC



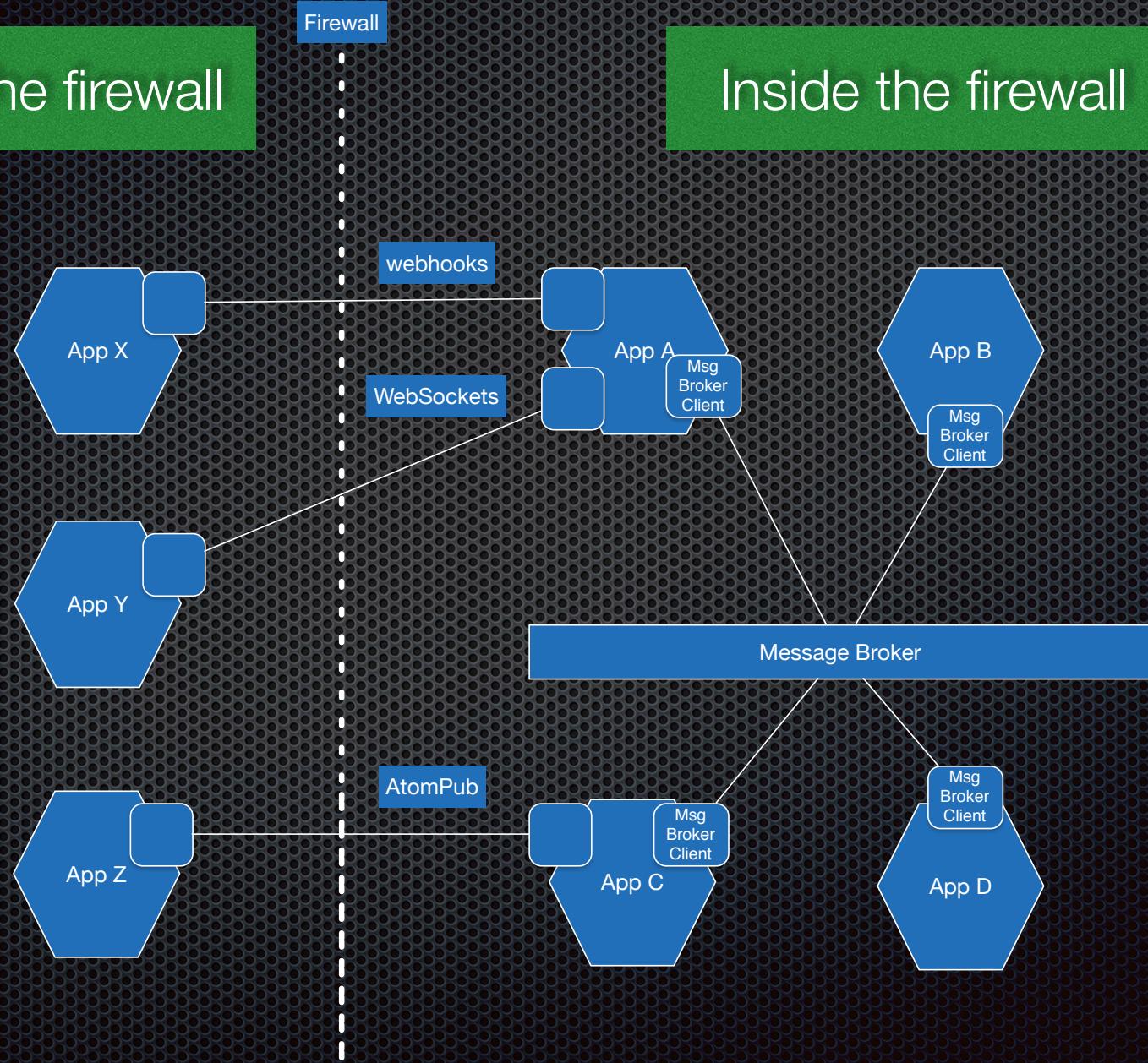
# Example messaging systems



# The event-driven enterprise

Outside the firewall

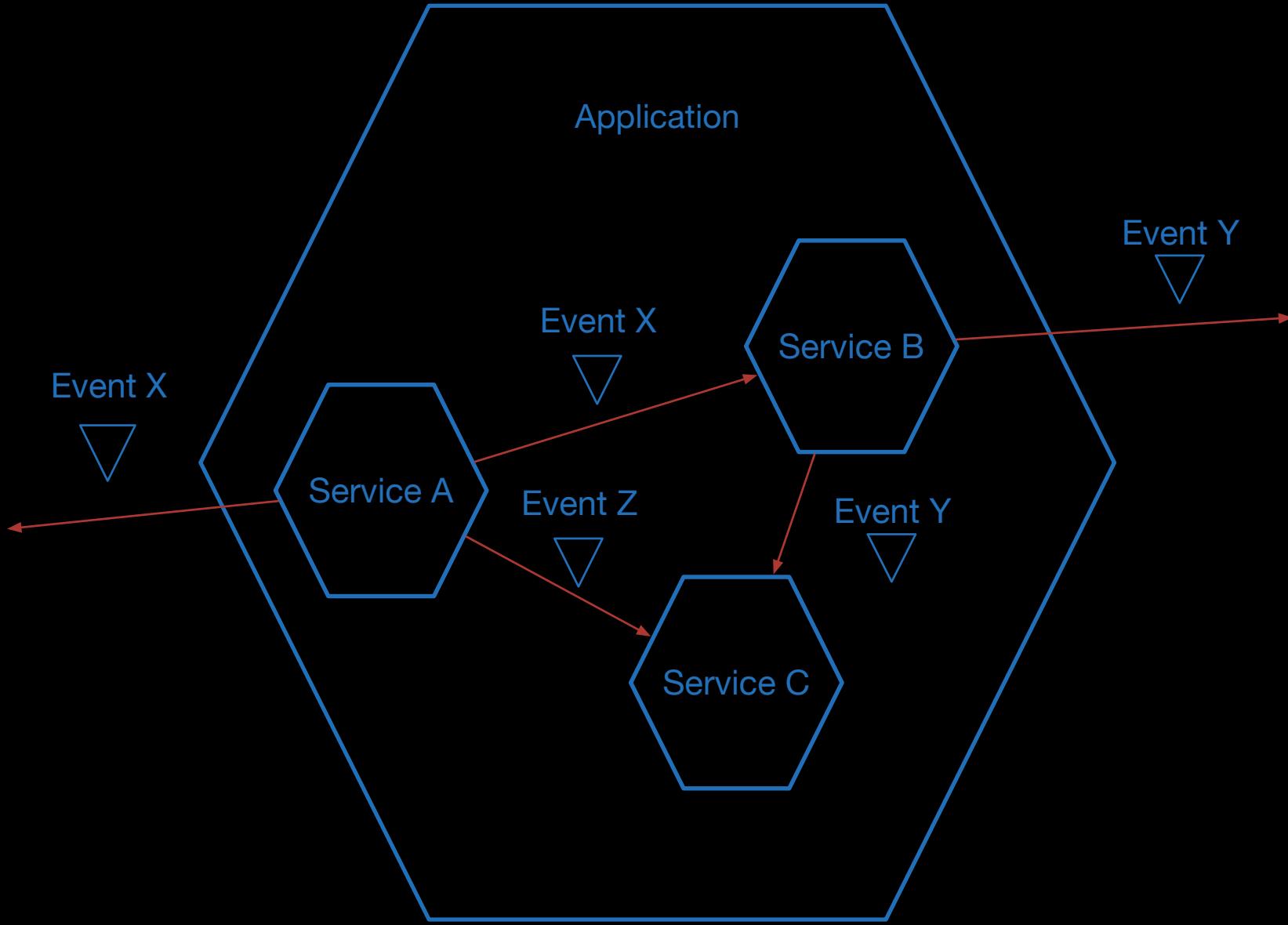
Inside the firewall



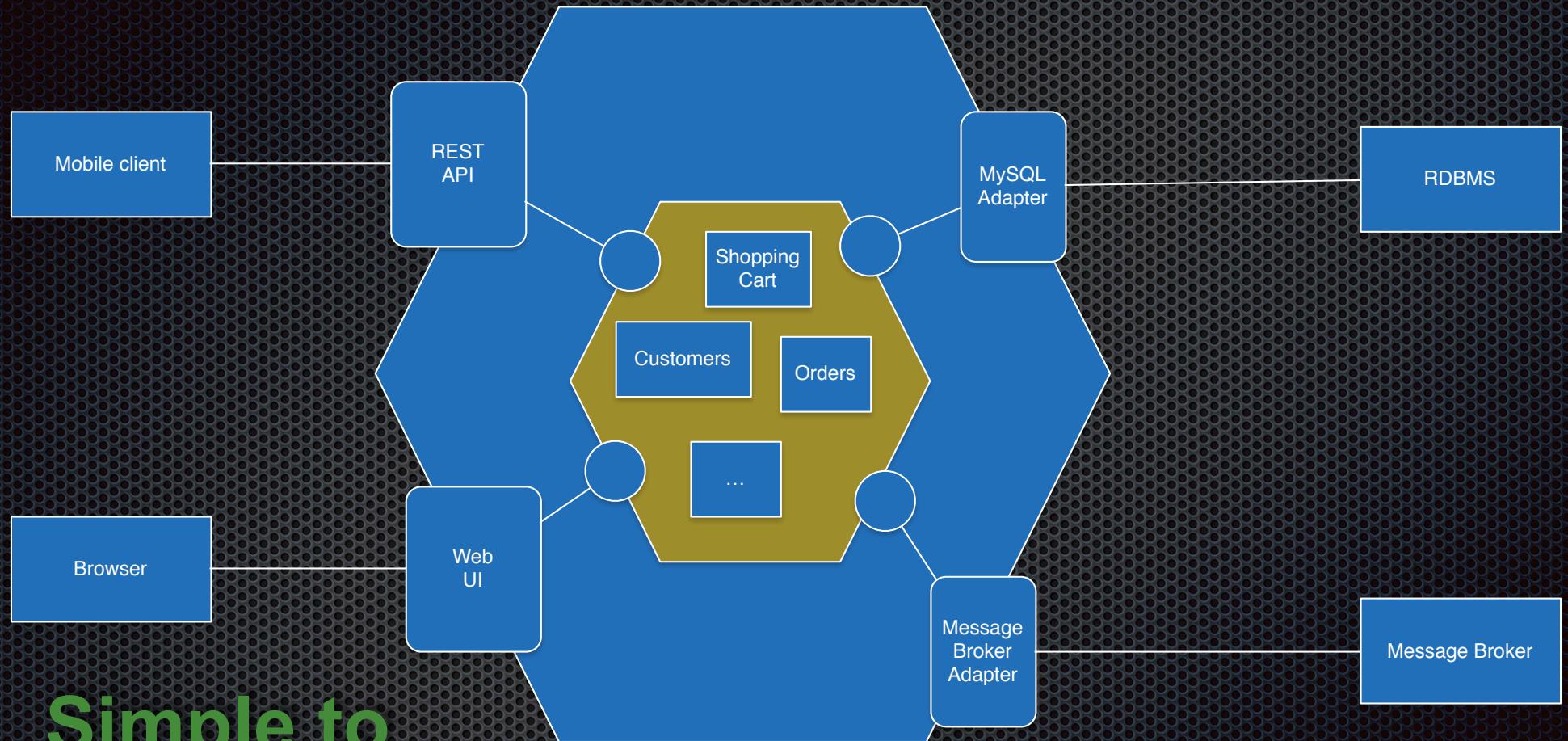
