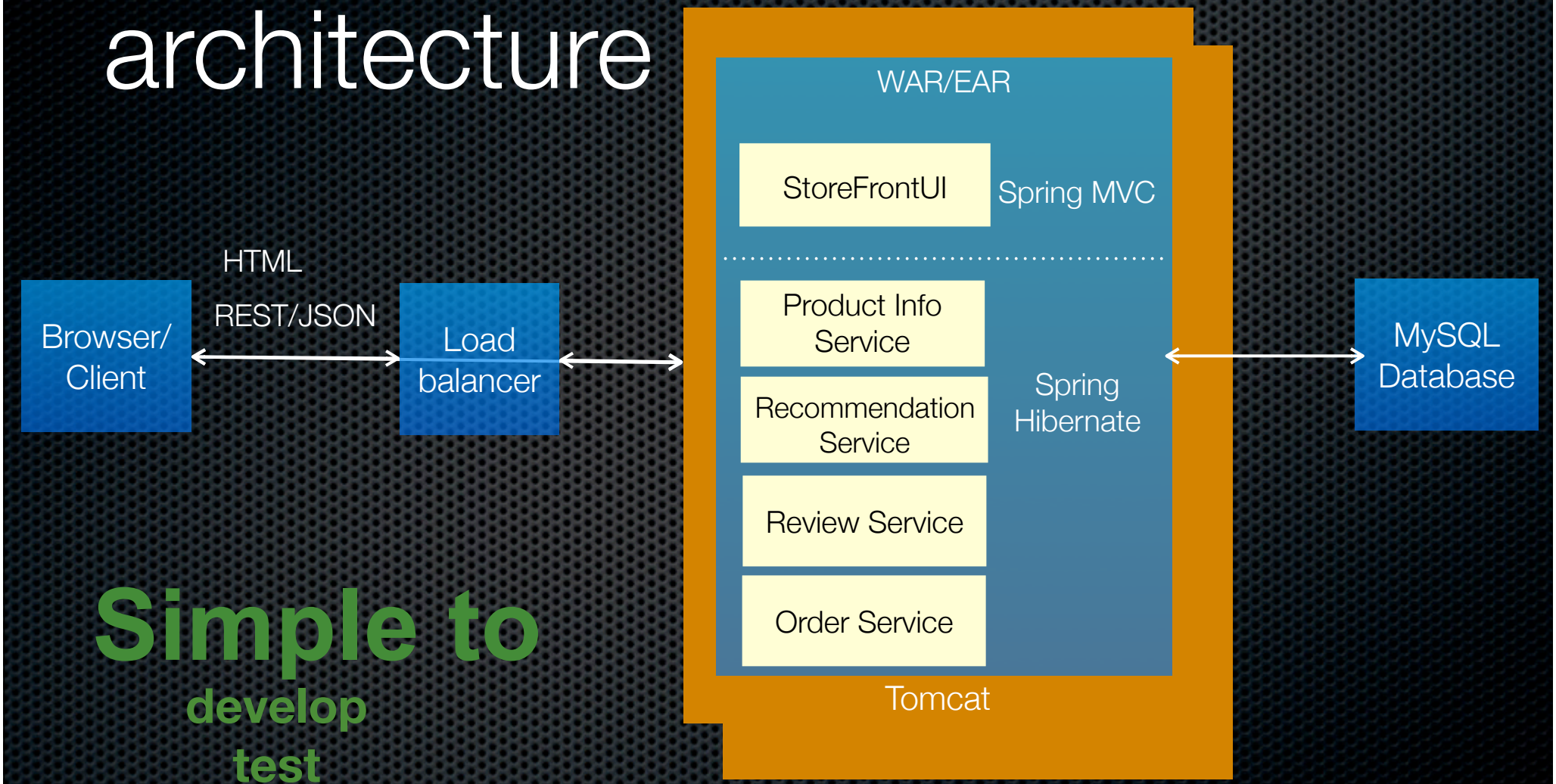
Let’s imagine you are building an online store

# Traditional application architecture




**Simple to**

**develop**

**test**

**deploy**

**scale**

But large, complex, monolithic applications

⇒

problems

# Intimidates developers

# Obstacle to frequent deployments

- Need to redeploy everything to change one component
- Interrupts long running background (e.g. Quartz) jobs
- Increases risk of failure

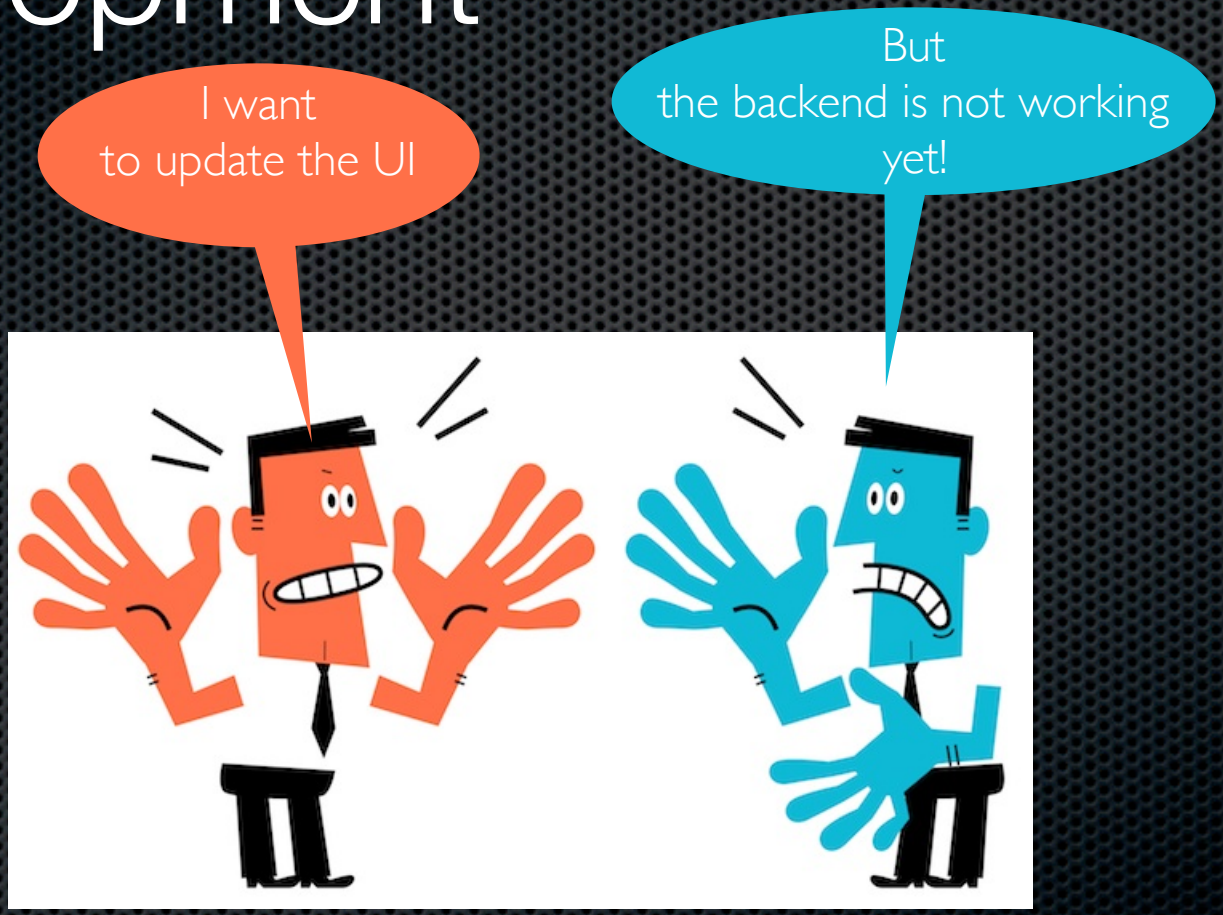
Eggs in one basket


- Updates will happen less often - really long QA cycles
- e.g. Makes A/B testing UI really difficult

# Overloads your IDE and container

Slows down development

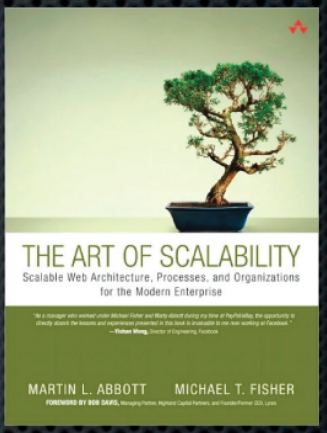
# Obstacle to scaling development




Lots of coordination and communication required

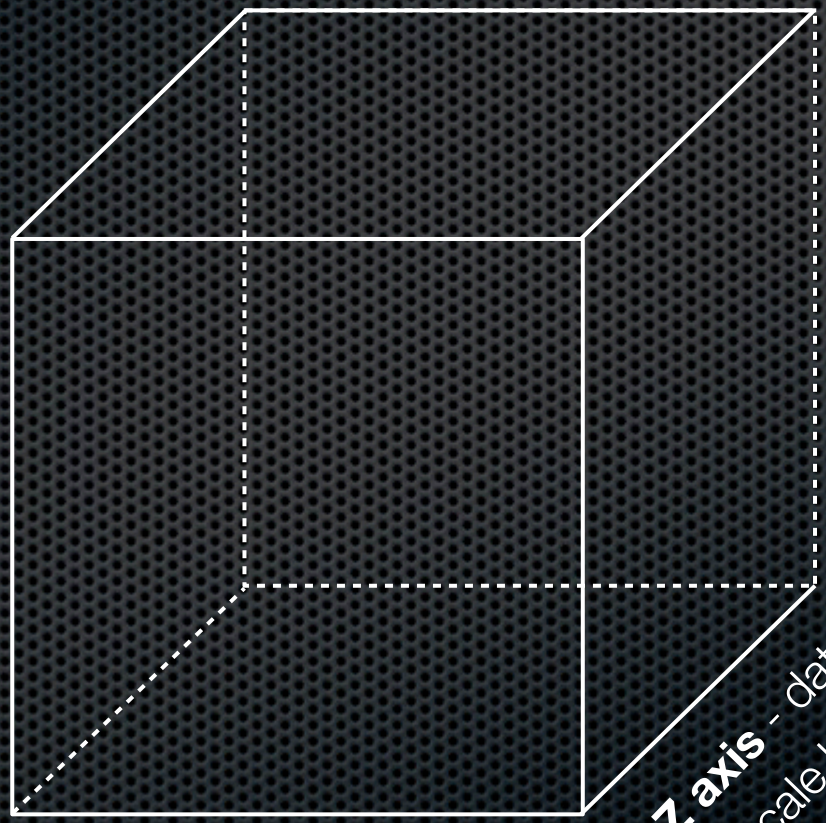
# Requires long-term commitment to a technology stack

