Microservices: Decomposing Applications for Deployability and Scalability

Chris Richardson

Author of POJOs in Action

Founder of the original CloudFoundry.com

@crichardson

chris@chrisrichardson.net

http://plainoldobjects.com

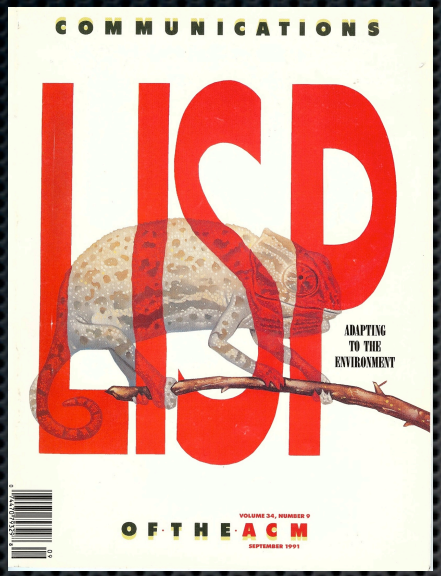
Presentation goal

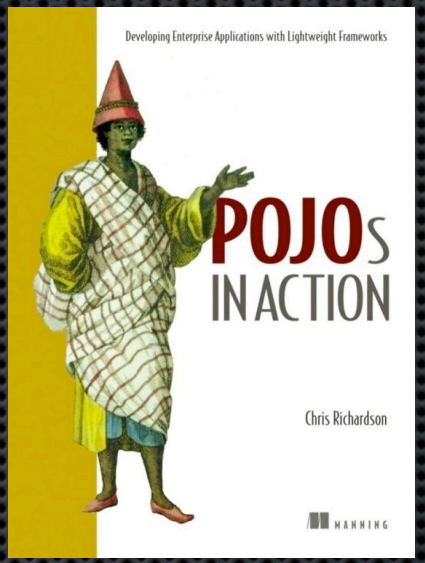
How decomposing applications improves deployability and scalability

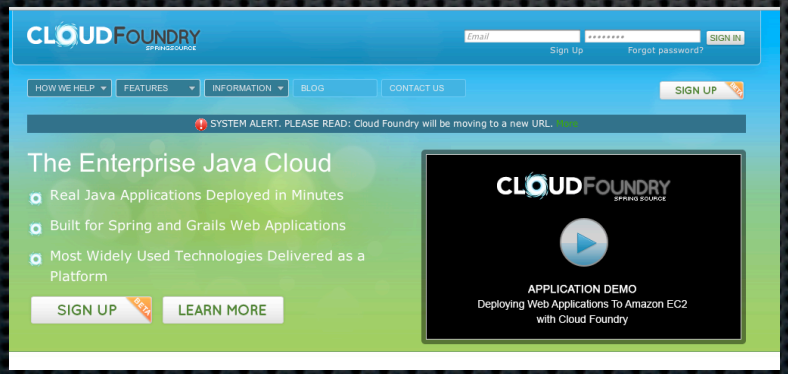
and

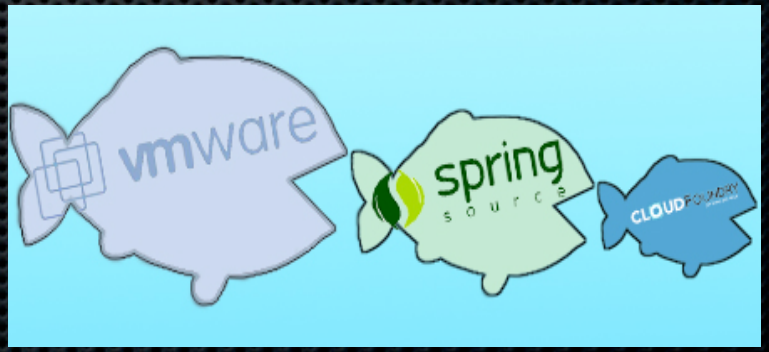
simplifies the adoption of new technologies