# Agenda

- 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



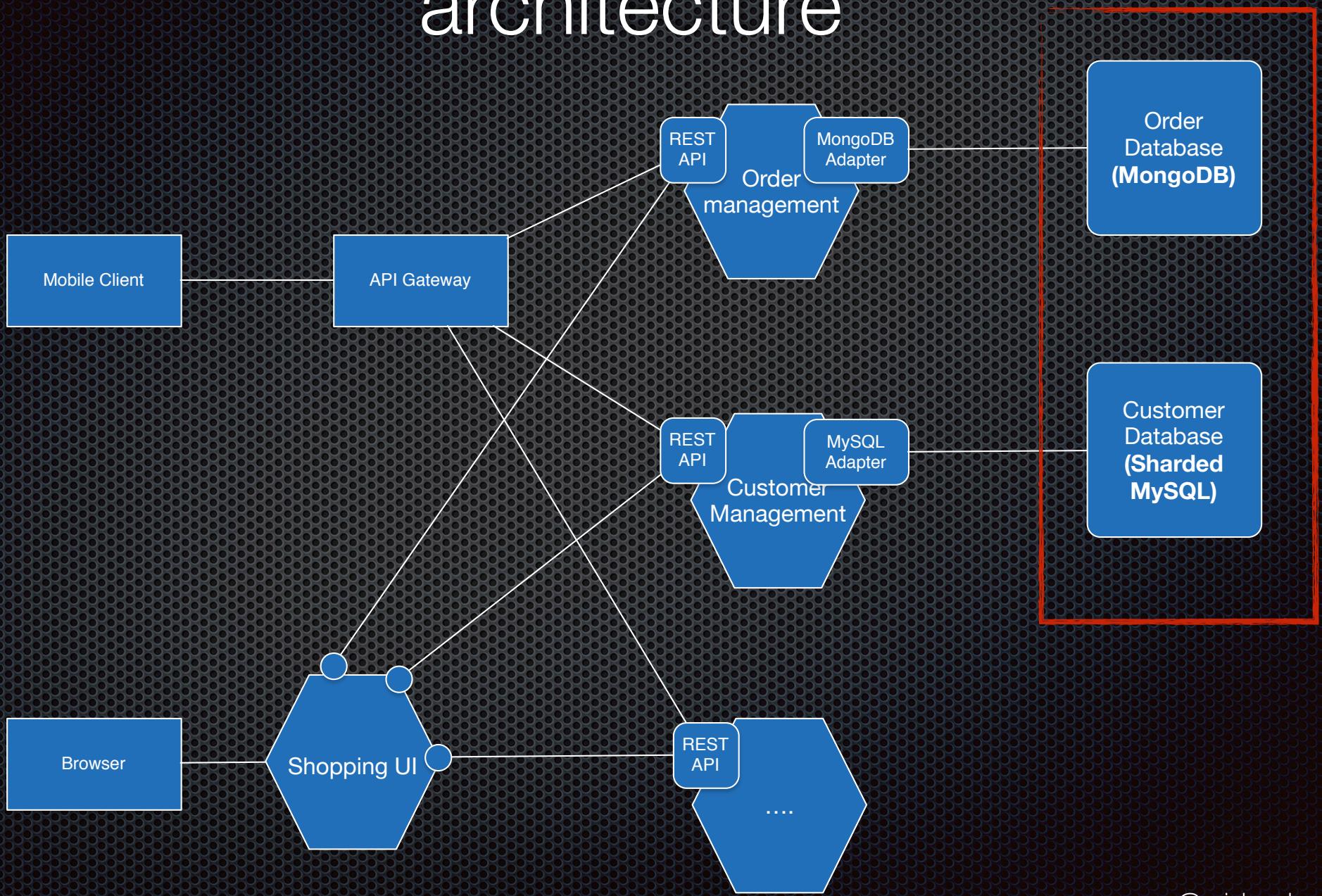
**Simple to develop  
test  
deploy  
scale**



But that leads\* to  
monolithic hell

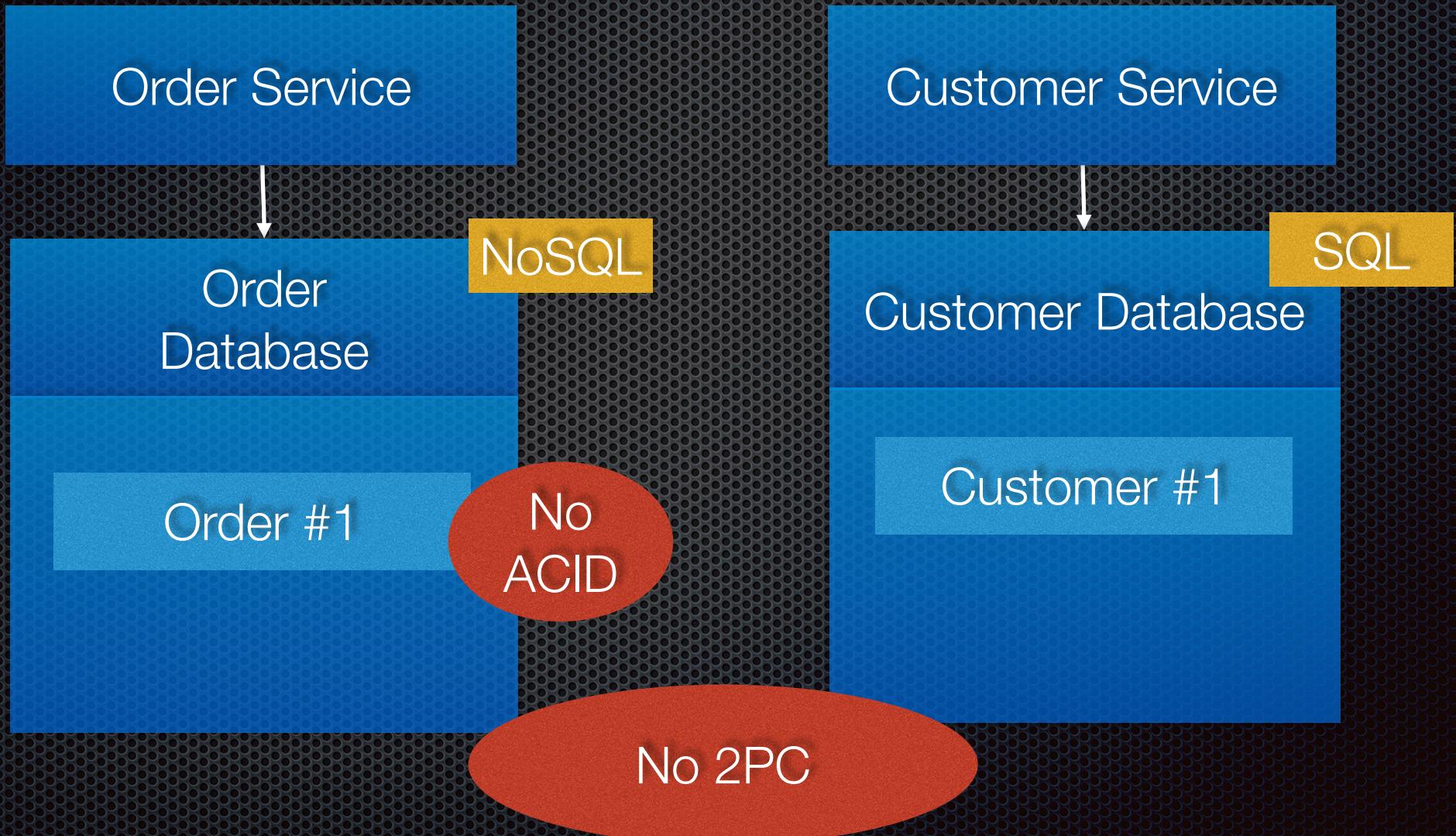
For large and/or complex applications...