# The scale cube




**Y axis** - functional decomposition

Scale by splitting different things



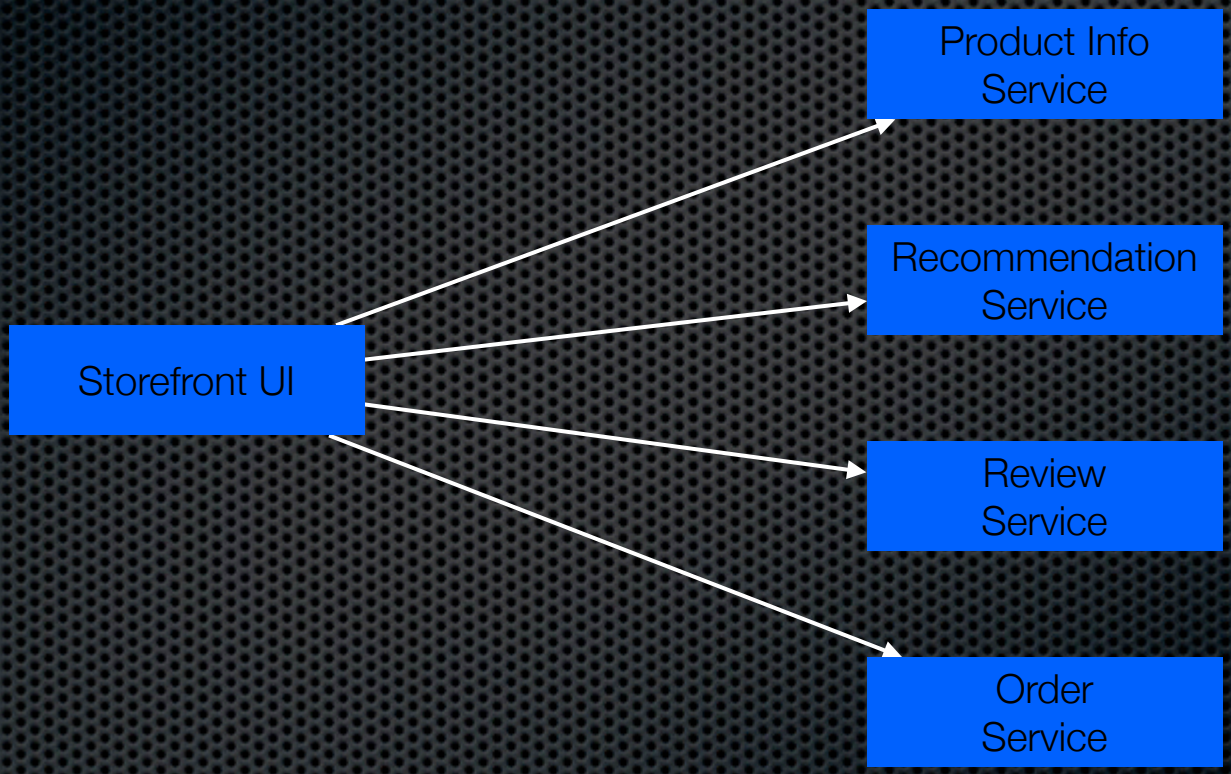
**Z axis** - data partitioning
Scale by splitting similar things

**X axis**
- horizontal duplication

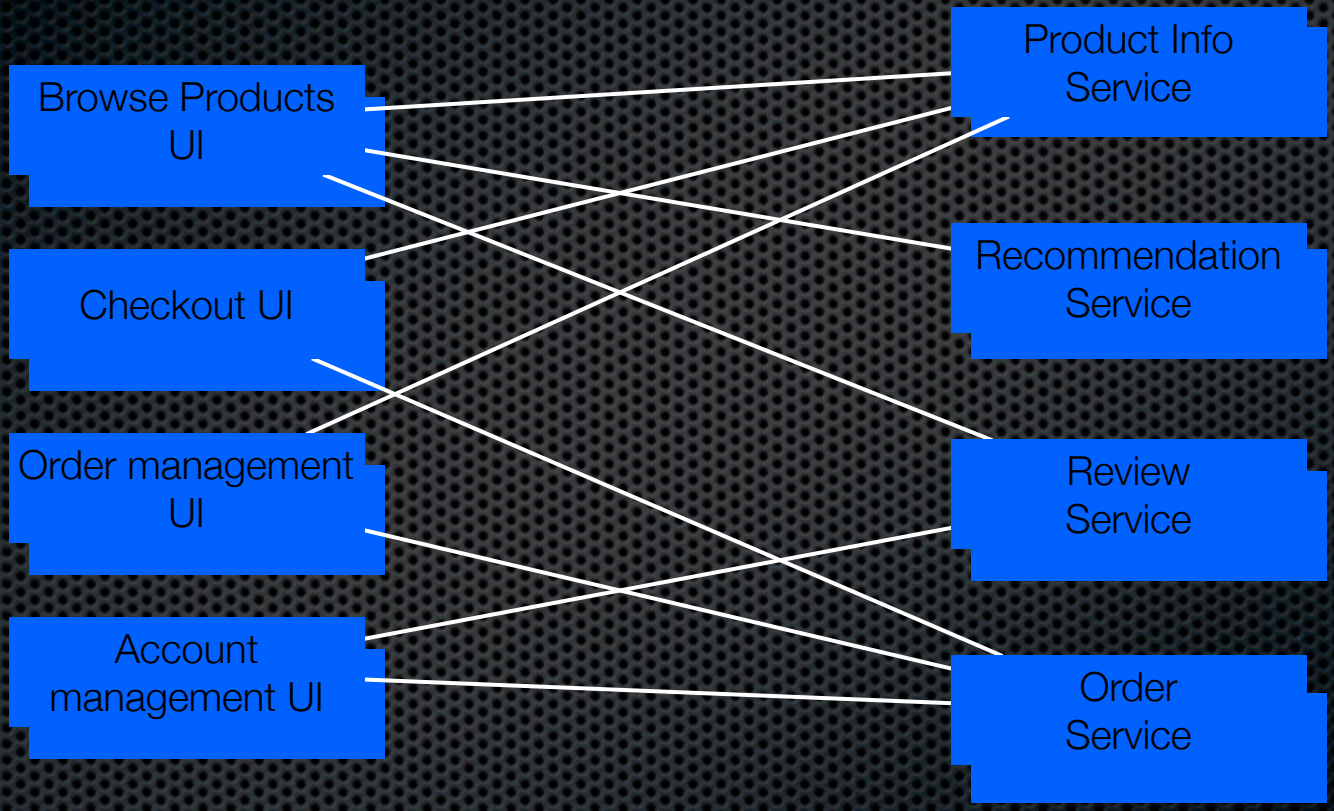
@crichardson

# Y-axis scaling - application level

# Y-axis scaling - application level



# Y-axis scaling - application level




Apply X-axis and Z-axis scaling to each service independently

# Service deployment options

Isolation, manageability

- VM or Physical Machine
- Docker/Linux container
- JVM
- JAR/WAR/OSGI bundle/...

Density/efficiency

# Partitioning strategies...

- Partition by noun, e.g. product info service
- Partition by verb, e.g. Checkout UI
- Single Responsibility Principle
- Unix utilities - do one focussed thing well

# Partitioning strategies

- Too few
  - Drawbacks of the monolithic architecture
- Too many - a.k.a. Nano-service anti-pattern
  - Runtime overhead
  - **Potential** risk of excessive network hops
  - **Potentially** difficult to understand system

Something of an art

# More service, less micro

But more realistically...

Focus on building services that make development and deployment easier

- not just tiny services

# Real world examples


http://techblog.netflix.com/

~600 services


http://highscalability.com/amazon-architecture

100-150 services to build a page


http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf

http://queue.acm.org/detail.cfm?id=1394128

There are many benefits

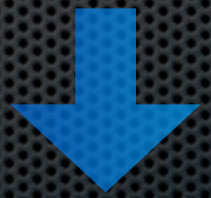
# Smaller, simpler apps

- Easier to understand and develop
- Less jar/classpath hell - who needs OSGI?
- Faster to build and deploy
- Reduced startup time - important for GAE

Scales development:
develop, deploy and scale
each service independently

Improves fault isolation

Eliminates long-term commitment to a single technology stack



Modular, polyglot, multi-framework applications

# Two levels of architecture

**System-level**

Services
Inter-service glue: interfaces and communication mechanisms
Slow changing

**Service-level**

Internal architecture of each service
Each service could use a different technology stack
Pick the best tool for the job
Rapidly evolving

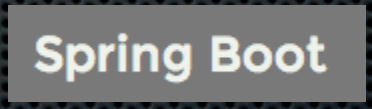
# Easily **try** other technologies










... and fail safely

# But there are drawbacks

# Complexity

# Complexity of developing a distributed system

http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

# Multiple databases
# &
# Transaction management

e.g. Fun with eventual consistency

# Complexity of testing a distributed system

http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

# Complexity of deploying and operating a distributed system

http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

You need a lot of automation

Developing and deploying features that span multiple services requires careful coordination

# When to use it?

**In the beginning:**

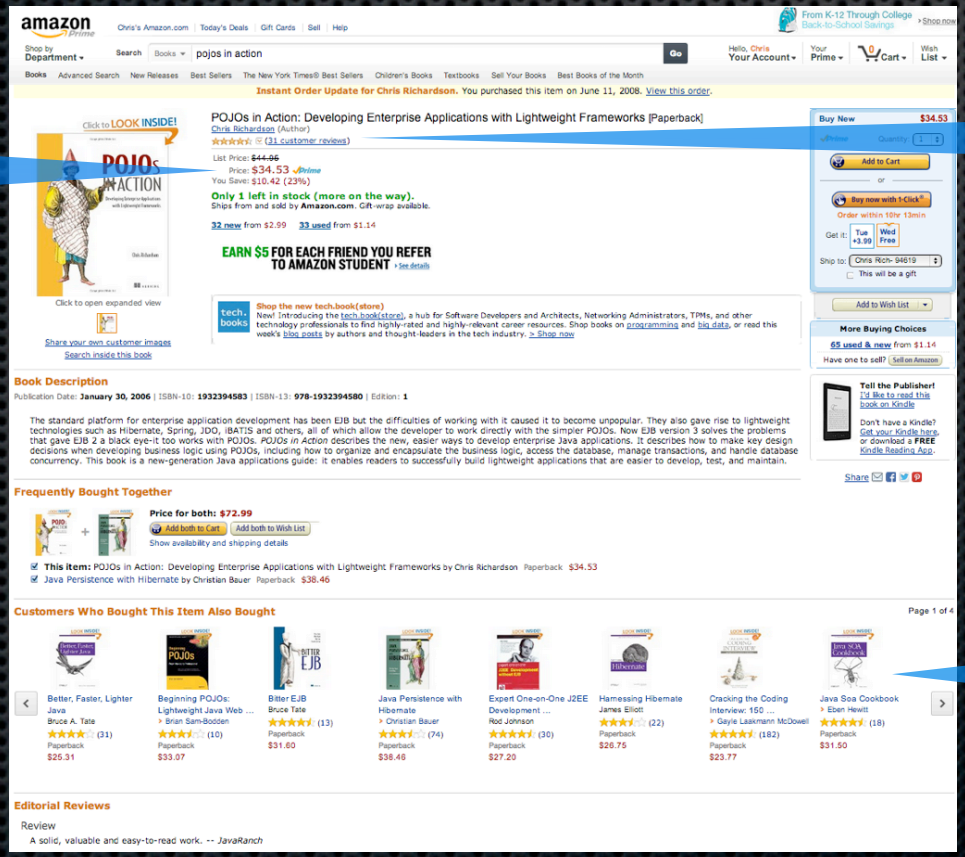
- You don't need it
- It will slow you down