About Chris


COMMUNICATIONS
LISP
ADAPTING TO THE ENVIRONMENT
OF THE ACM
SEPTEMBER 1991


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


CLOUDFOUNDRY
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

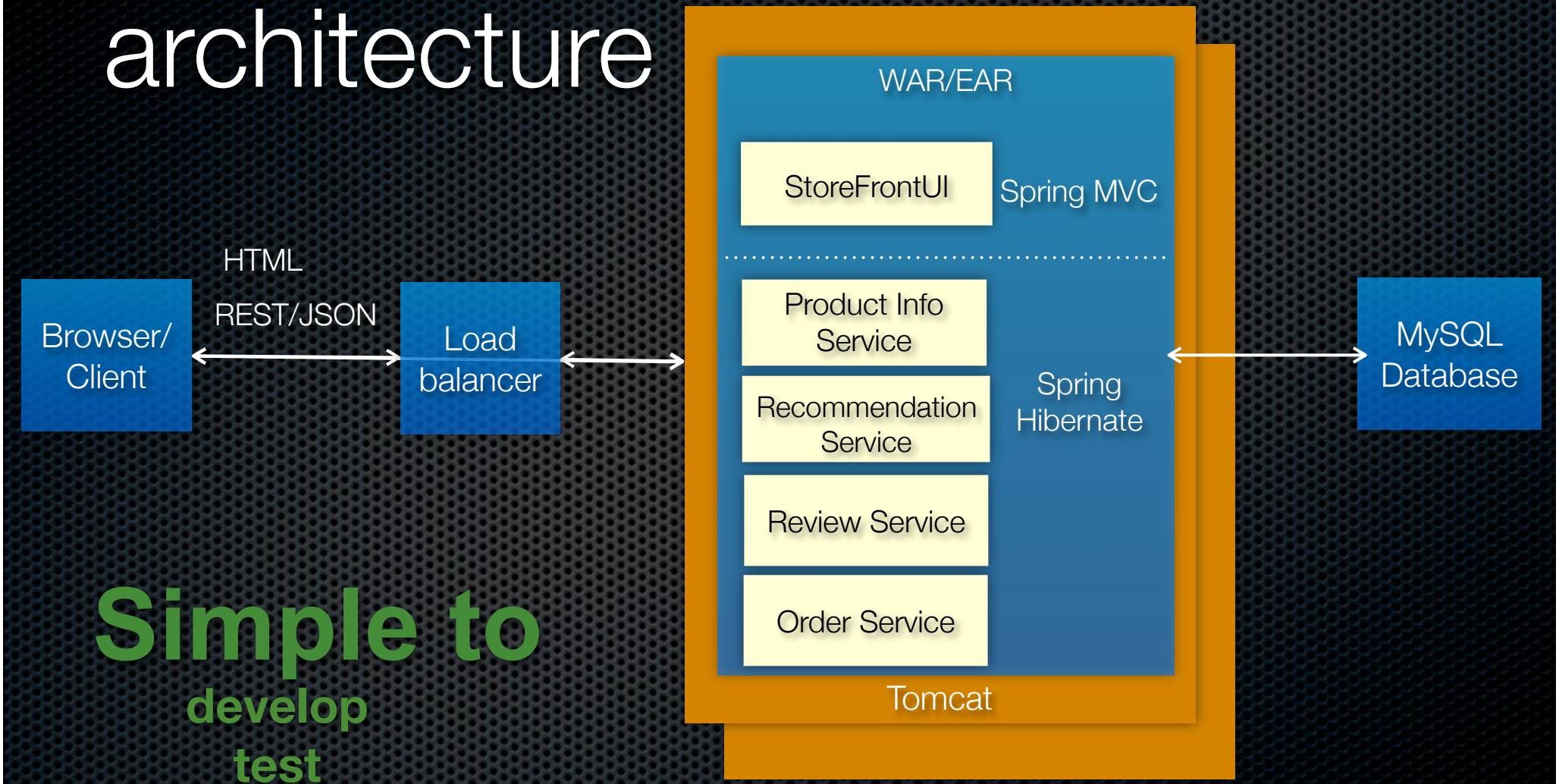
- Founder of a buzzword compliant (stealthy, social, mobile, big data, machine learning, ...) startup
- Consultant helping organizations improve how they architect and deploy applications using cloud computing, micro services, polyglot applications, NoSQL, ...

Agenda

- The (sometimes evil) monolith
- Decomposing applications into services
- Client ⇔ service interaction design
- Decentralized data management