# Today: use a microservice, polyglot architecture

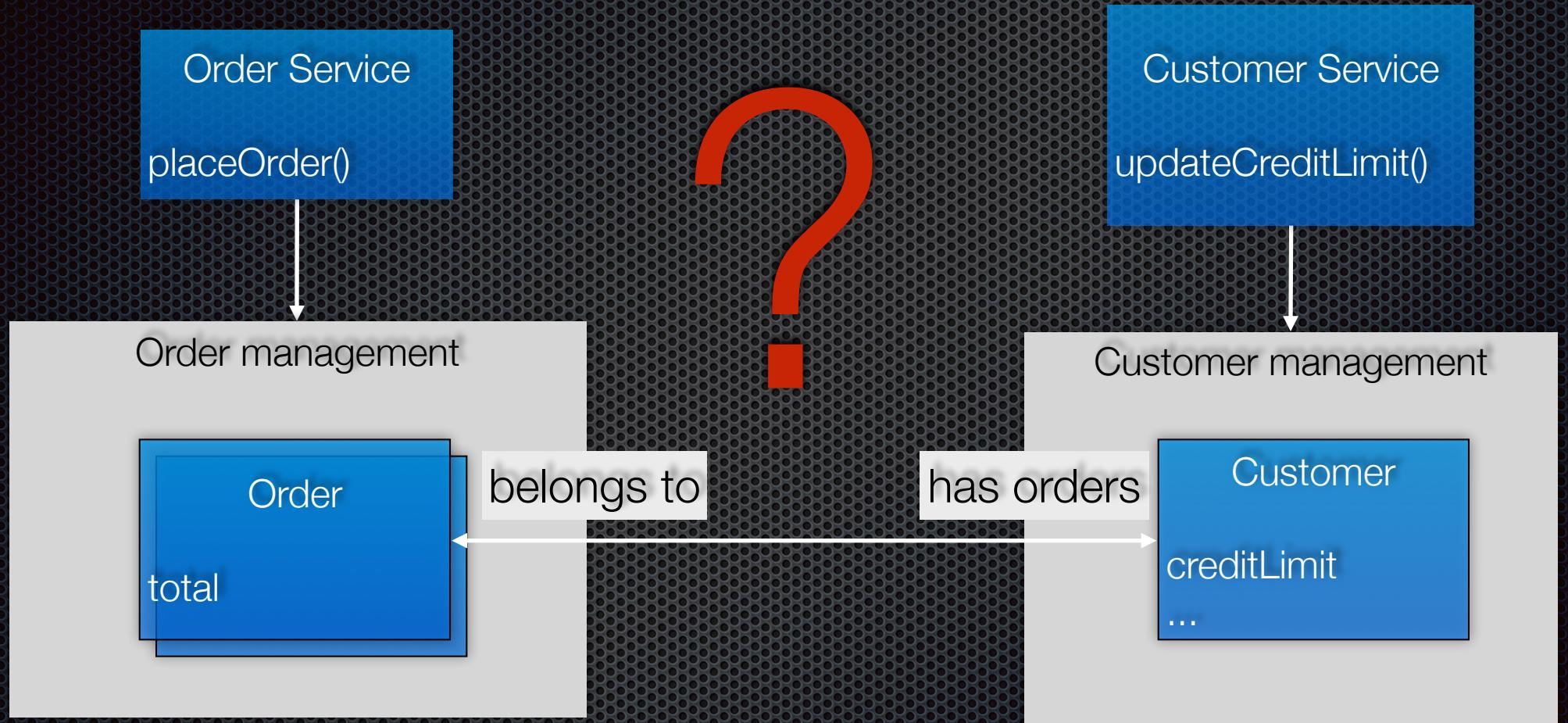


But now we have  
distributed data management  
problems

# Example: placing an order



# How to maintain invariants?

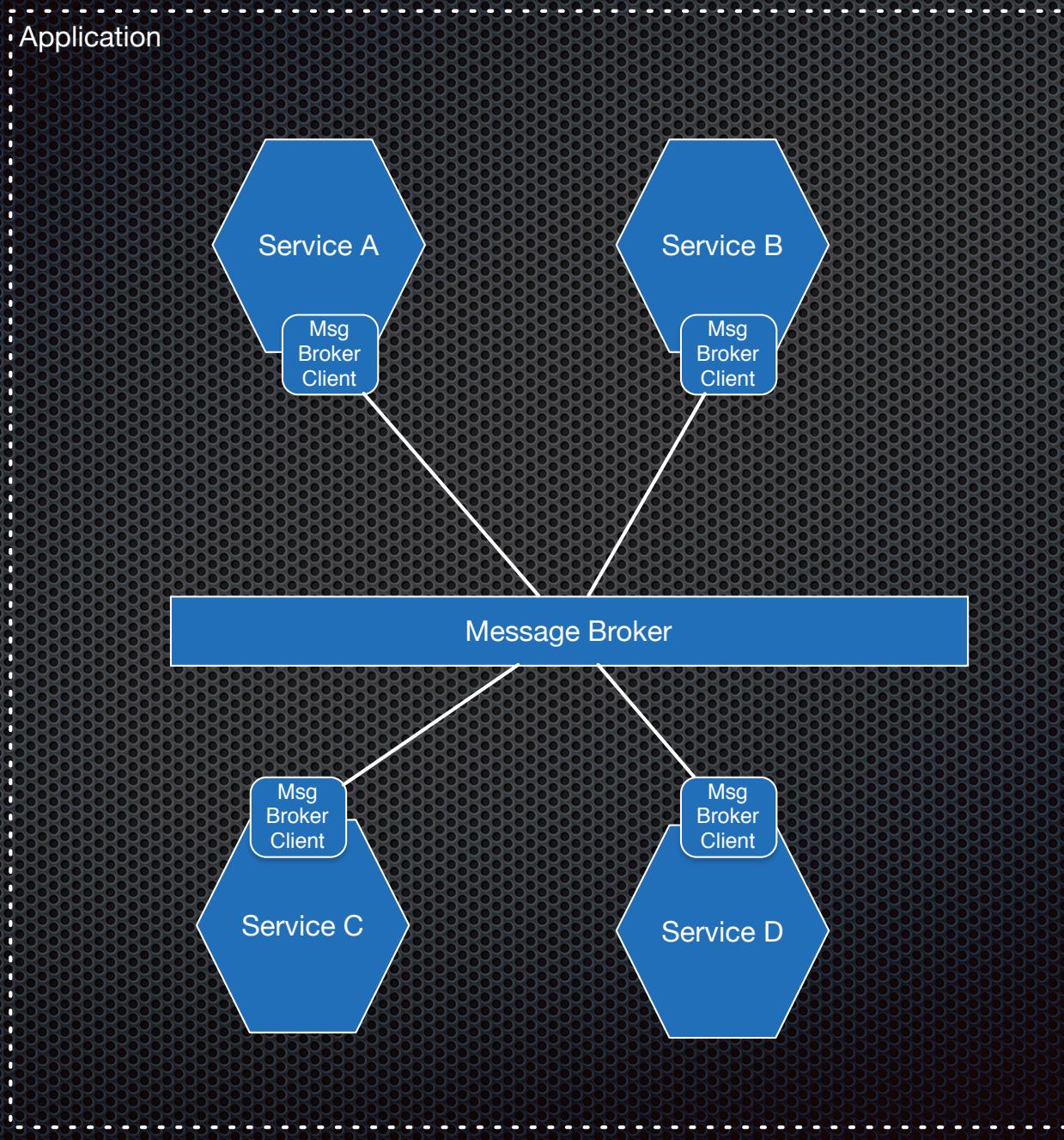


Invariant:  
 $\text{sum(open order.total)} \leq \text{customer.creditLimit}$

# 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

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

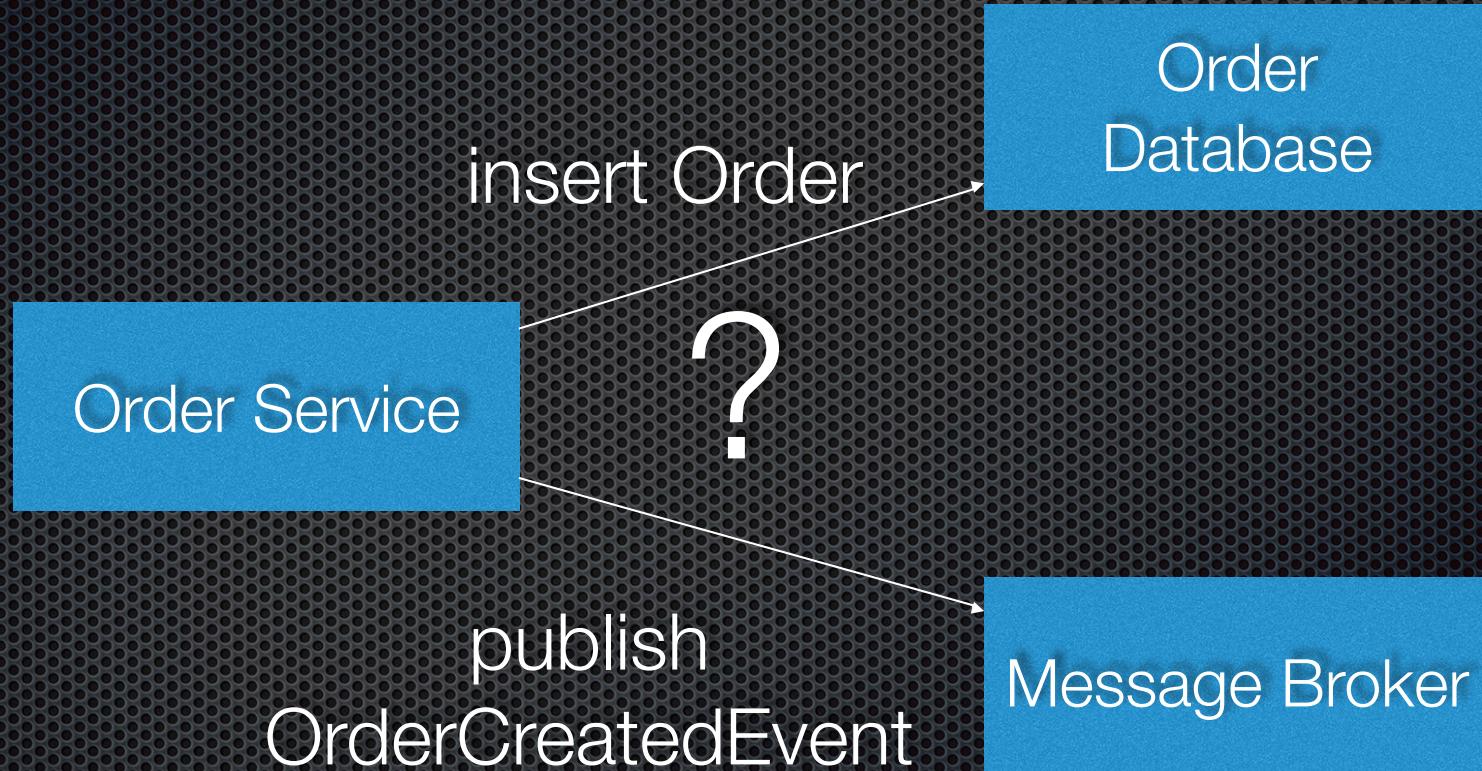
Message Bus

Now there are two problems  
to solve....

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

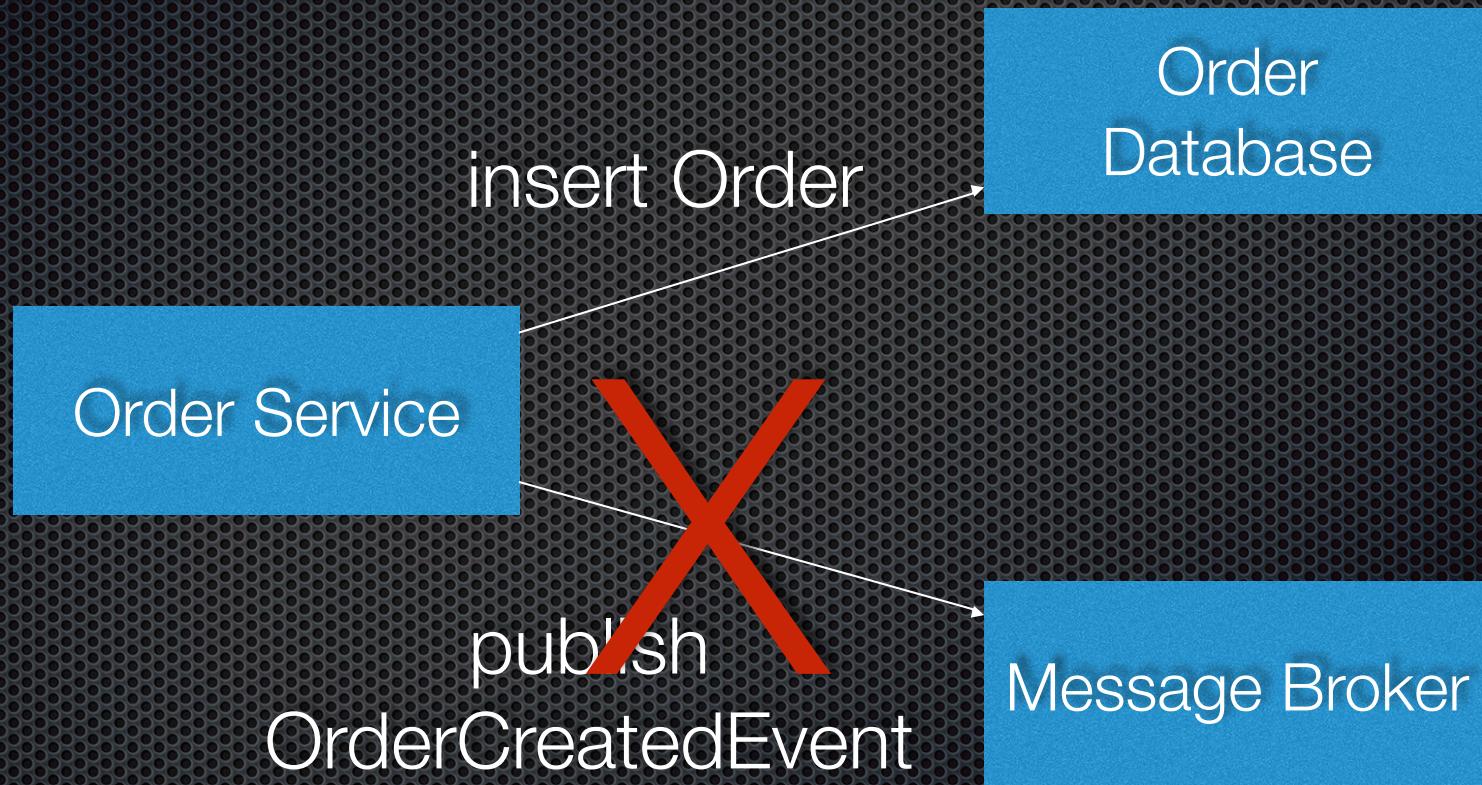
More on that later....