**Later on:**

- You need it
- Refactoring is painful

# Let's imagine that you want to display a product's details...

Product Info

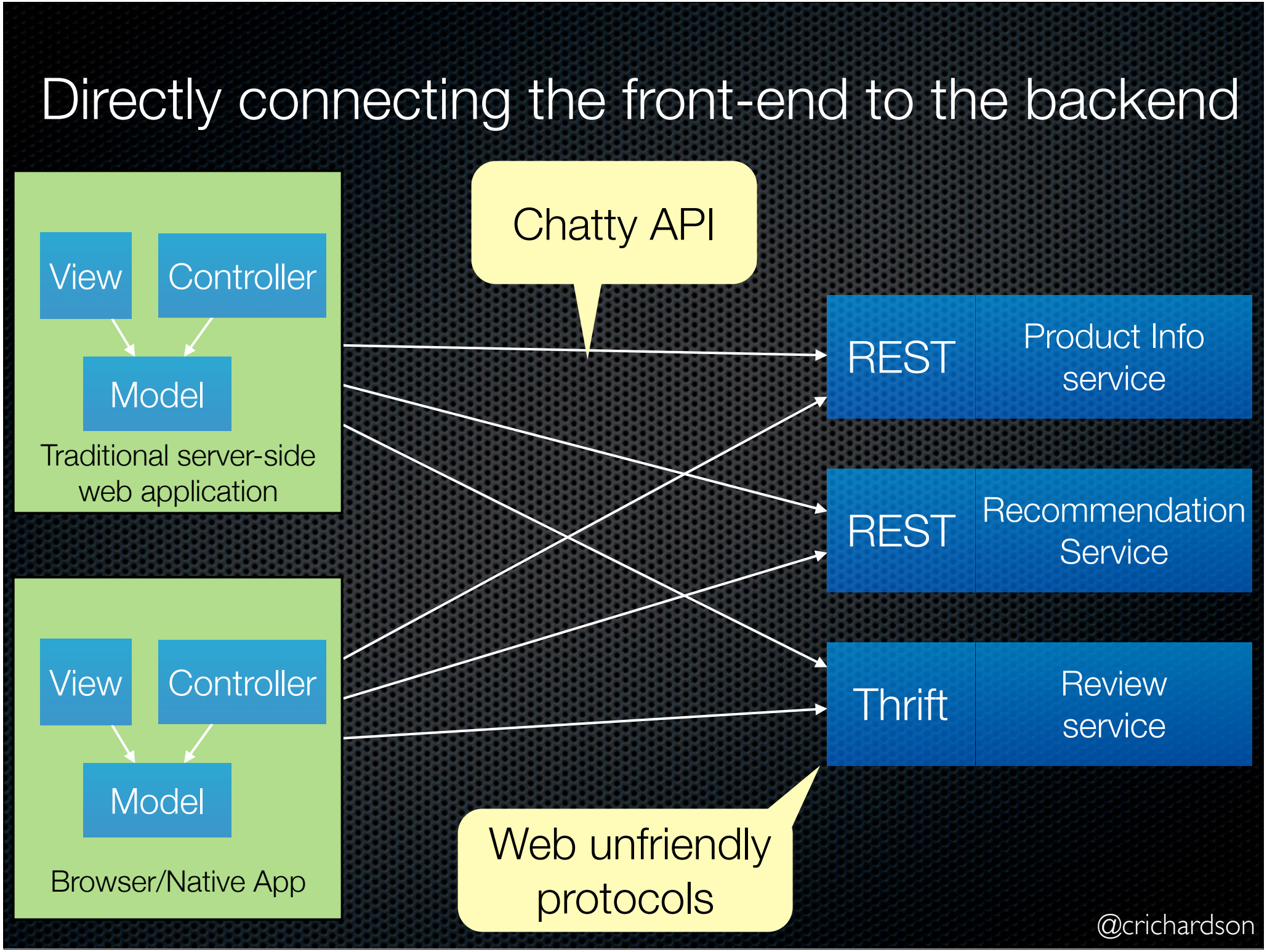



Reviews

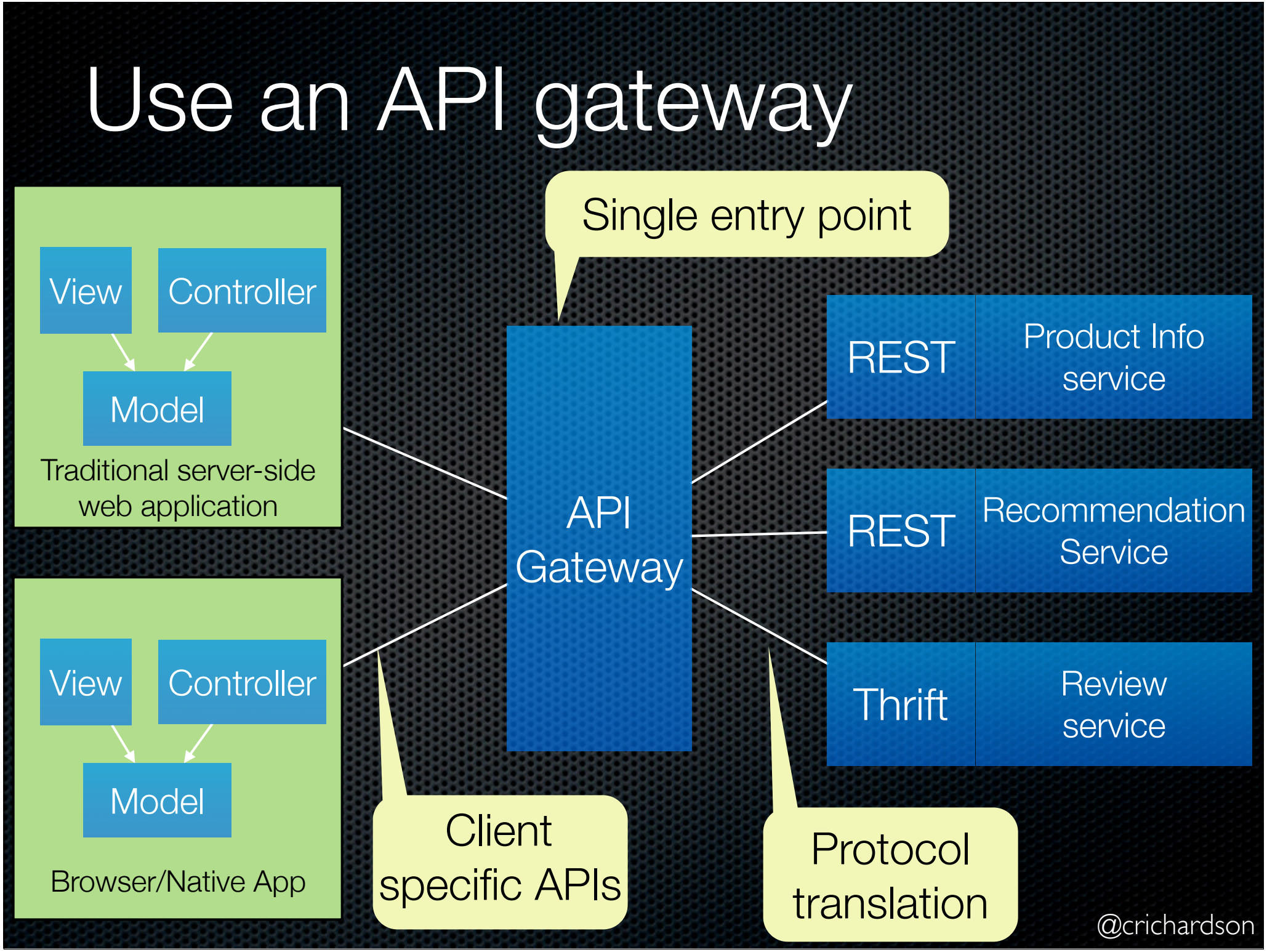
Recommendations

@crichardson

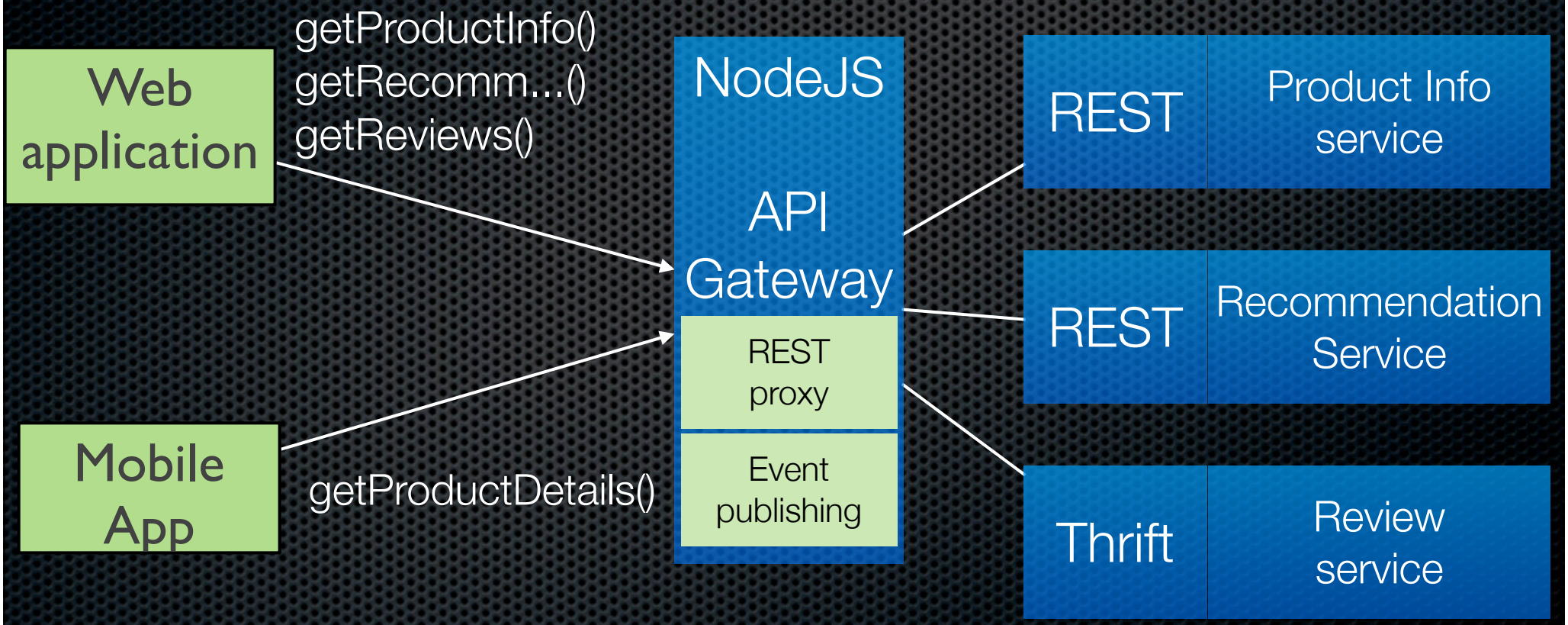
# Directly connecting the front-end to the backend