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

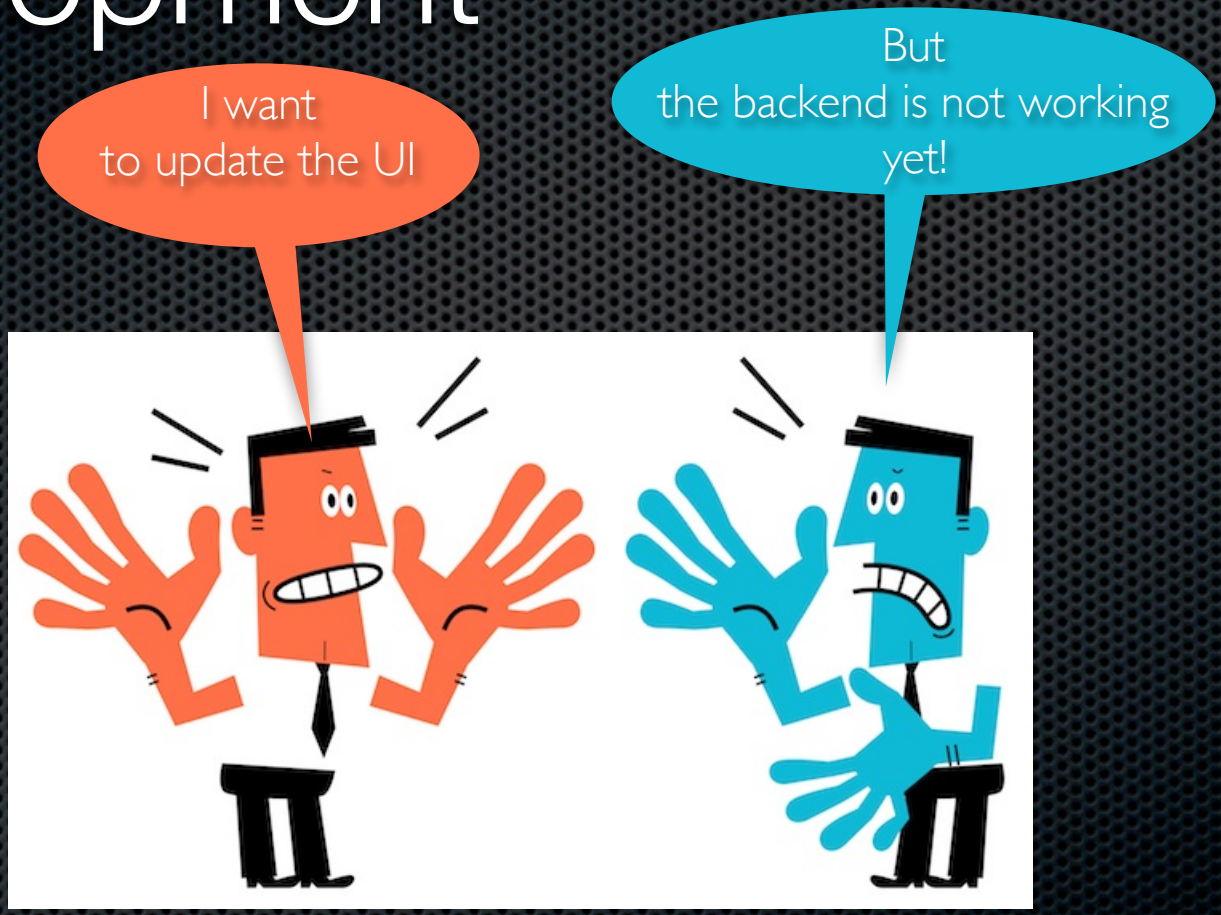
Eggs in one basket

- 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

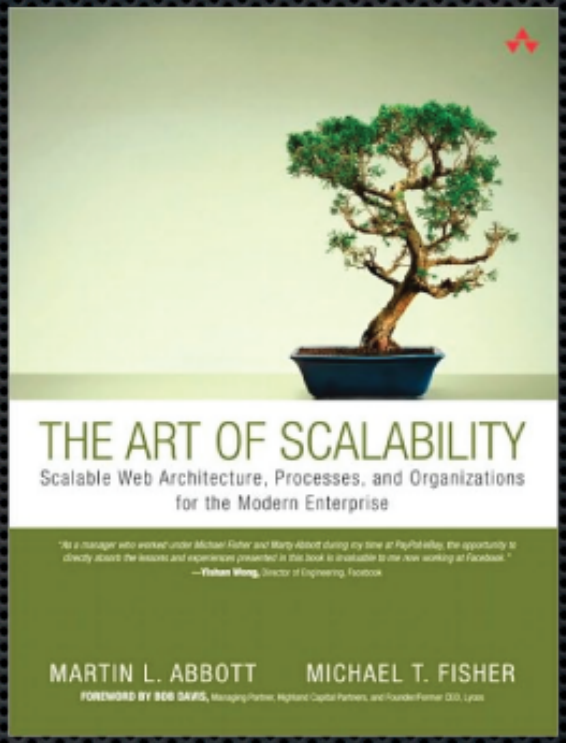


Lots of coordination and communication required

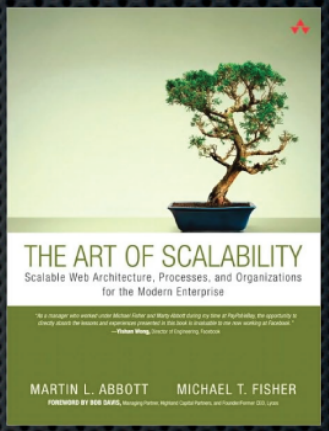
Requires long-term commitment to a technology stack

Agenda

- The (sometimes evil) monolith
- Decomposing applications into services
- Client ⇔ service interaction design
- Decentralized data management