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



dual write problem

# Failure = inconsistent system



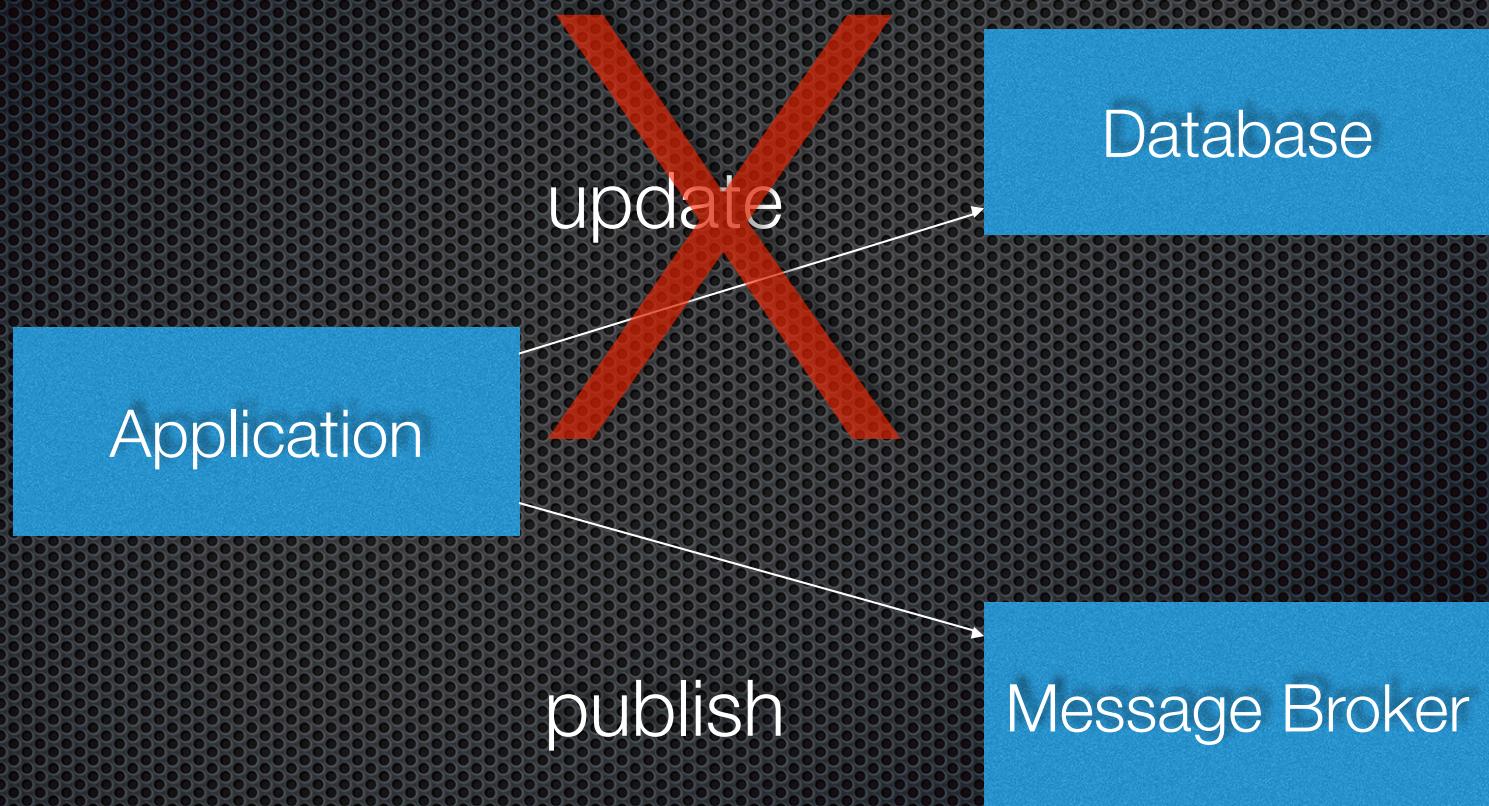


Two-phase commit

# Agenda

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

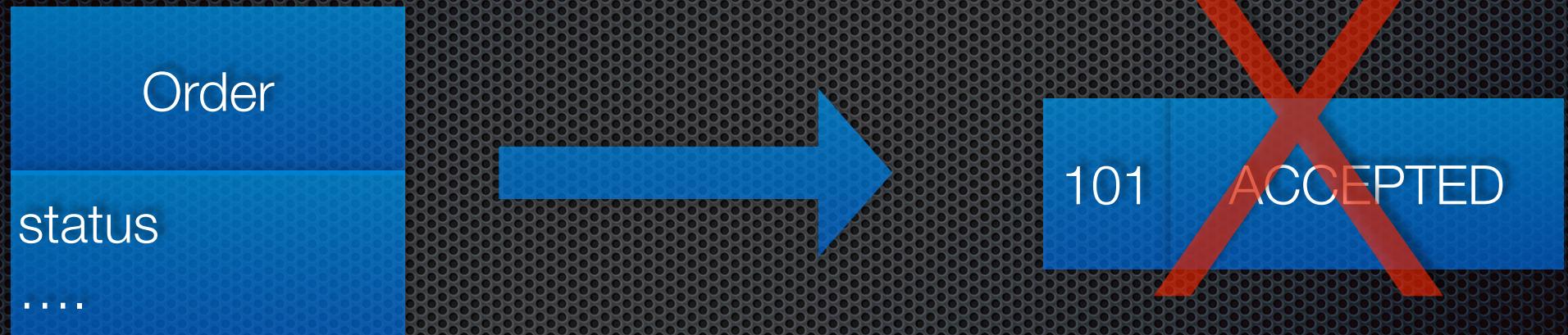
# Just publish events



# 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

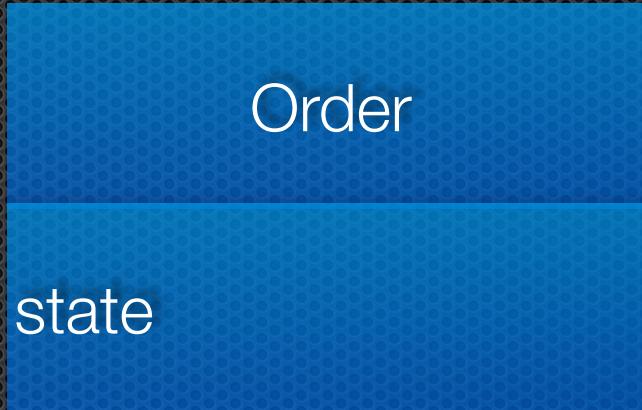
Event table

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