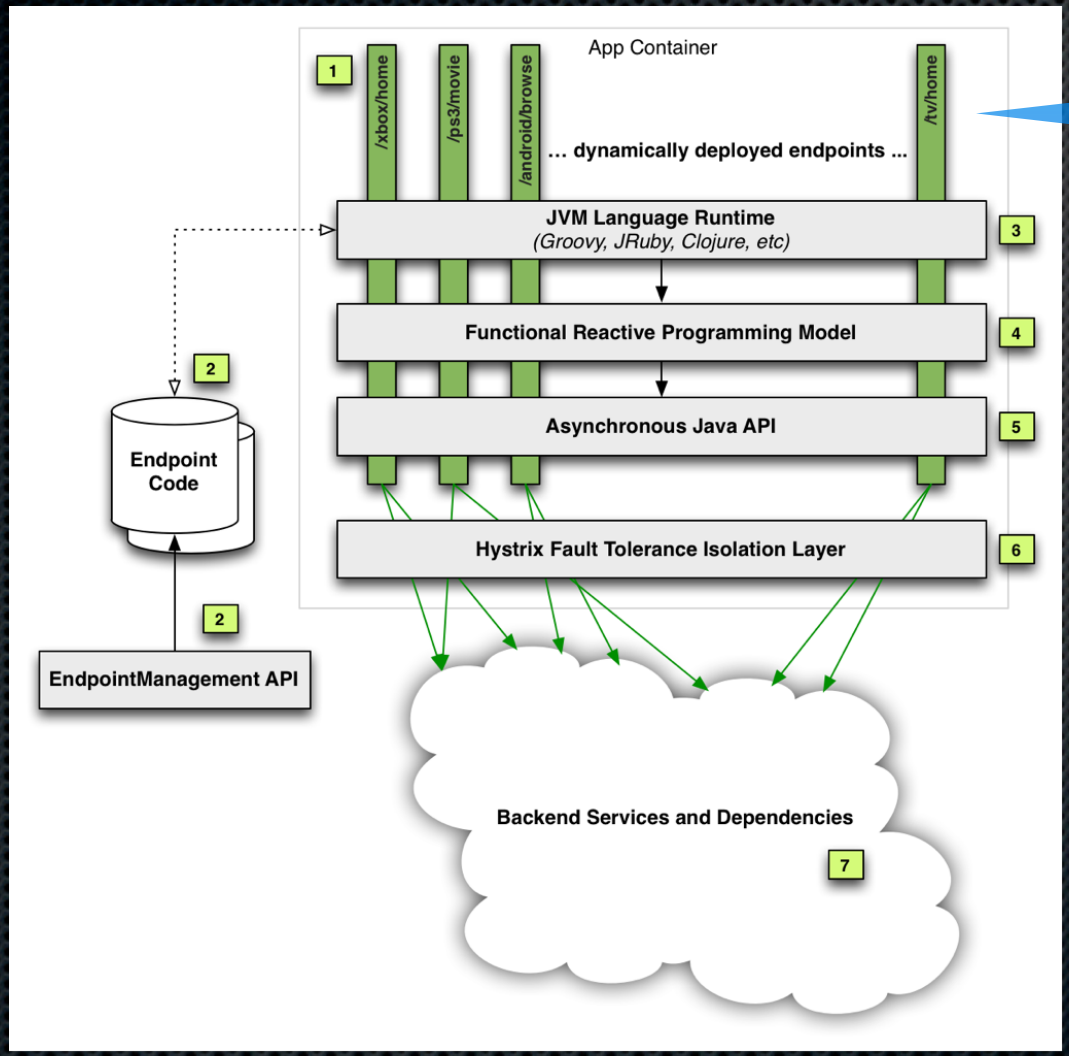
# Use an API gateway



# Optimized client-specific APIs



# Netflix API Gateway




Device specific end points

http://techblog.netflix.com/2013/01/optimizing-netflix-api.html

# API gateway design challenges

- Performance and scalability
  - Non-blocking I/O
  - Asynchronous, concurrent code
- Handling partial failures
  - ....

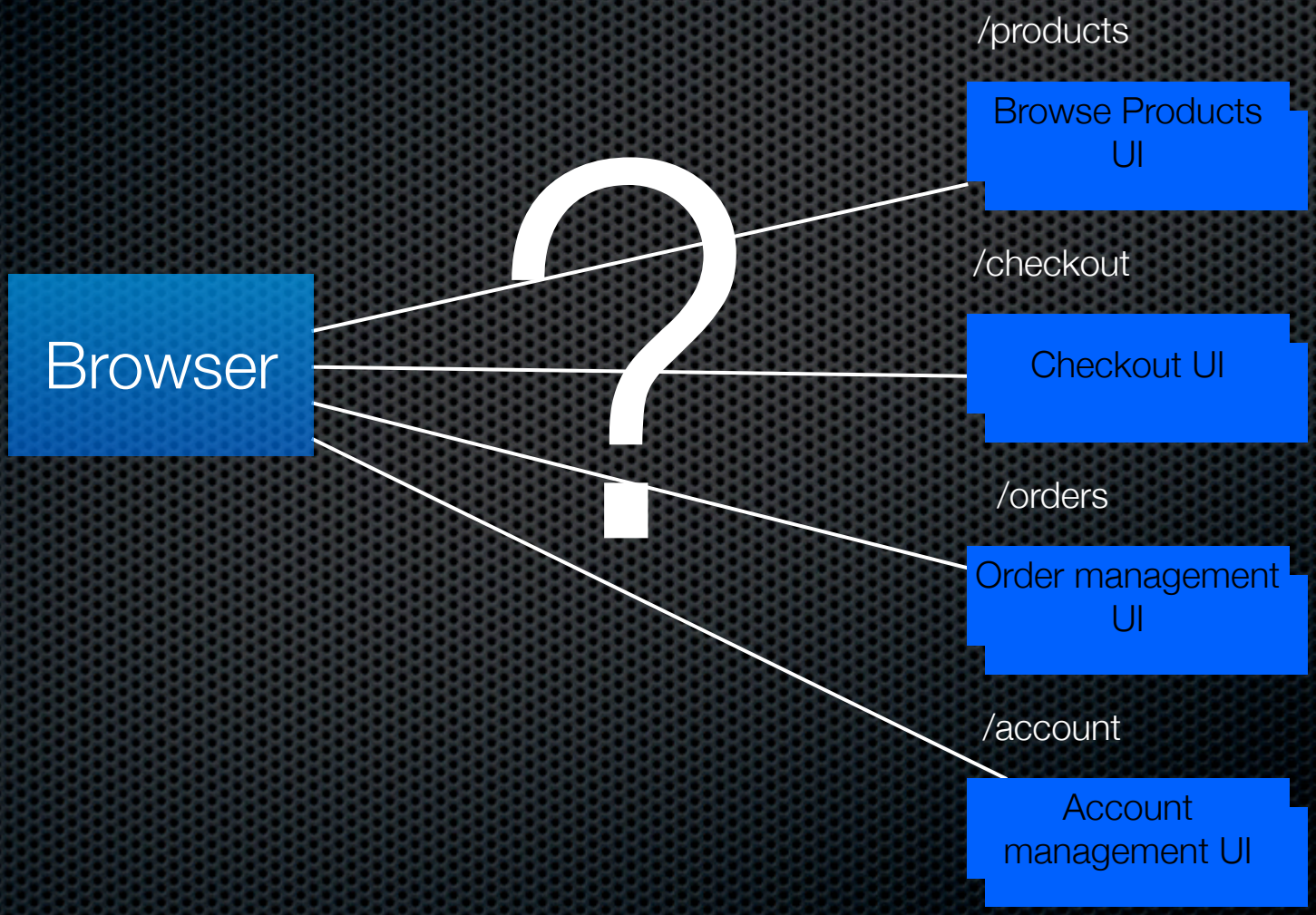
http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

# Useful frameworks for building an API gateway

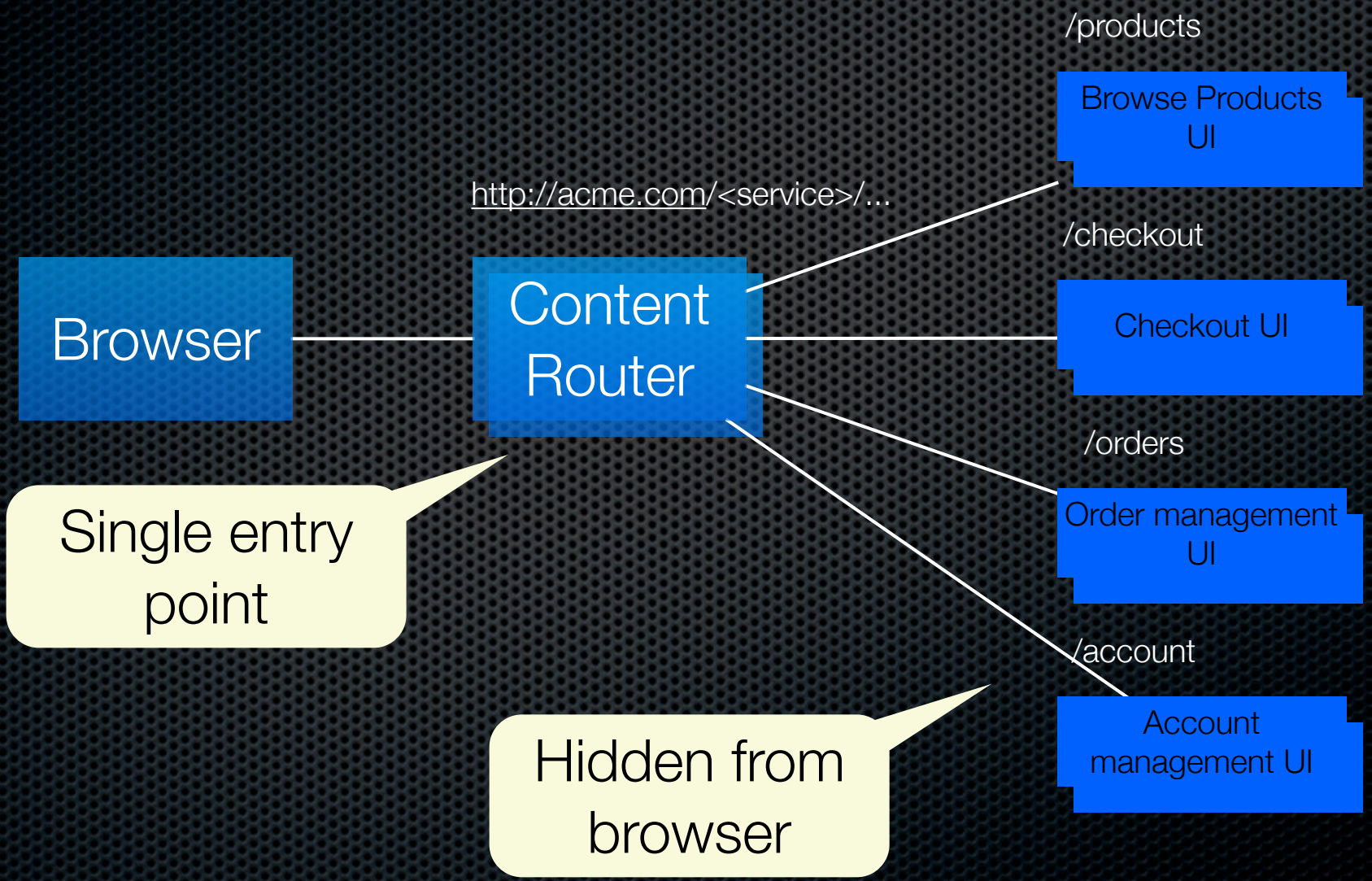
- JVM:
  - Netty, Vertex
  - Netflix Hystrix
  - ...
- Other:
  - NodeJS

How does a browser interact with the partitioned web application?