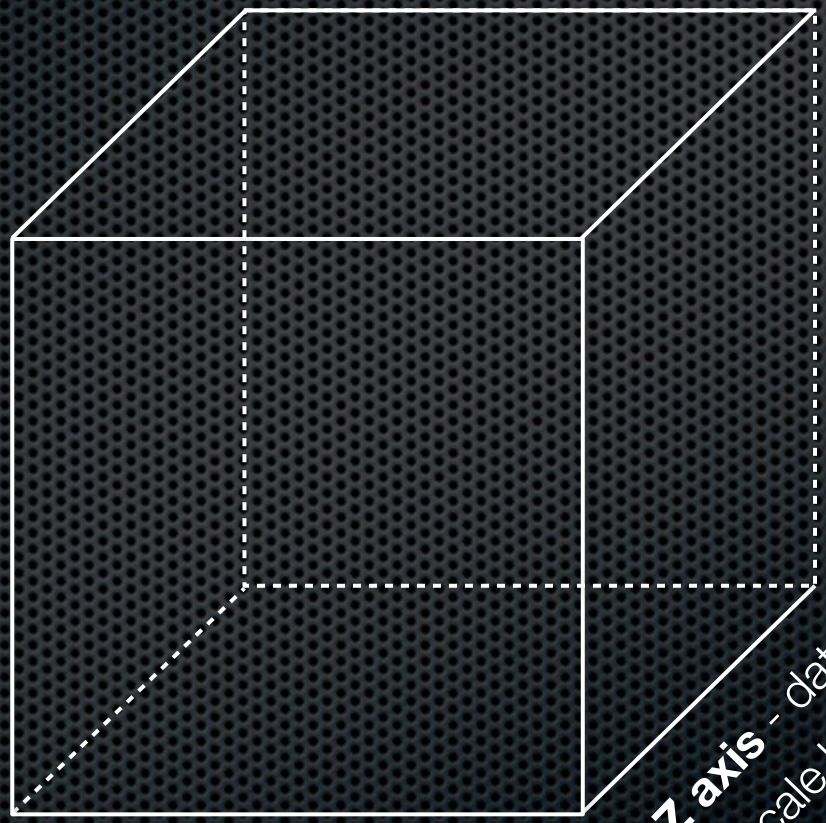

THE ART OF SCALABILITY
Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise
MARTIN L. ABBOTT
MICHAEL T. FISHER

The scale cube



Y axis - functional decomposition

Scale by splitting different things



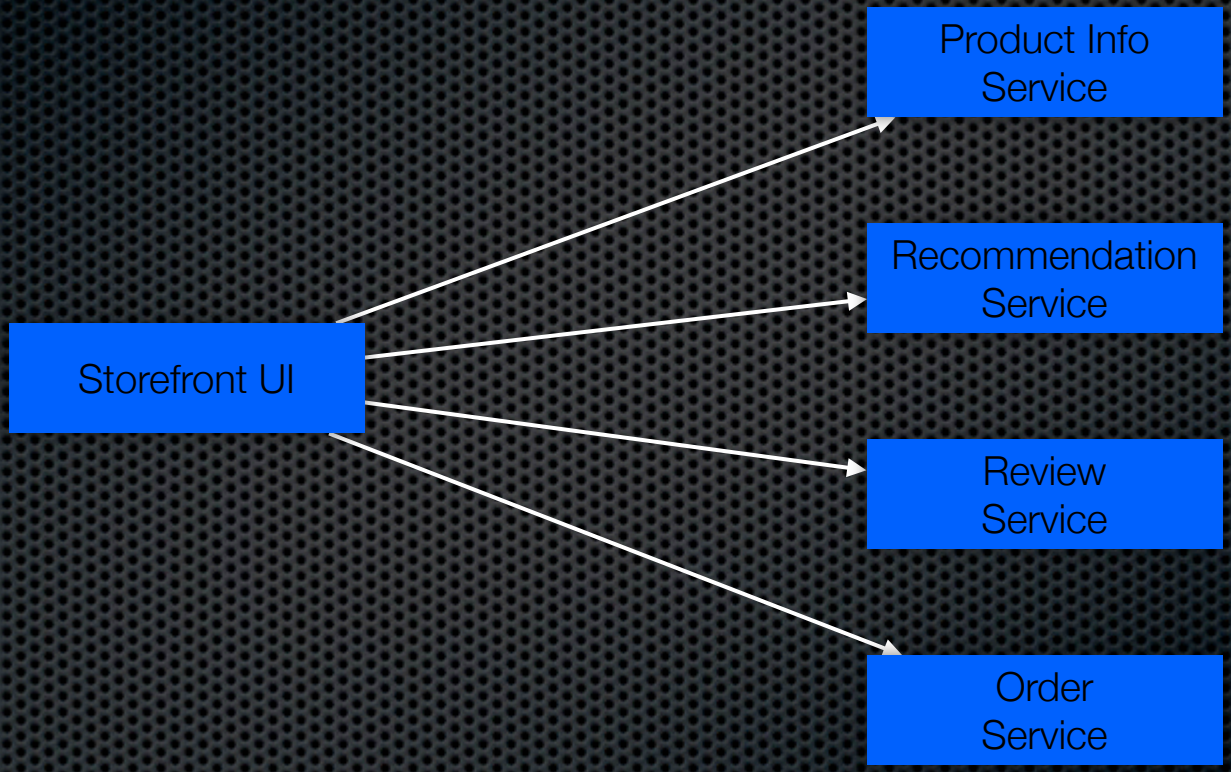
Z axis - data partitioning
Scale by splitting similar things