Events

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

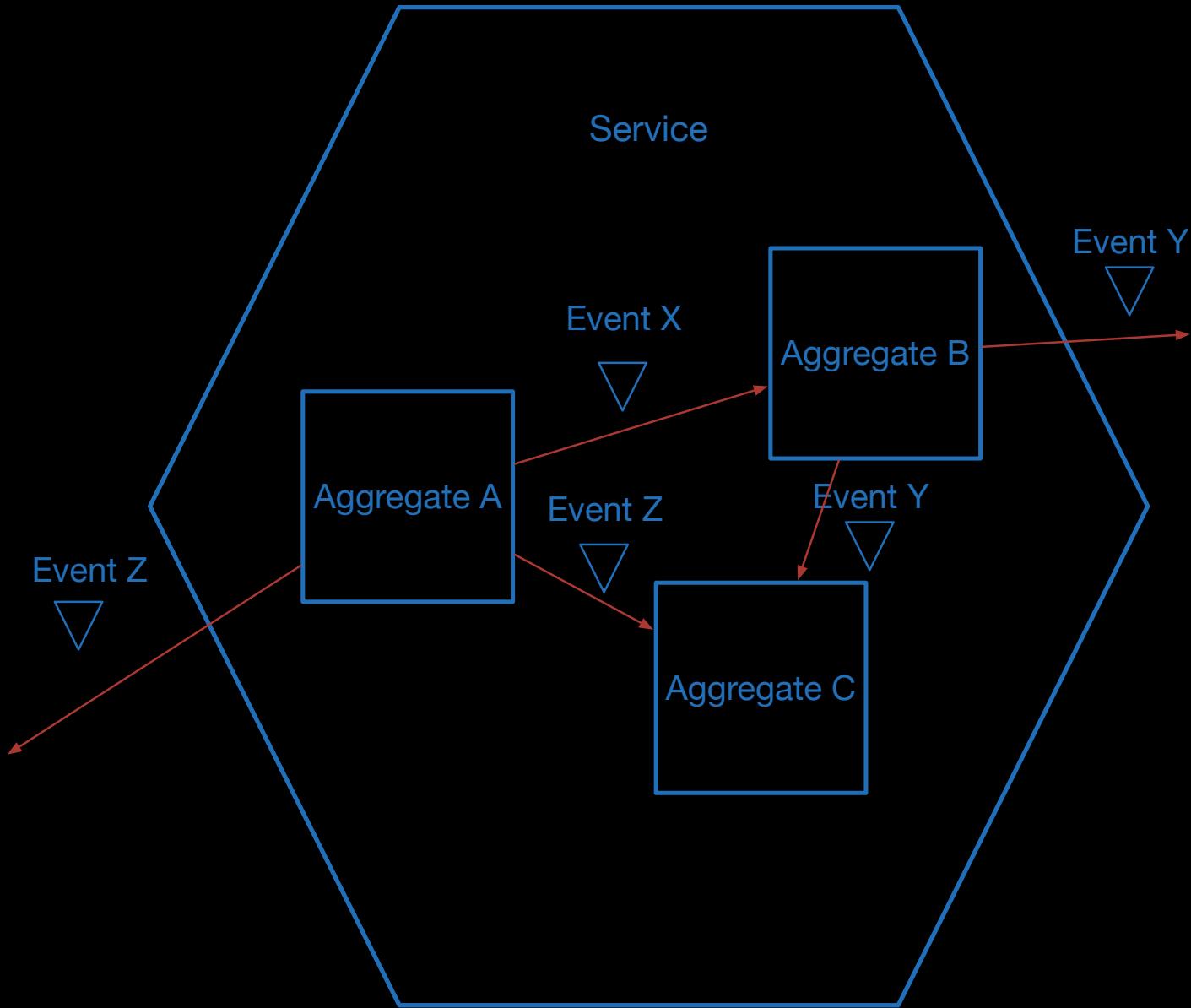


Periodically snapshot to avoid loading all events

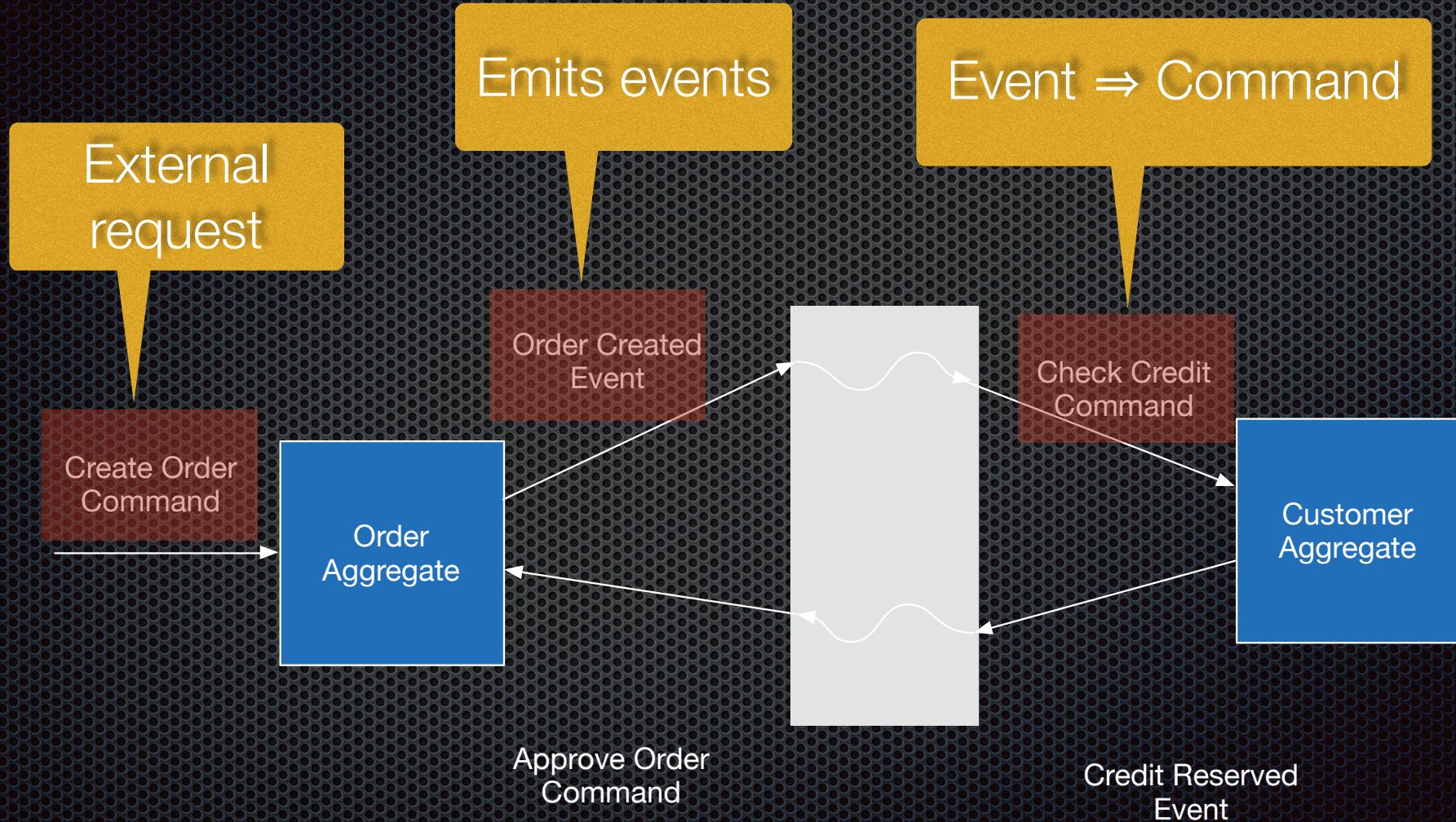
The present is a fold over  
history

currentState = foldl(applyEvent, initialState, events)

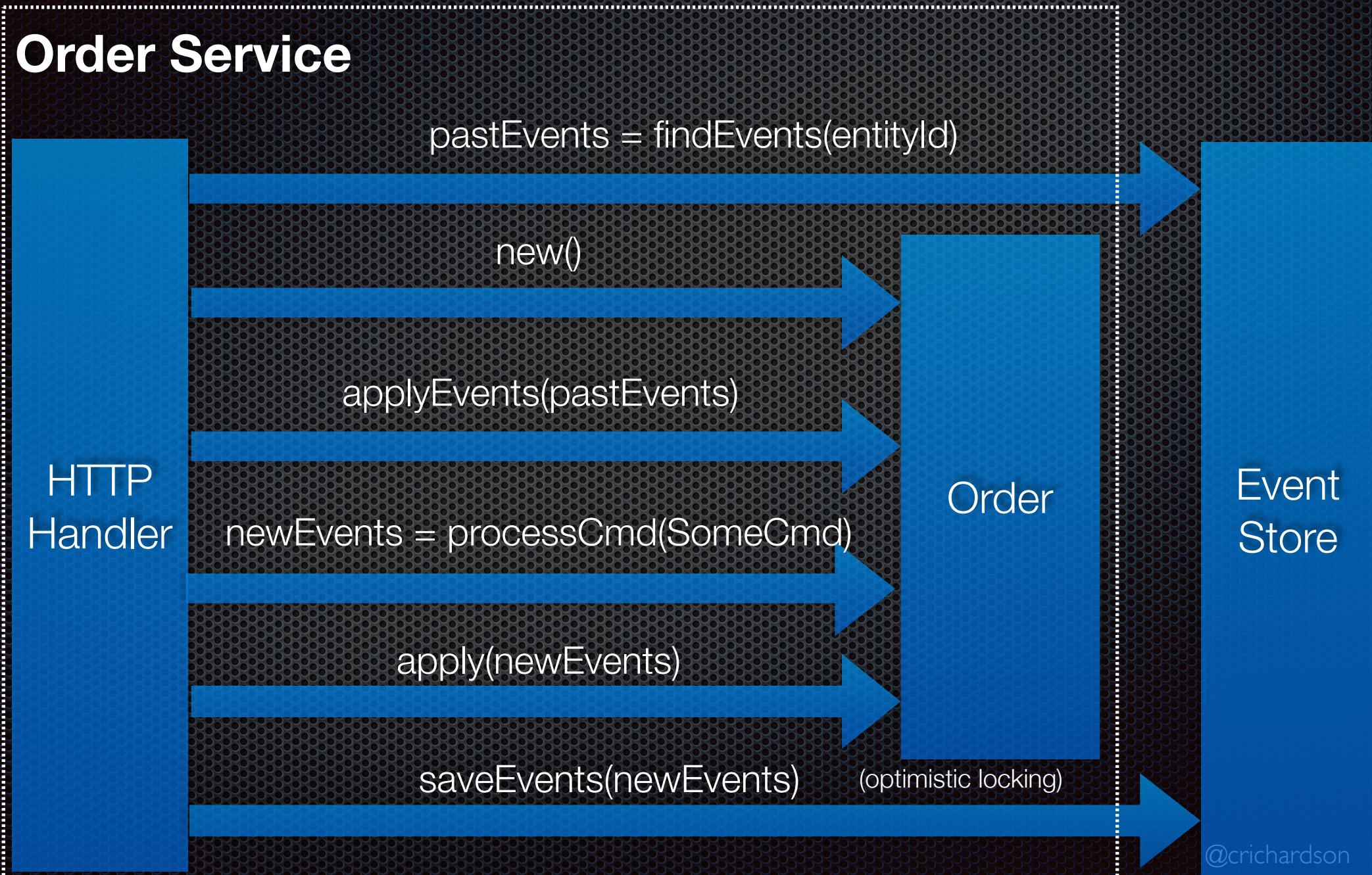
# Events at the core



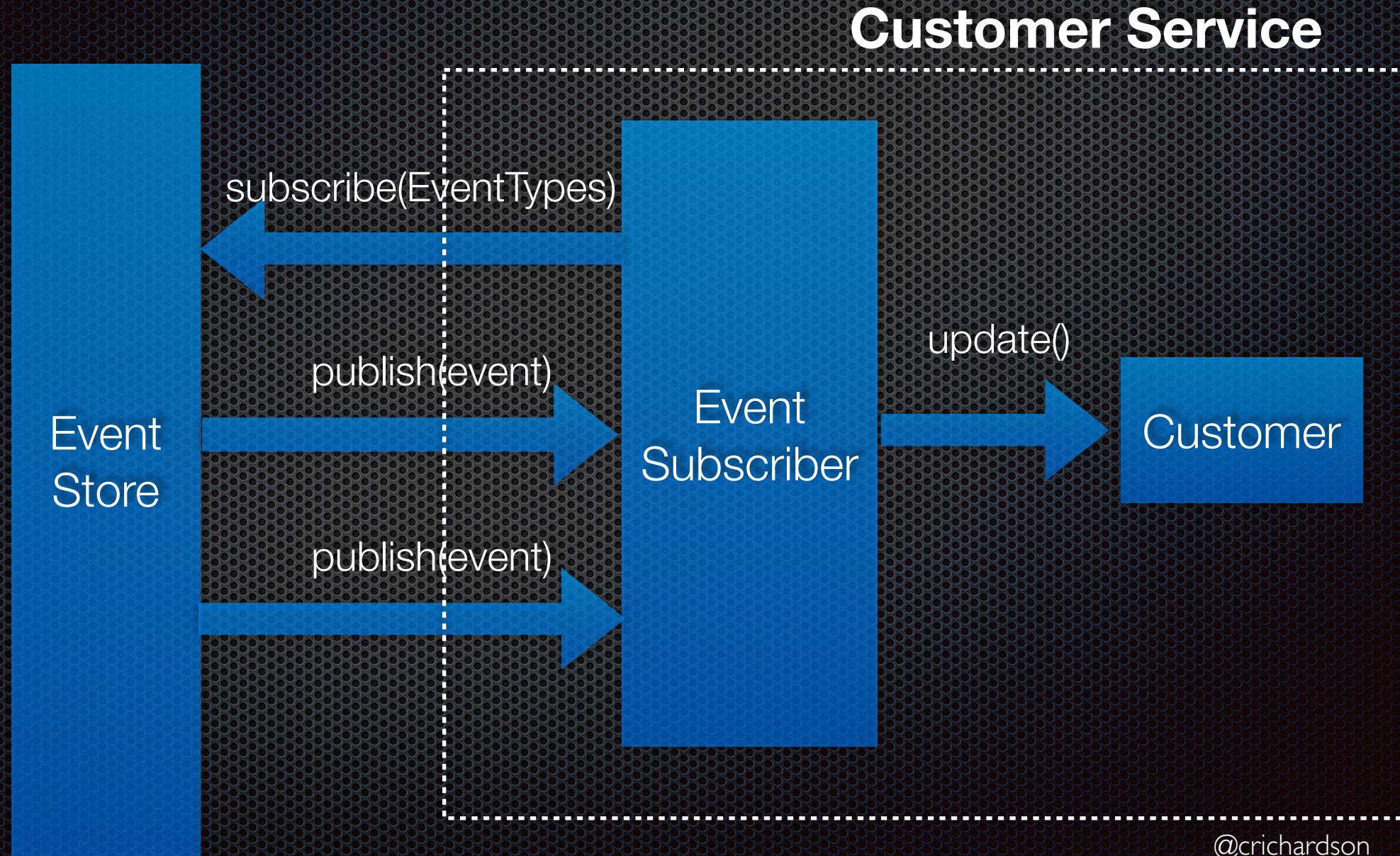
# Domain logic = event-driven aggregates