# Partitioned web app ⇒ no longer a single base URL



# The solution: single entry point that routes based on URL



# How do the services communicate?

# Inter-service communication options

- Synchronous HTTP ⇔ asynchronous AMQP
- Formats: JSON, XML, Protocol Buffers, Thrift, ...

Asynchronous is preferred
JSON is fashionable but binary format is more efficient

# Pros and cons of messaging

**Pros**

- Decouples client from server
- Message broker buffers messages
- Supports a variety of communication patterns

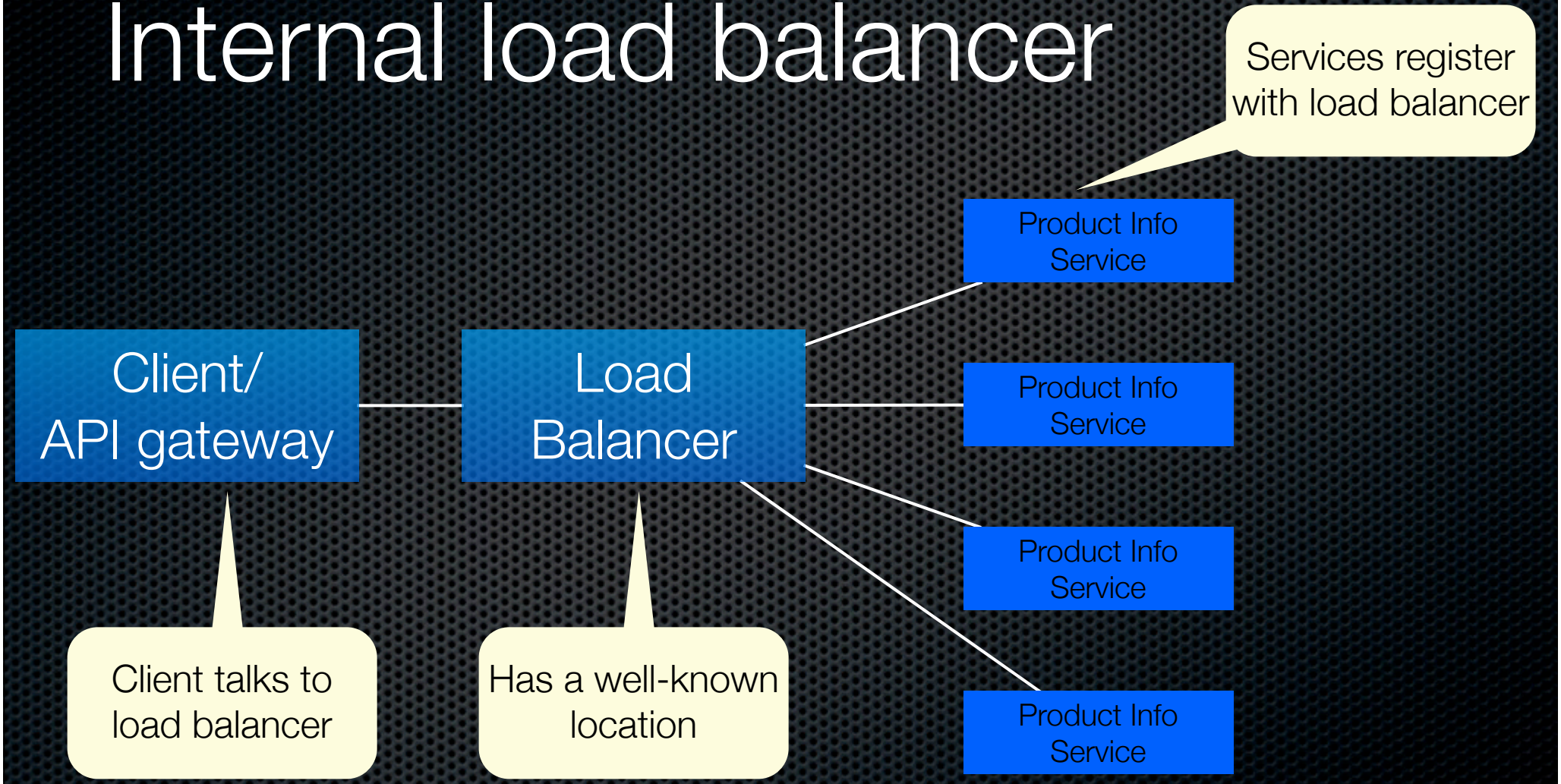
**Cons**

- Additional complexity of message broker
- Request/reply-style communication is more complex

# Pros and cons of HTTP

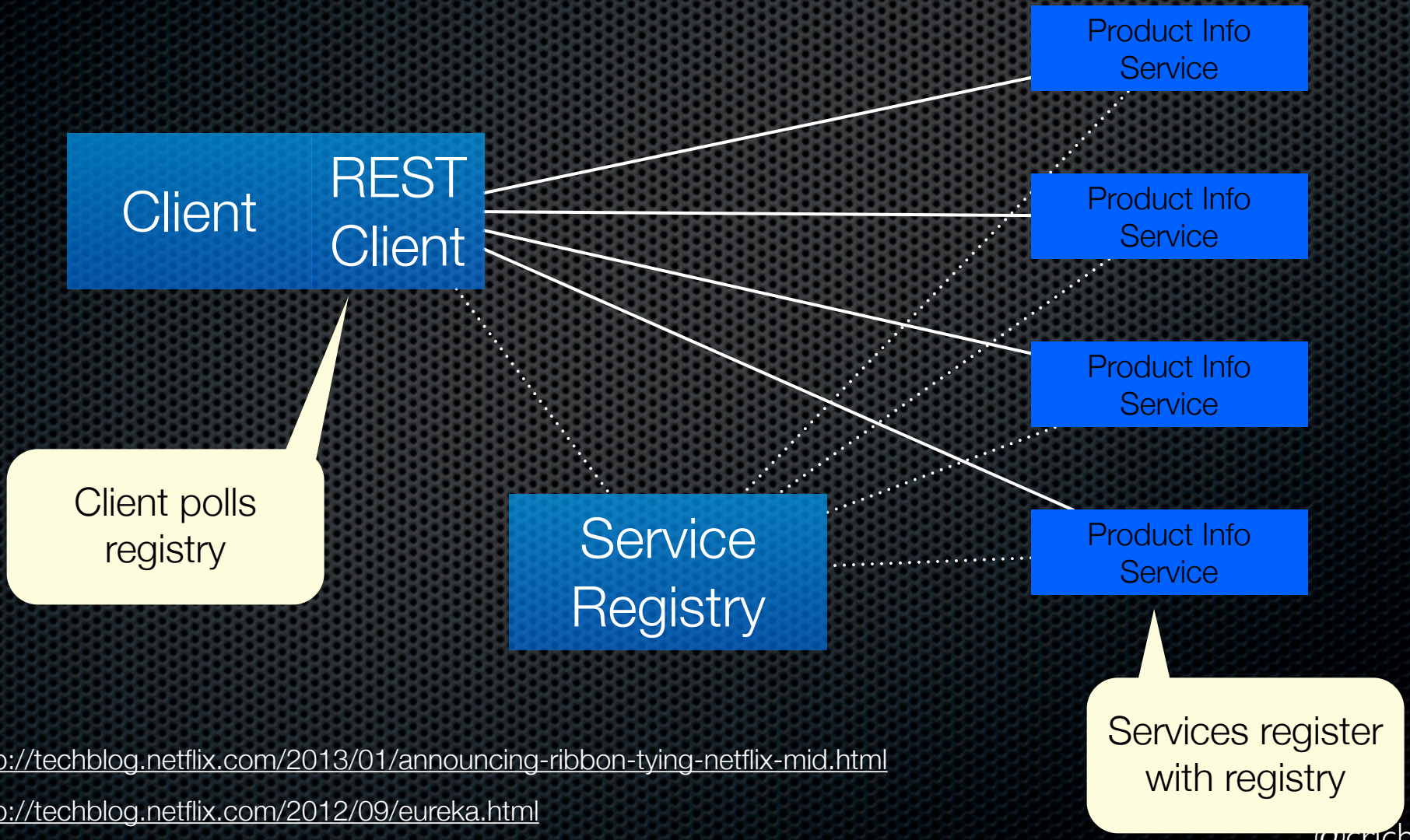
- Pros
  - Simple and familiar
  - Request/reply is easy
  - Firewall friendly
  - No intermediate broker
- Cons
  - Only supports request/reply
  - Server must be available
  - Client needs to discover URL(s) of server(s)

# Discovery option #1: Internal load balancer




http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/USVPC_creating_basic_lb.html