X axis
- horizontal duplication

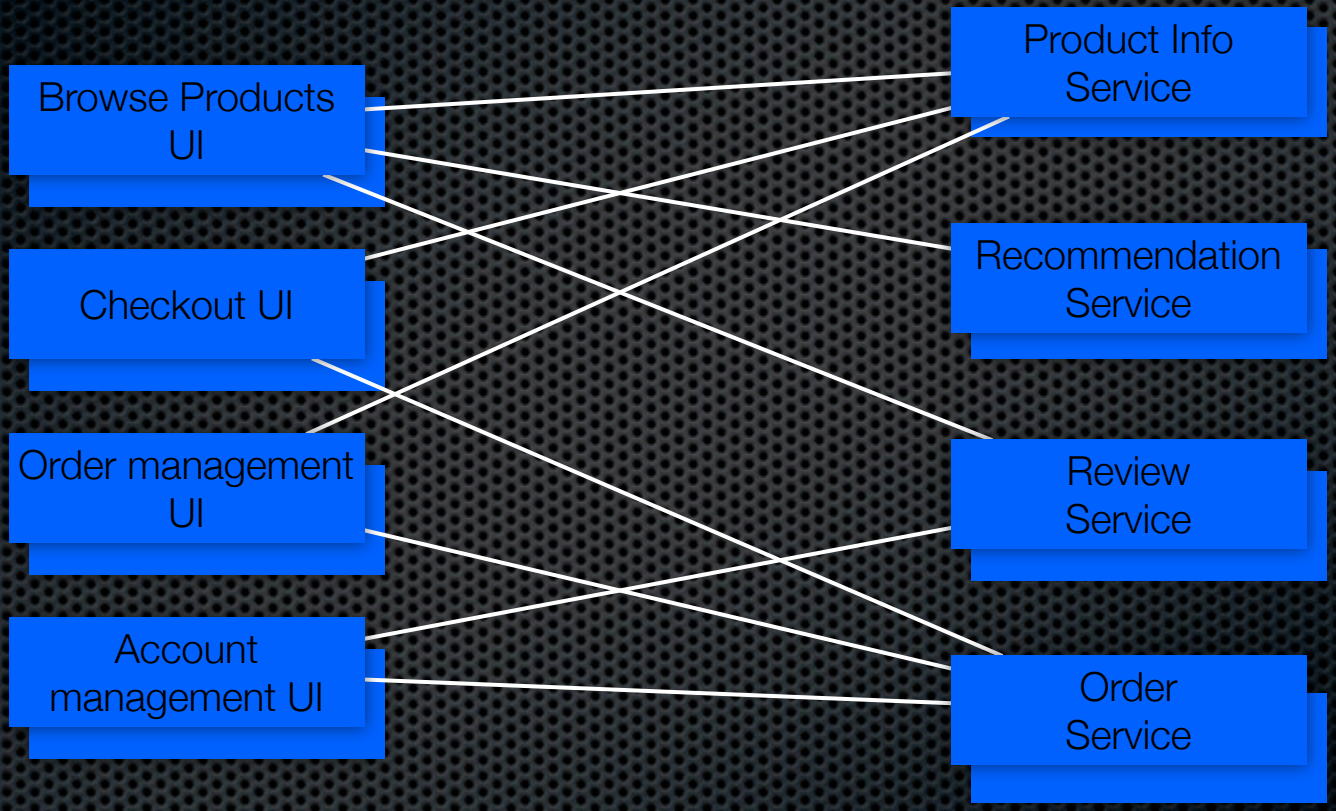
@crichardson

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

Example micro-service using Spring Boot

For more on micro-services see http://microservices.io

@crichardson

But more realistically...

Focus on building cohesive 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 drawbacks