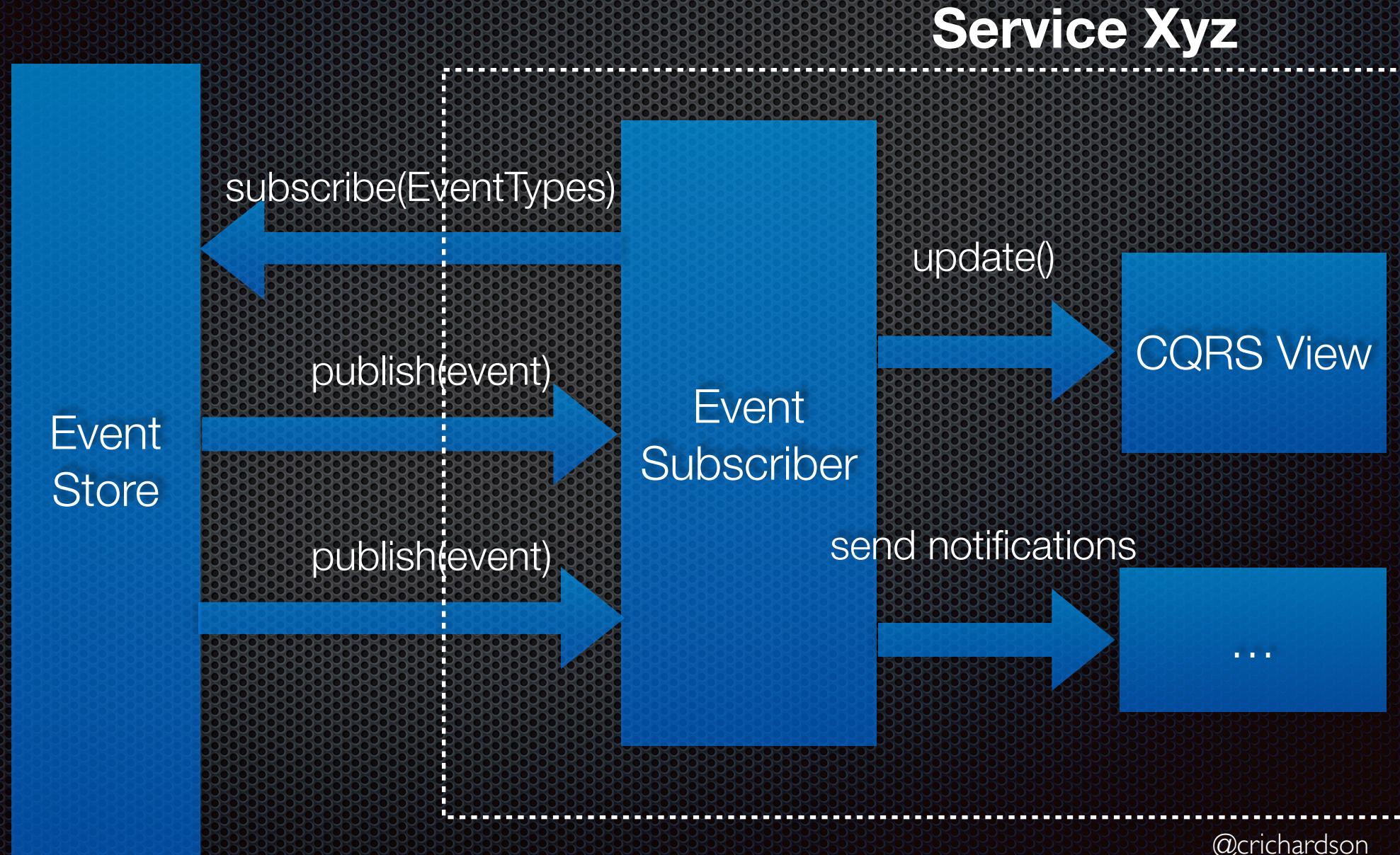
# Request handling in an event sourced application



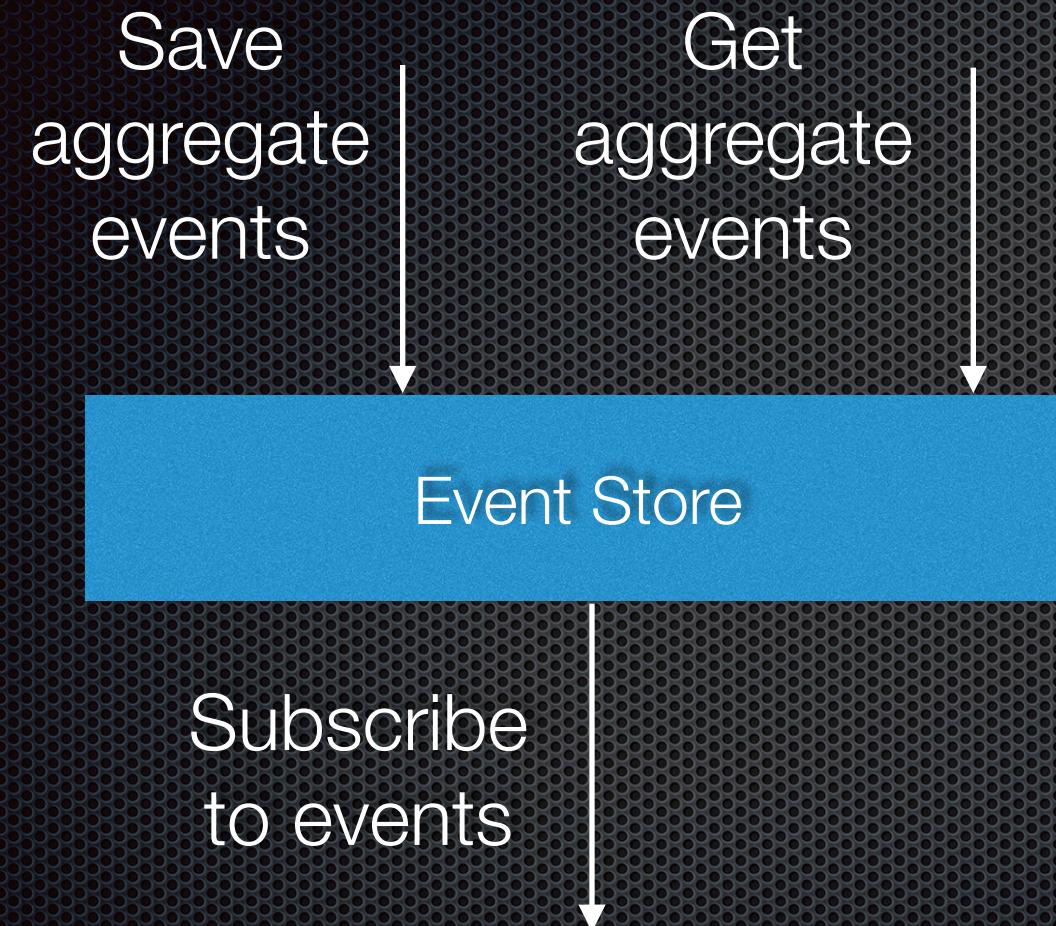
# Event Store publishes events consumed by other services



# Event Store publishes events consumed by other services



# Event store = database + message broker



- Hybrid database and message broker
- Implementations:
  - Home-grown/DIY
  - [geteventstore.com](http://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

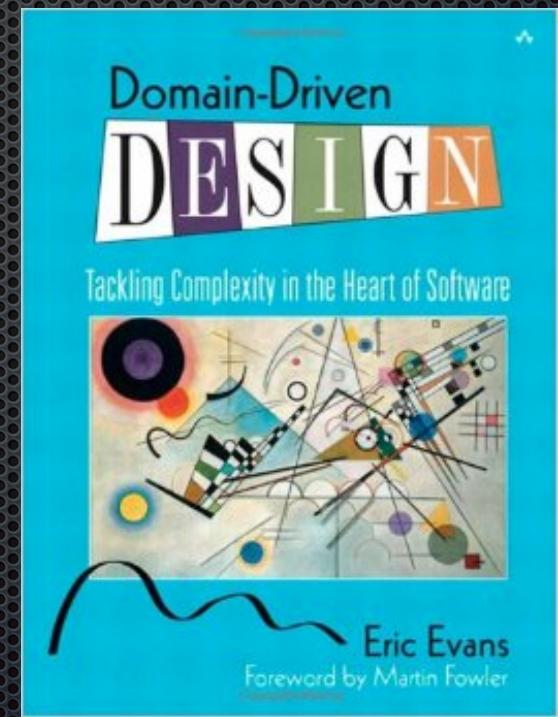
- Querying the event store can be challenging
- Some queries might be complex/inefficient, e.g. accounts with a balance > X
- Event store might only support lookup of events by entity id
- Must use Command Query Responsibility Segregation (CQRS) to handle queries ⇒ application must handle eventually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