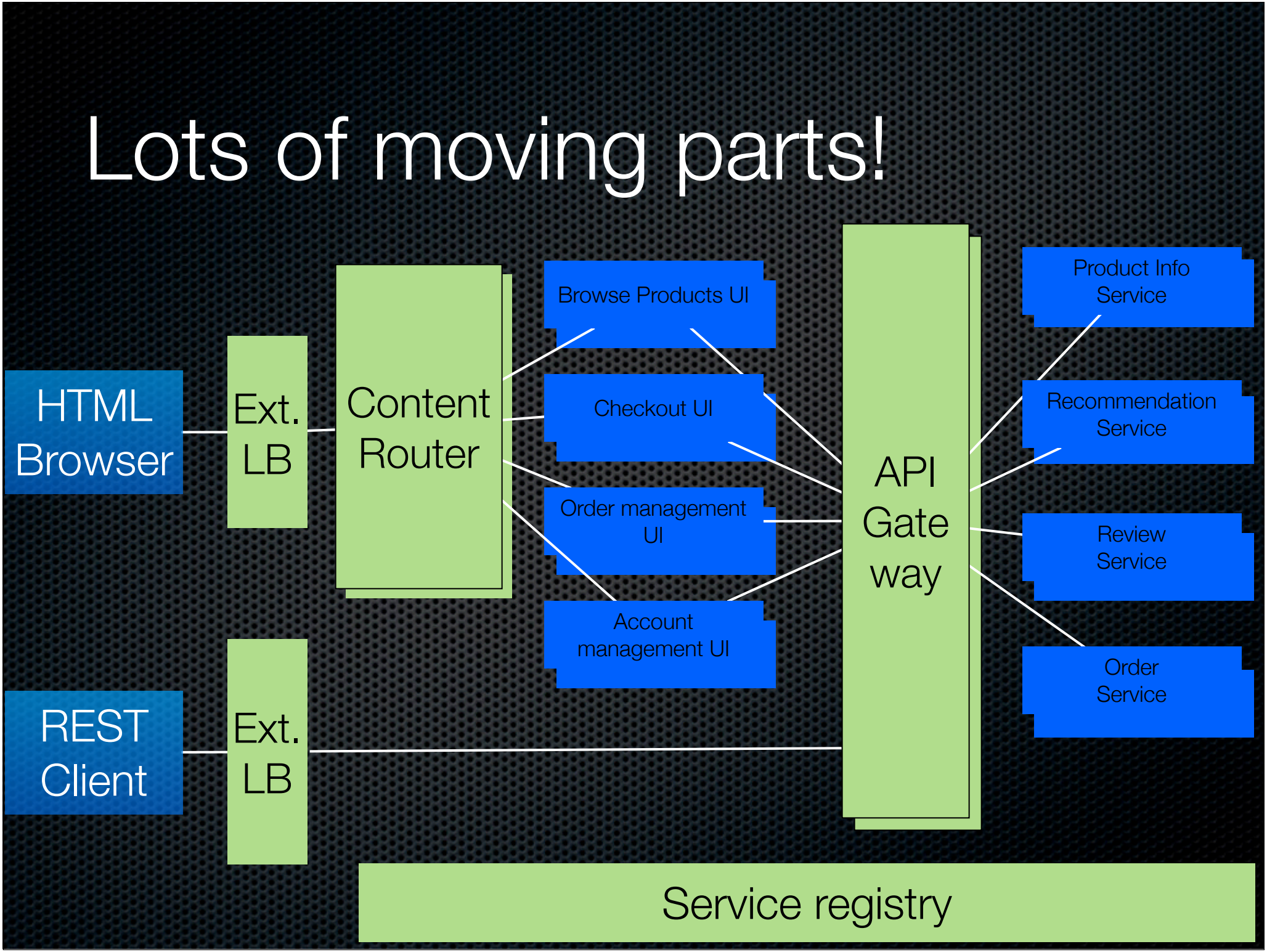
# Discovery option #2: client-side load balancing




http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html

http://techblog.netflix.com/2012/09/eureka.html

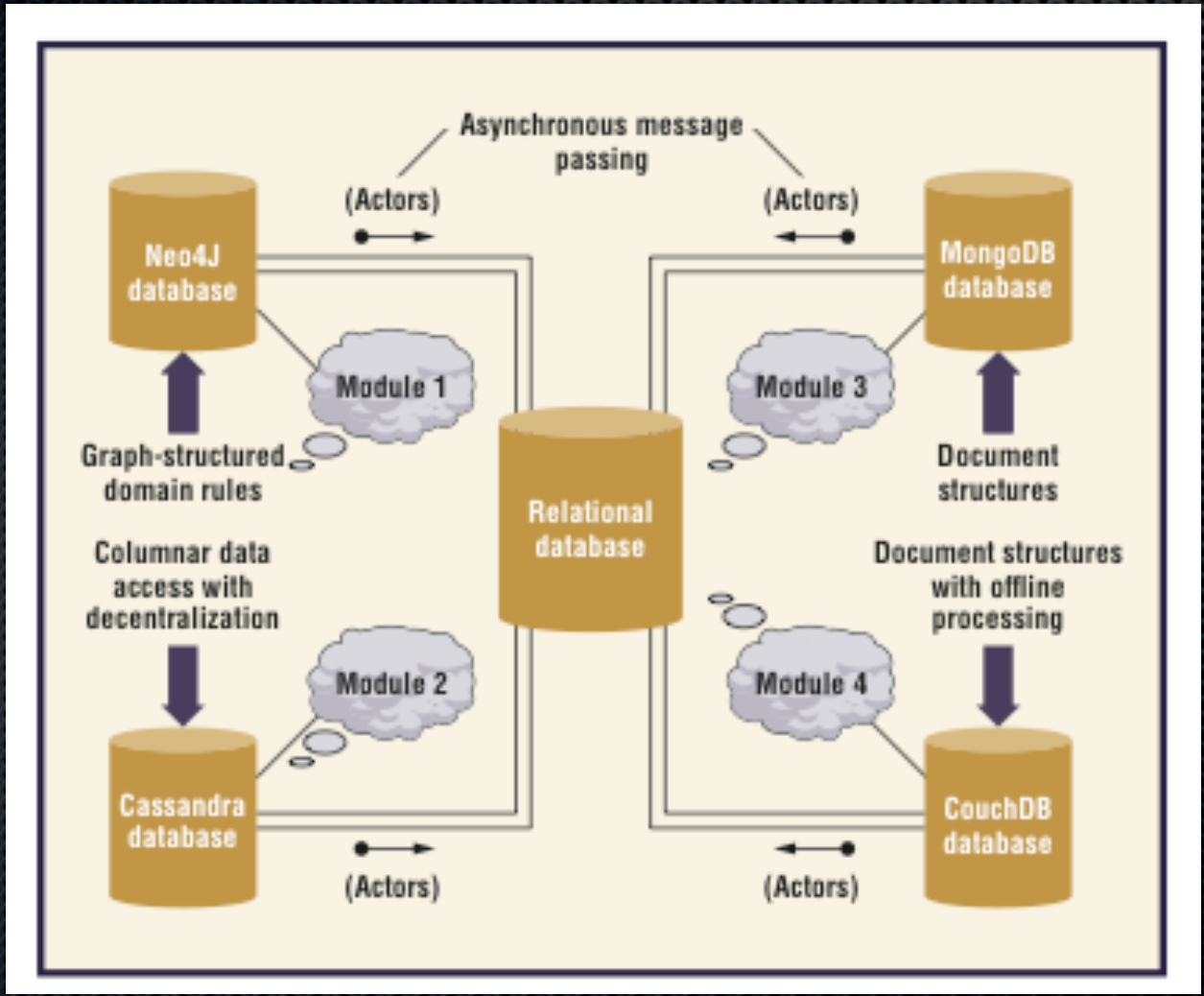
# Lots of moving parts!