Complexity of developing and managing a distributed system

http://contino.co.uk/blog/2013/03/31/microservices-no-free-lunch.html

Using a PaaS can significantly simplify deployment

Multiple databases
&
Transaction management

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

But there are many benefits

Smaller, simpler apps

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

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

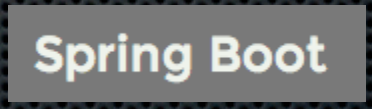
System-level

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



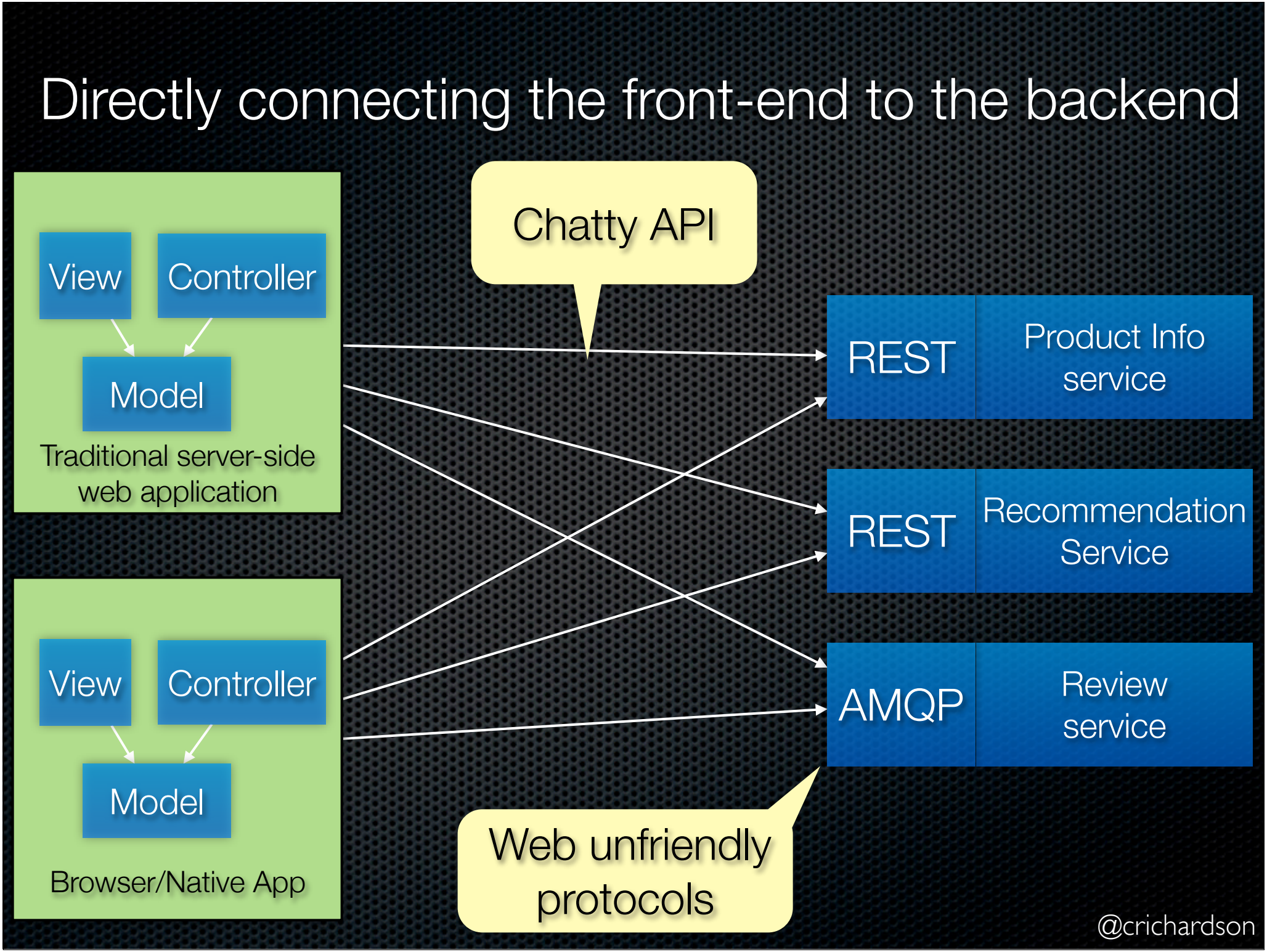
... and fail safely

Agenda

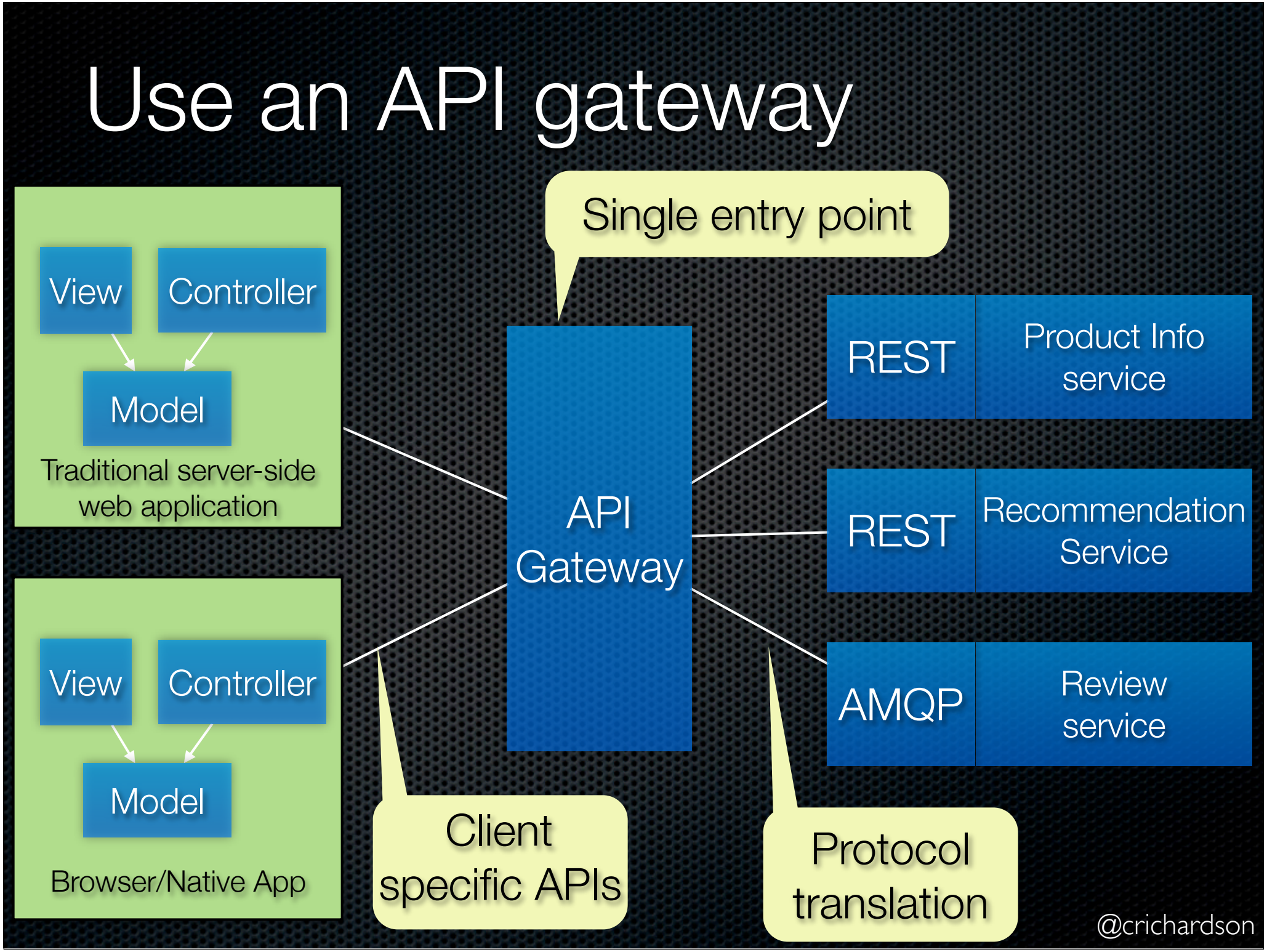
- The (sometimes evil) monolith
- Decomposing applications into services
- Client ⇔ service interaction design
- Decentralized data management

How do clients of the system interact with the services?

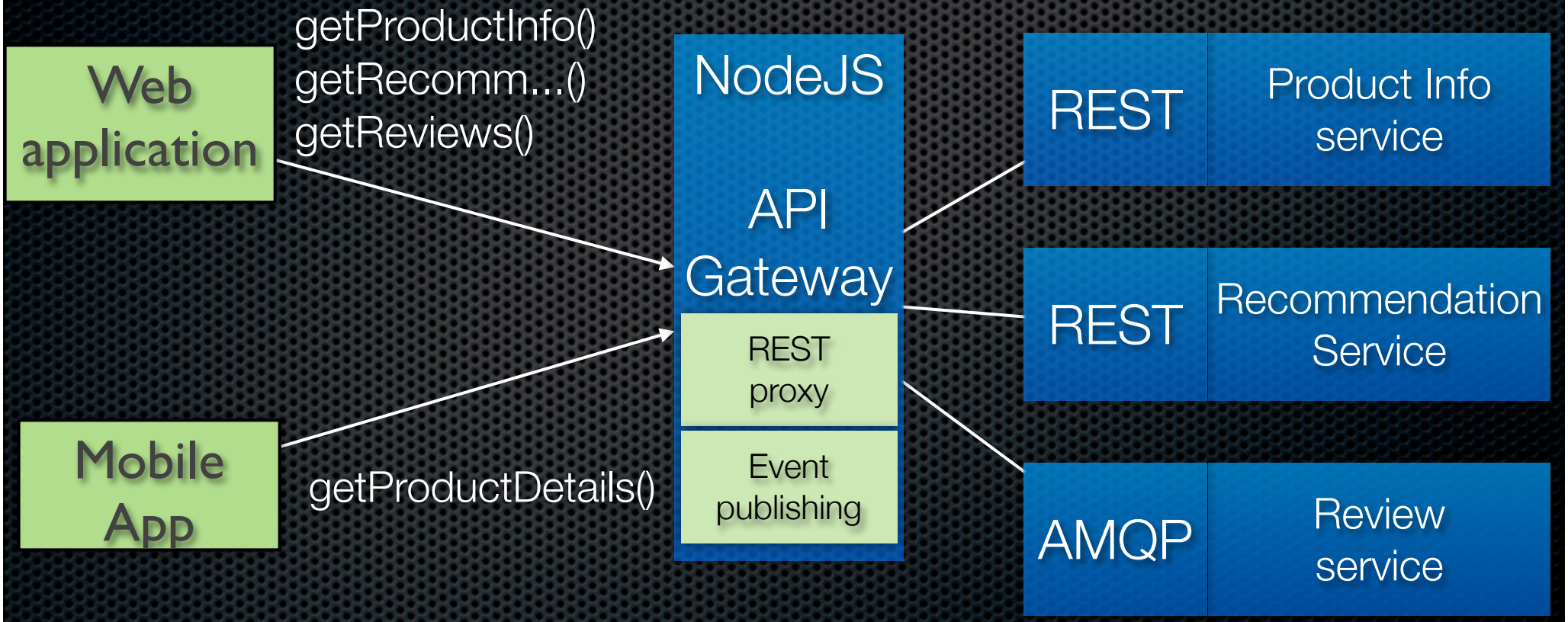
Directly connecting the front-end to the backend