# Agenda

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

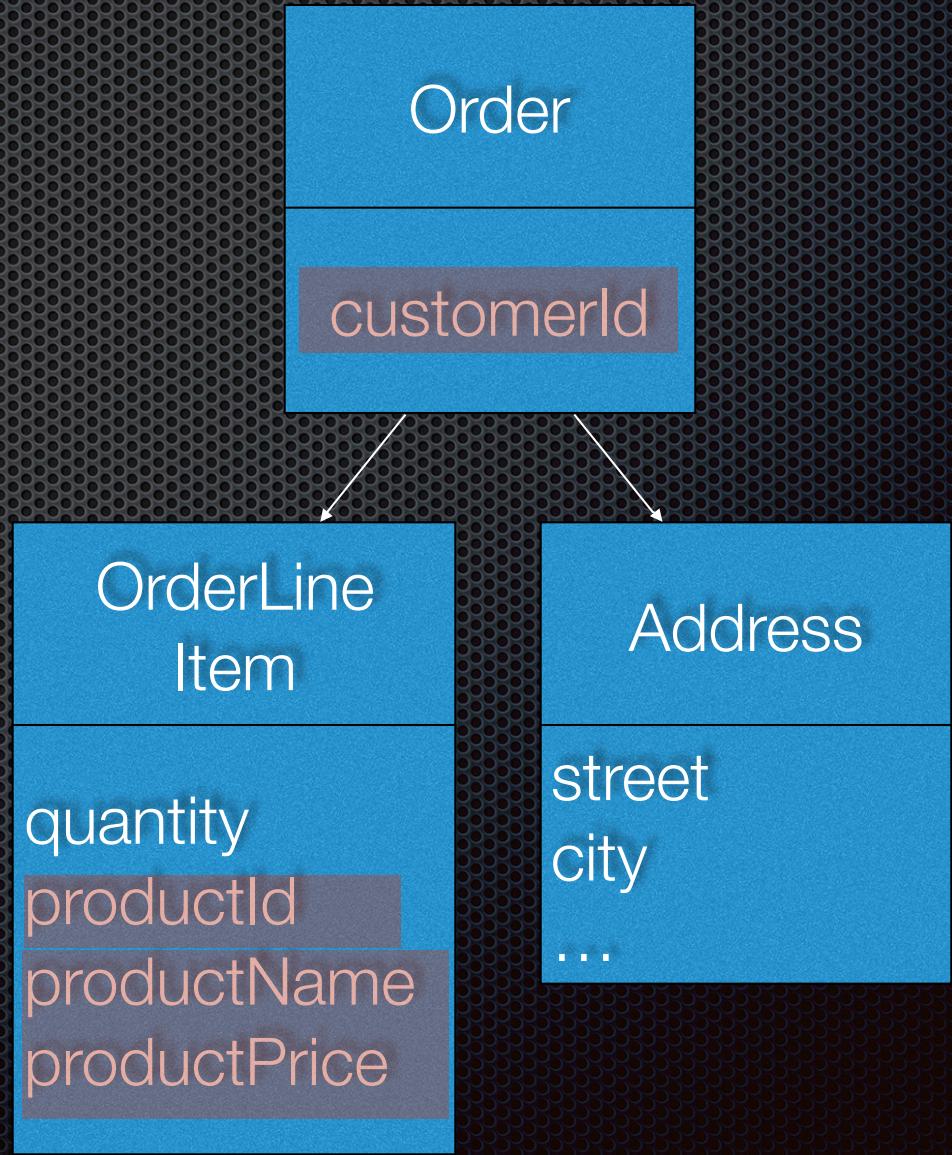
# Use the familiar building blocks of DDD

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

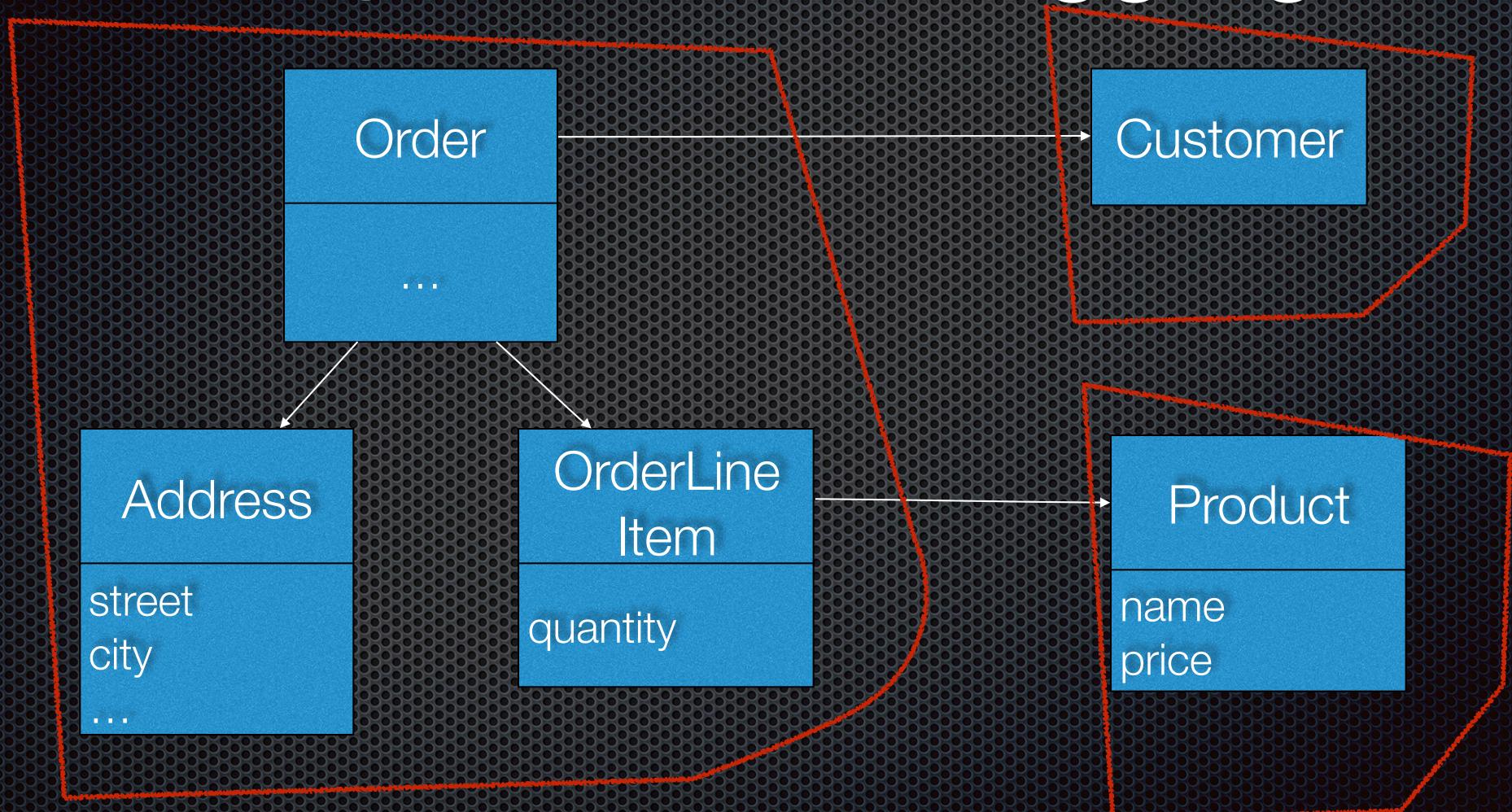


# About Aggregates

- 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

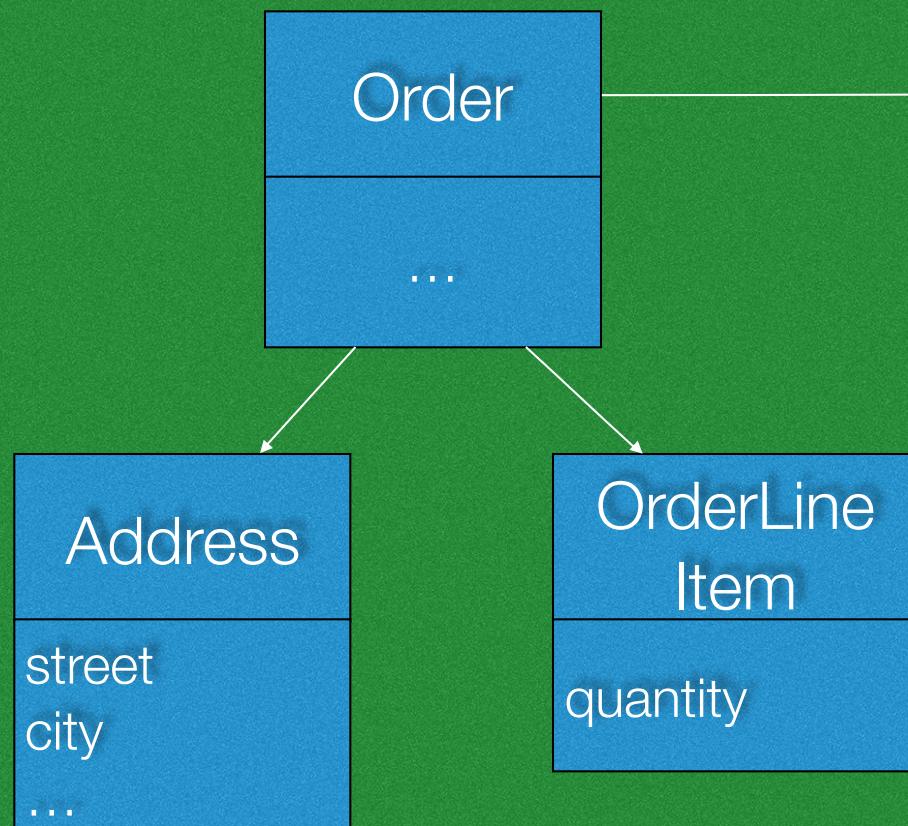


# Domain model = collection of **loosely** connected aggregates

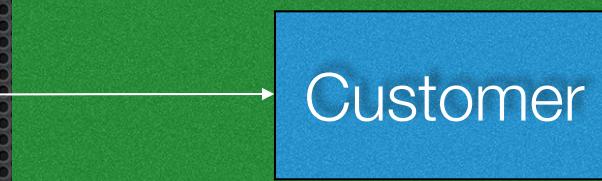


# Easily partition into microservices

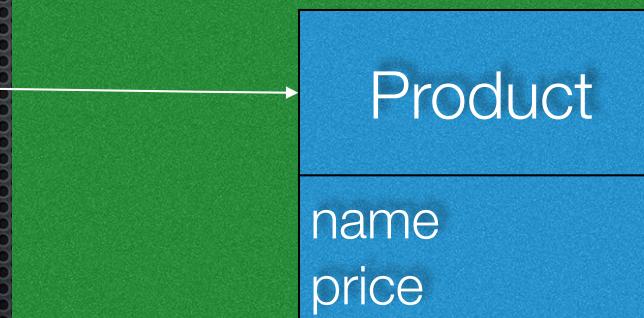
Order service



Customer service



Product service

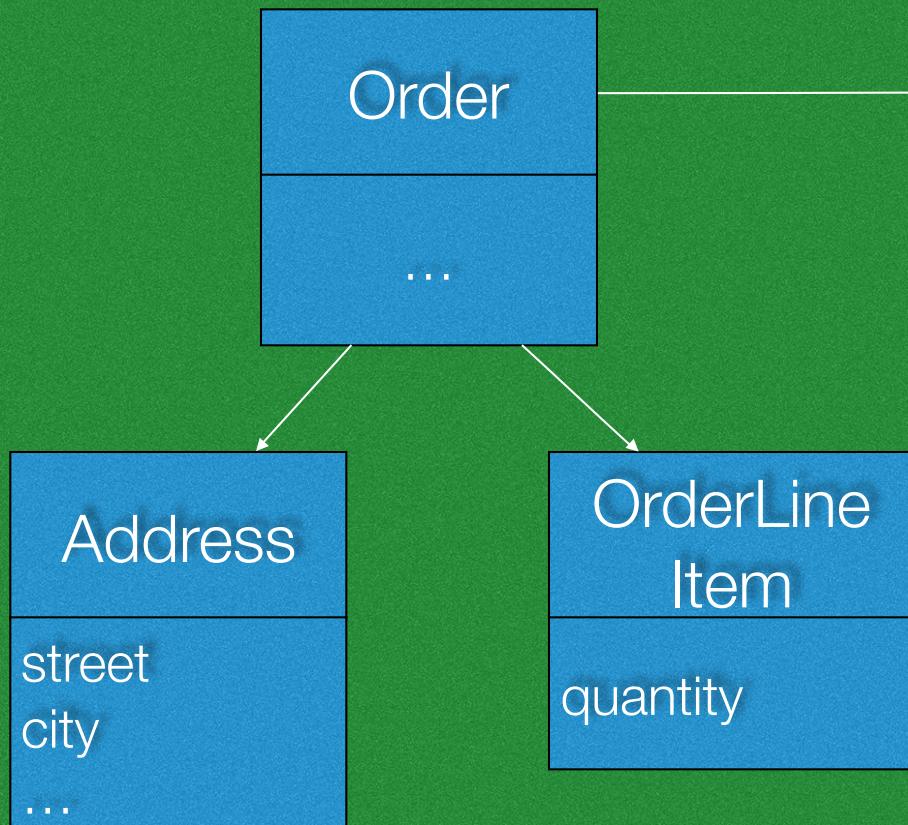


# Transaction = processing one command by one aggregate

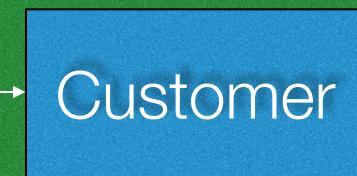
- No opportunity to update multiple aggregates within a transaction ⇒ event driven eventual consistency between aggregates
- If an update must be atomic (i.e. no compensating transaction) then it must be handled by a single aggregate
- **Therefore**, aggregate granularity is important