# Decomposed services ⇒ decomposed databases

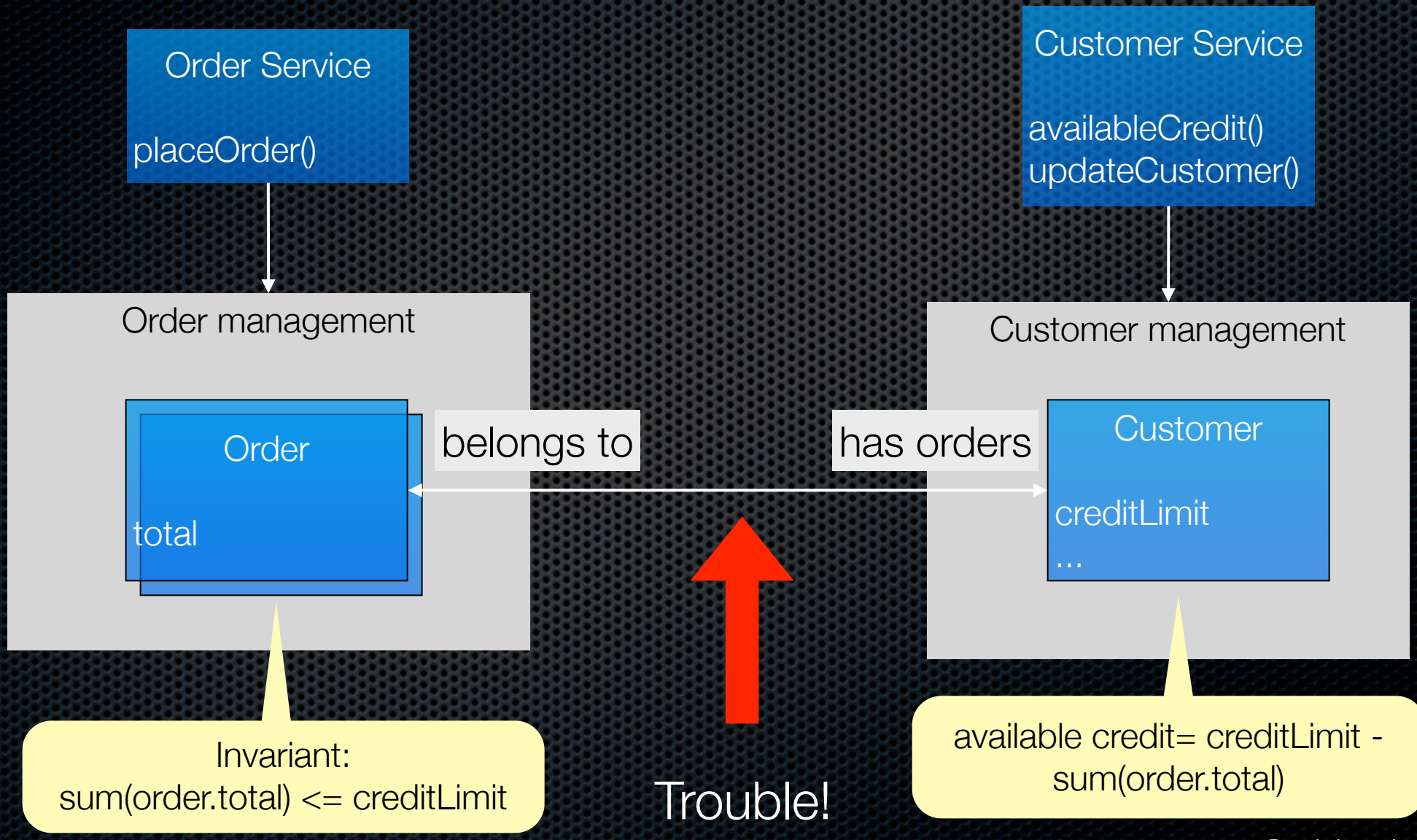

Separate databases ⇒ less coupling

# Decomposed databases ⇒ polyglot persistence




IEEE Software Sept/October 2010 - Debasish Ghosh / Twitter @debasishg

# Untangling orders and customers



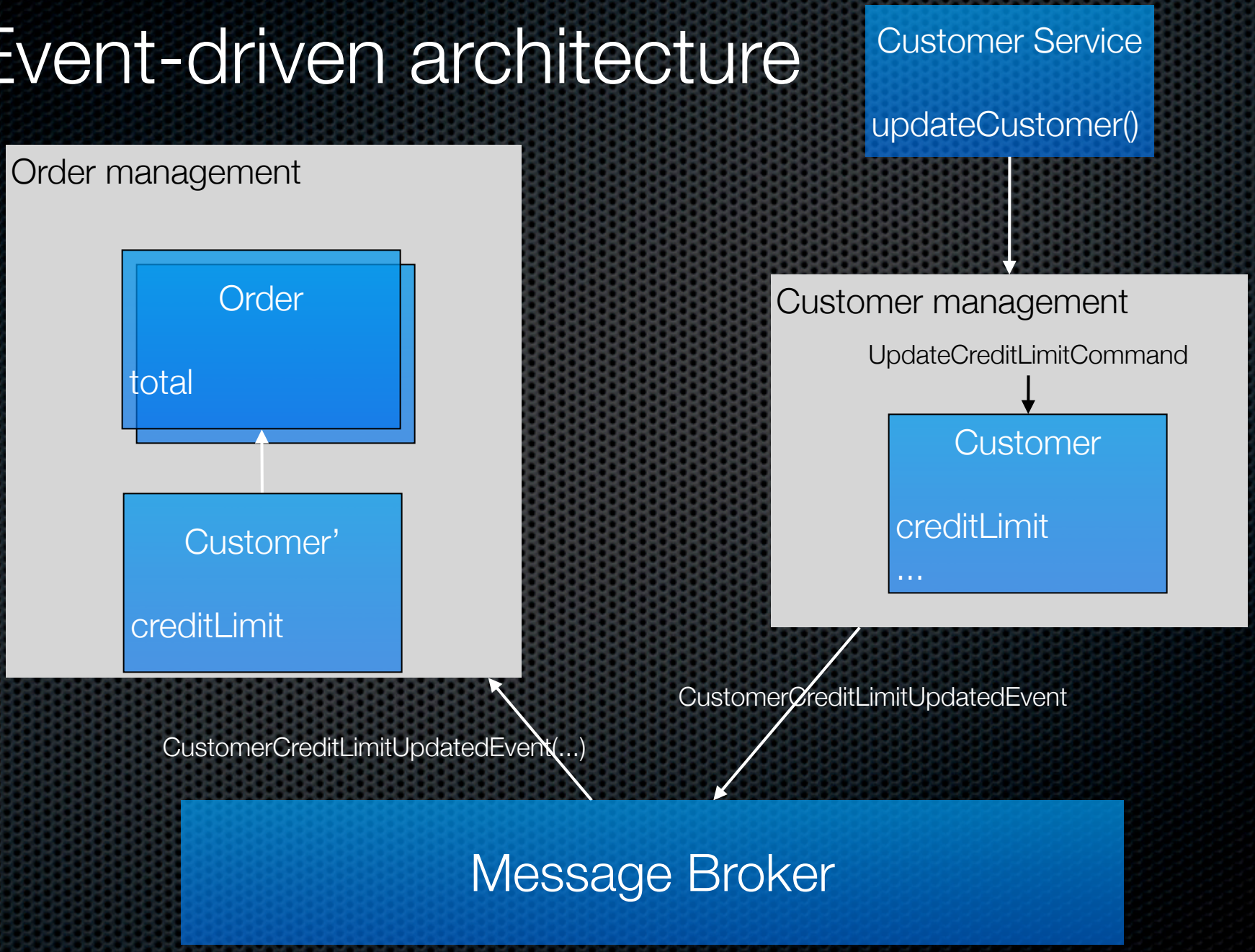
# Problems

- Reads
  - Service A needs to read data owned by service B
  - e.g. OrderService needs customer's credit limit
  - e.g Customer Service needs outstanding order total
- Updates
  - "Business transaction" must update data owned by multiple services
  - e.g Creating a User => Create their ShoppingCart

You could use two-phase commit (2PC) but ...

# Use an event-driven architecture

# Event-driven architecture



# Benefits and drawbacks of an event-driven architecture

- Benefits:
  - Simple infrastructure, e.g. a message broker
  - Better availability (than using 2PC)
- Drawbacks:
  - Application has to handle eventually consistent data
  - Application has to handle duplicate events

# How do services publish events?

To maintain consistency the application must **atomically** publish an event **whenever** a domain object changes

# How to generate events reliably?

- Database triggers
- Hibernate event listener
- Ad hoc event publishing code mixed into business logic
- Domain events - “formal” modeling of events
- Event Sourcing

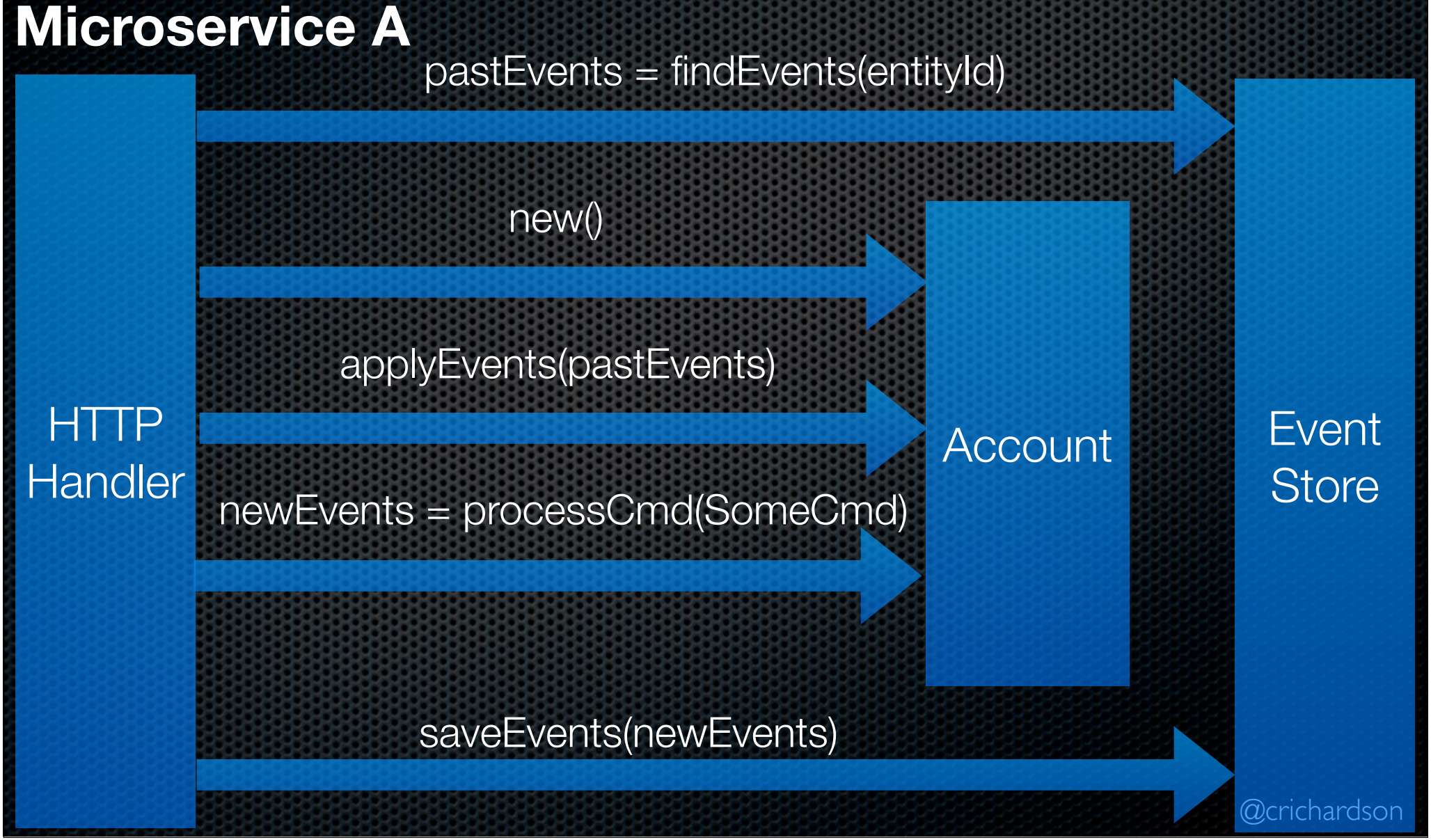
# Atomically publishing events

- Use distributed transactions to update database **and** publish to message
  - Database and message broker must support 2PC
  - 2PC is best avoided
- Use eventual consistency mechanism:
  1. Update database: new entity state & event to publish
  2. Publish event  & mark event as published
  - Difficult to implement when using a NoSQL database :-(

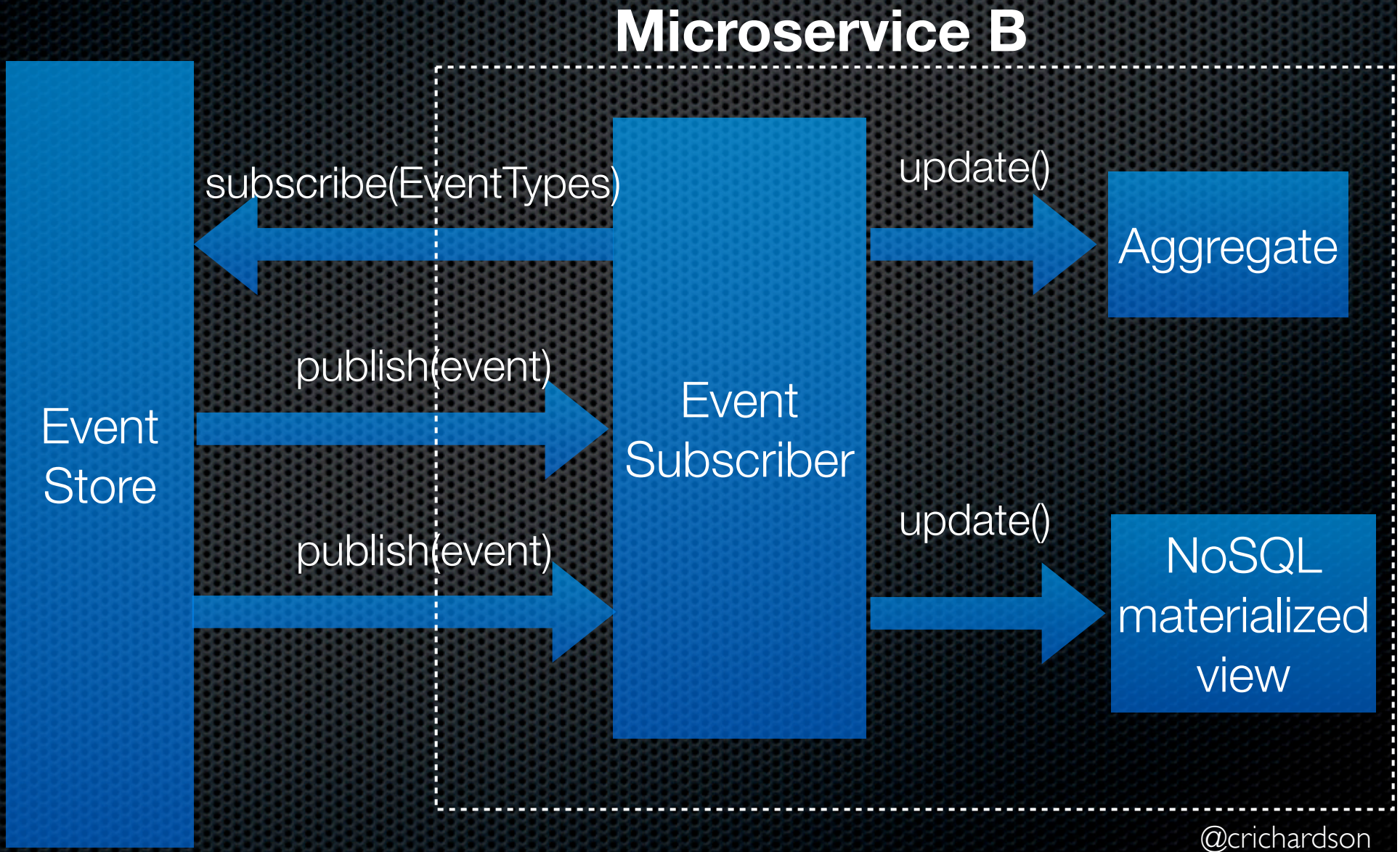
# Event sourcing

- An event-centric approach to designing domain models
  - Request becomes a command that is sent to an Aggregate
  - Aggregates handle commands by generating events
  - Apply events to an aggregate to update state
- Persist events **NOT** state
  - Replay events to recreate the current state of an aggregate
- Event Store ≃ database + message broker