# Transaction scope = service

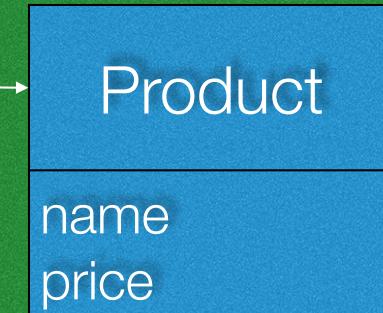
Order service



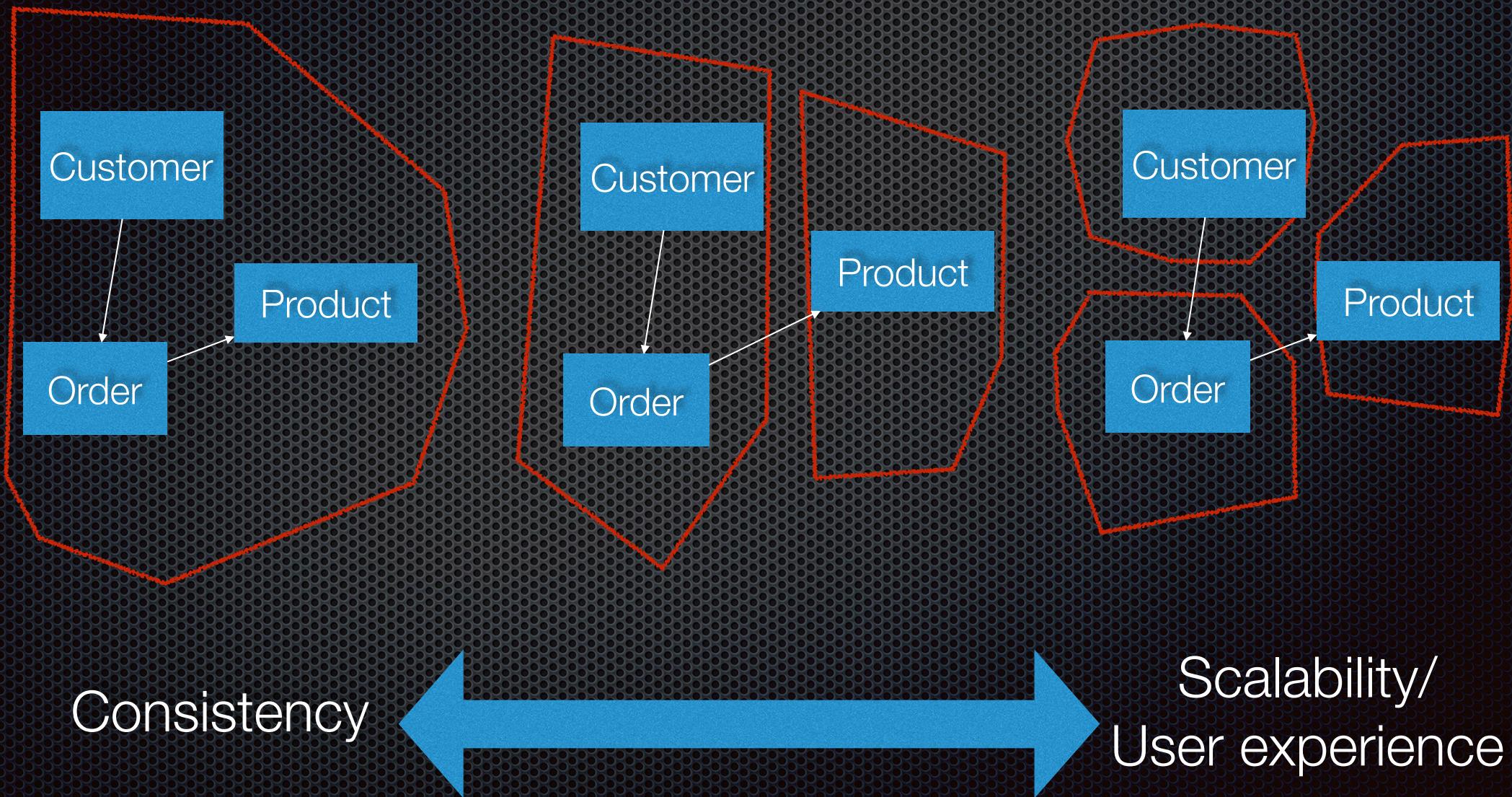
Customer service



Product service



# Aggregate granularity



# Designing domain events

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

Event metadata

Required by aggregate

Enrichment:  
Useful for consumers

ProductAddedToCart

id : TimeUUID  
senderId: UUID  
productId  
productName  
productPrice  
...

# 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>

# OO = State + Behavior

State

Customer

creditLimit

creditReservations : Map<OrderId, Money>

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

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

...

void apply(CustomerCreatedEvent anEvent) { ... }

void apply(CreditServedEvent anEvent) { ... }

...

# 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  
}
```

# Kanban board example

Kanban App

chris+kb1@chrisrichardson.net ▾ Sign Out

Board: Alpha release

To Do: 0

Doing: 0

Done: 1

create documentation H x  
Created by:  
chris+kb1@chrisrichardson.net  
Changed less than a minute ago

Close x

Backlog: 0

Create Task

@christianrichardson

The image shows a Kanban board with the following state:

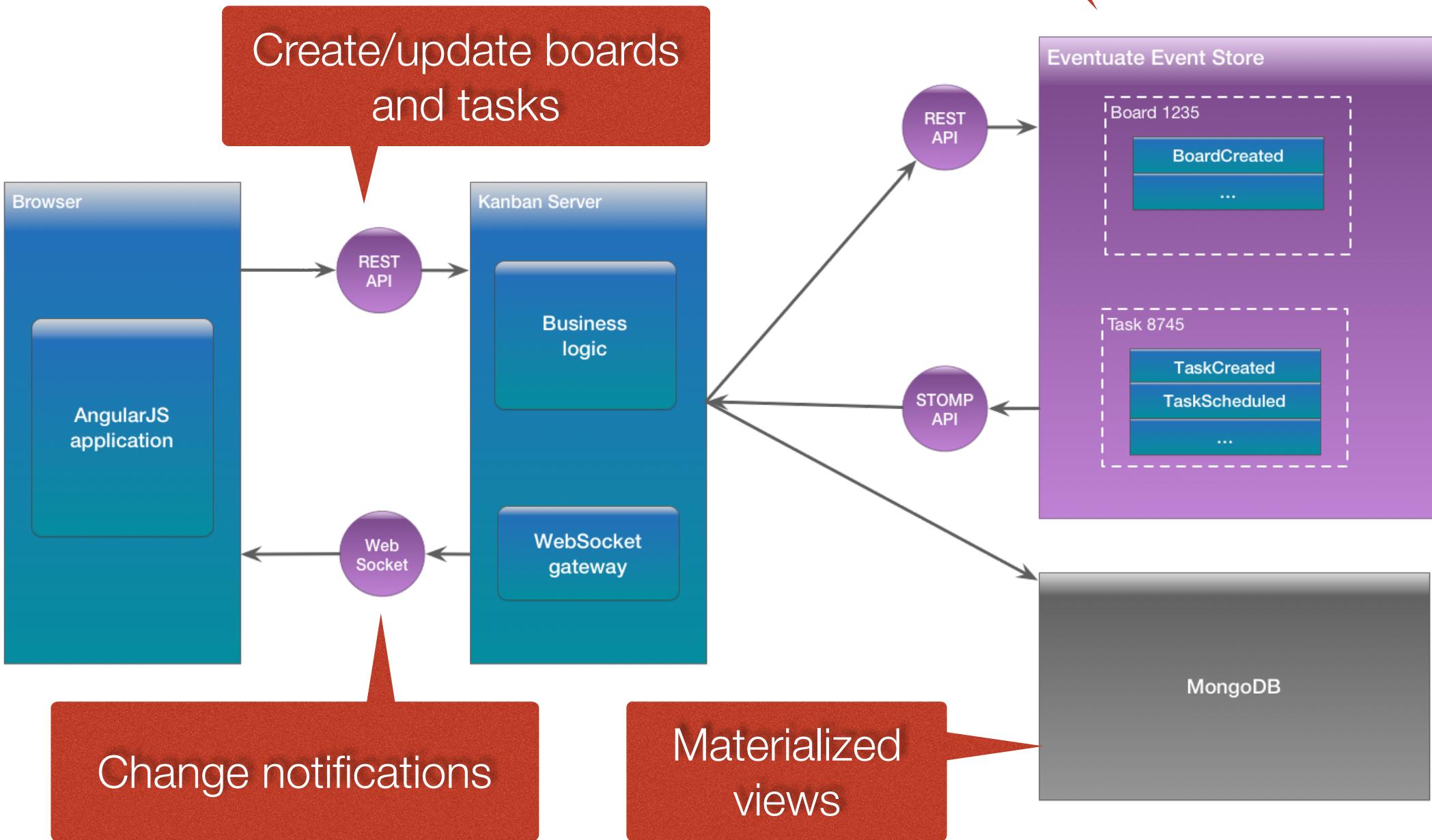
- To Do:** 0
- Doing:** 0
- Done:** 1
- Backlog:** 0

The 'Done' column contains one task card:

**create documentation** H x  
Created by:  
chris+kb1@chrisrichardson.net  
Changed less than a minute ago

A 'Create Task' button is located in the 'Backlog' area.

# Architecture



Chrome File Edit View History Bookmarks People Window Help

Chris Kanban Board App 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

Chris Kanban Board App 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

• @crichton chris@chrisrichardson.net

A close-up photograph of an emu's head, showing its large, dark brown eyes and hooked beak. The feathers around the eyes are dark and textured. The background is blurred green foliage.

Questions?

<http://plainoldobjects.com>

<http://microservices.io>

<http://eventuate.io>