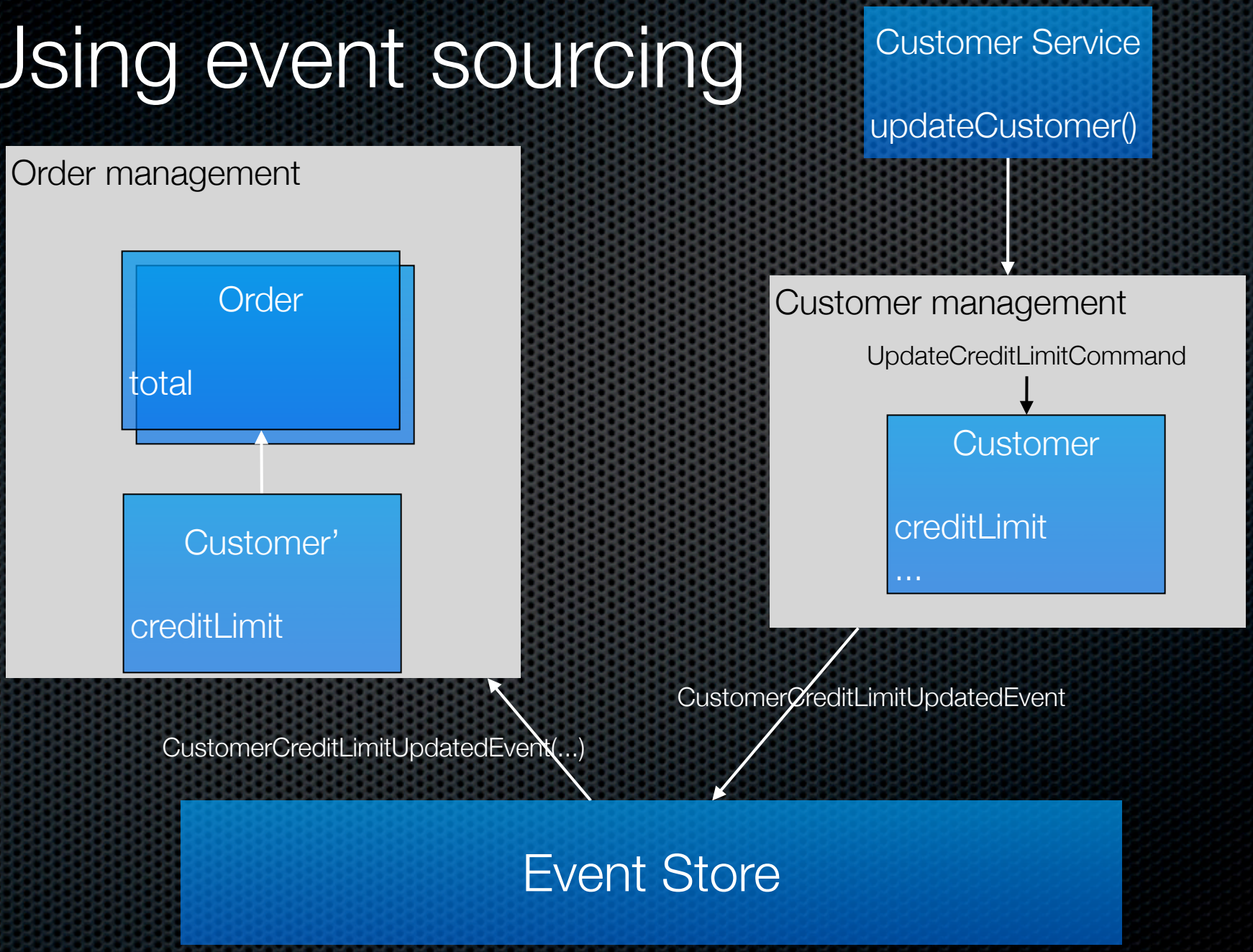
# Request handling in an event-sourced application



# Event Store publishes events - consumed by other services



# Using event sourcing



# Unfamiliar but it solves many problems

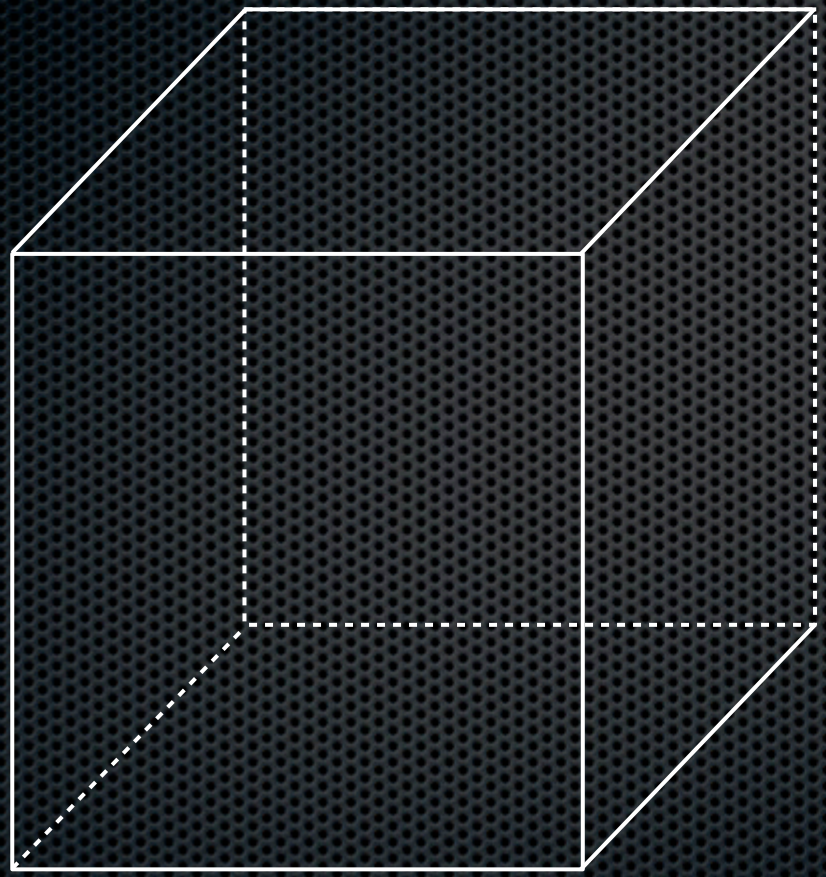
- Eliminates O/R mapping problem
- Supports both SQL and NoSQL databases
- Publishes events reliably
- Reliable eventual consistency framework
- ...

# Summary

Monolithic applications are simple to develop and deploy

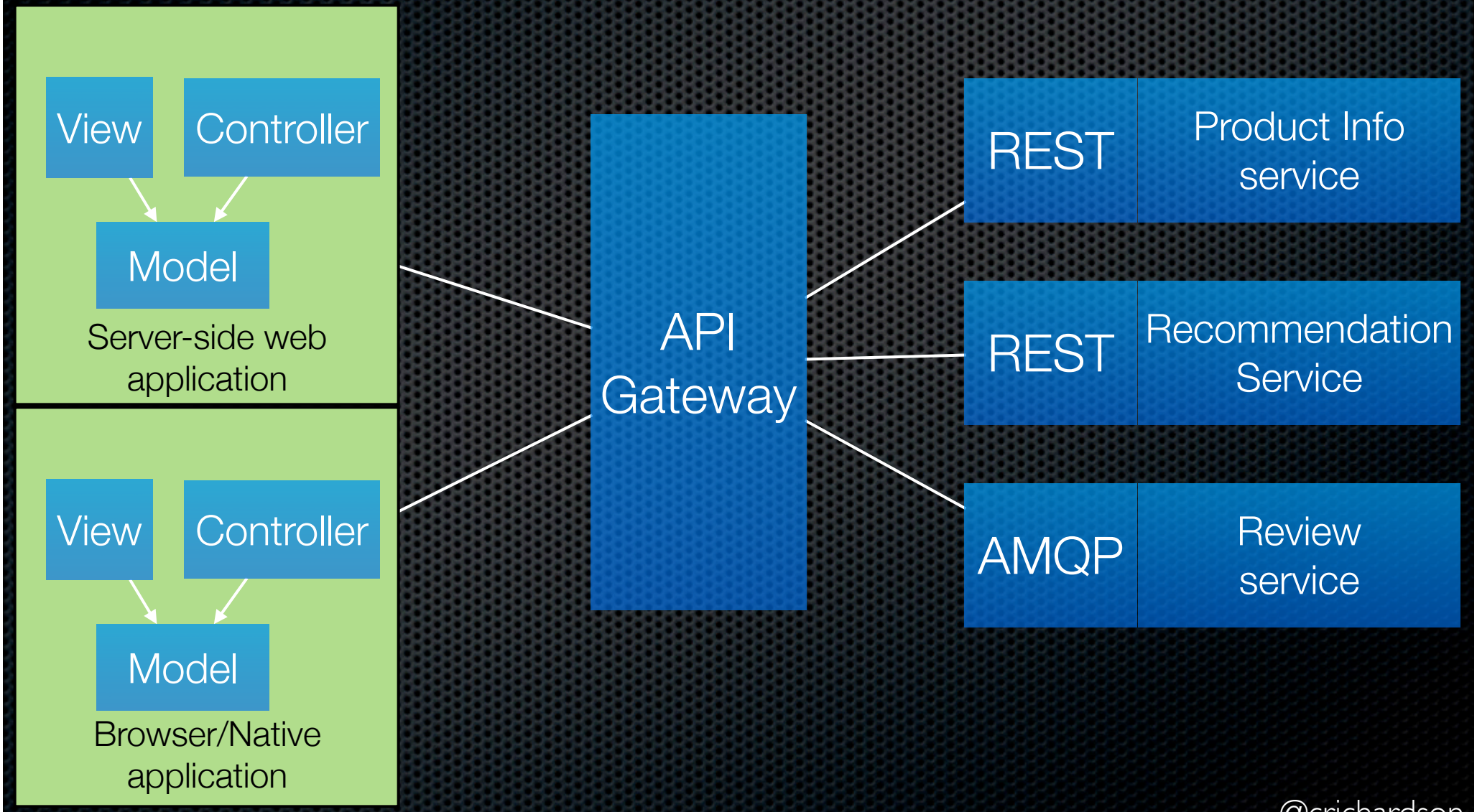
BUT have significant drawbacks

# Apply the scale cube



- Modular, polyglot, and scalable applications
- Services developed, deployed and scaled independently

# Use a modular, polyglot architecture



# Use an event-driven architecture

# Start refactoring your monolith