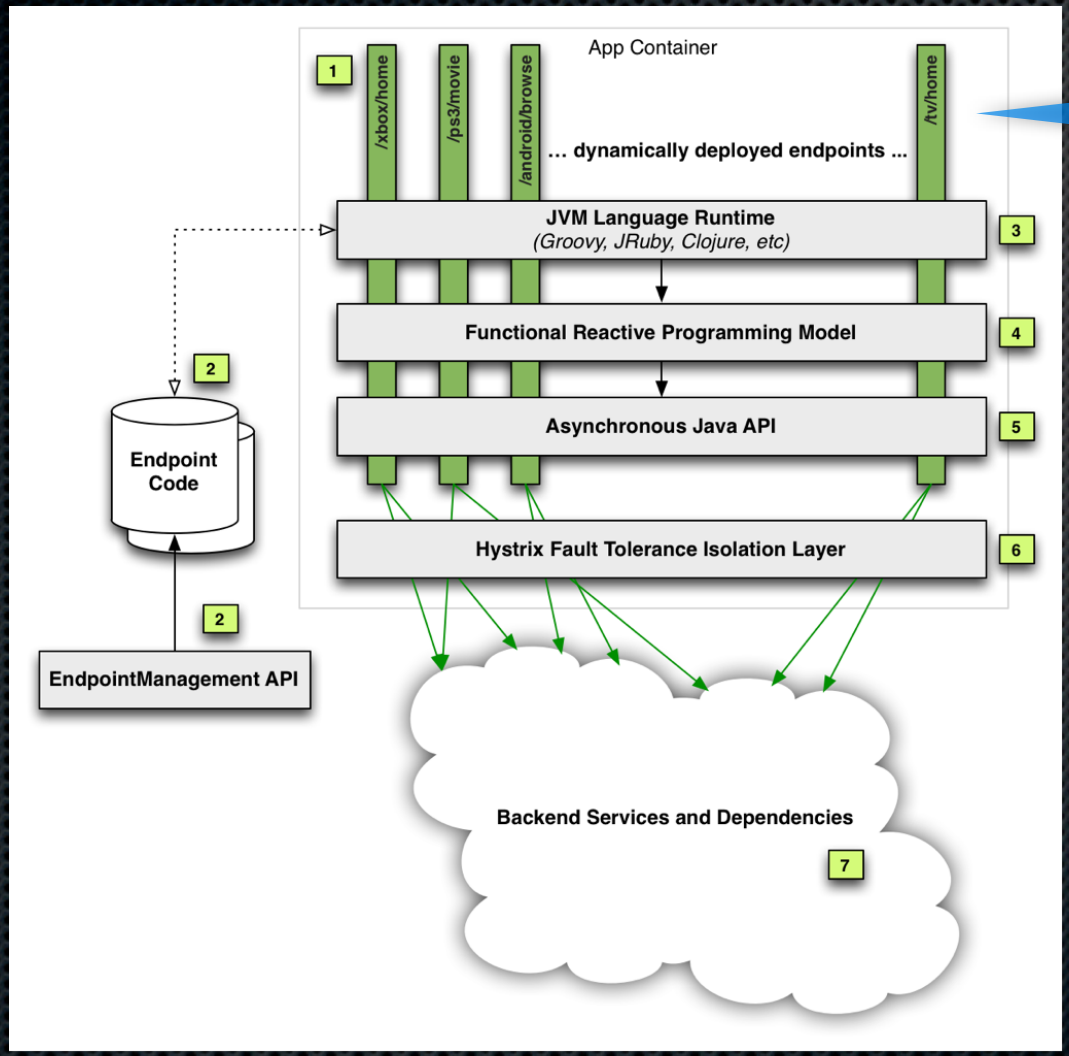
Use an API gateway



Optimized client-specific APIs



Netflix API Gateway



Device specific end points

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

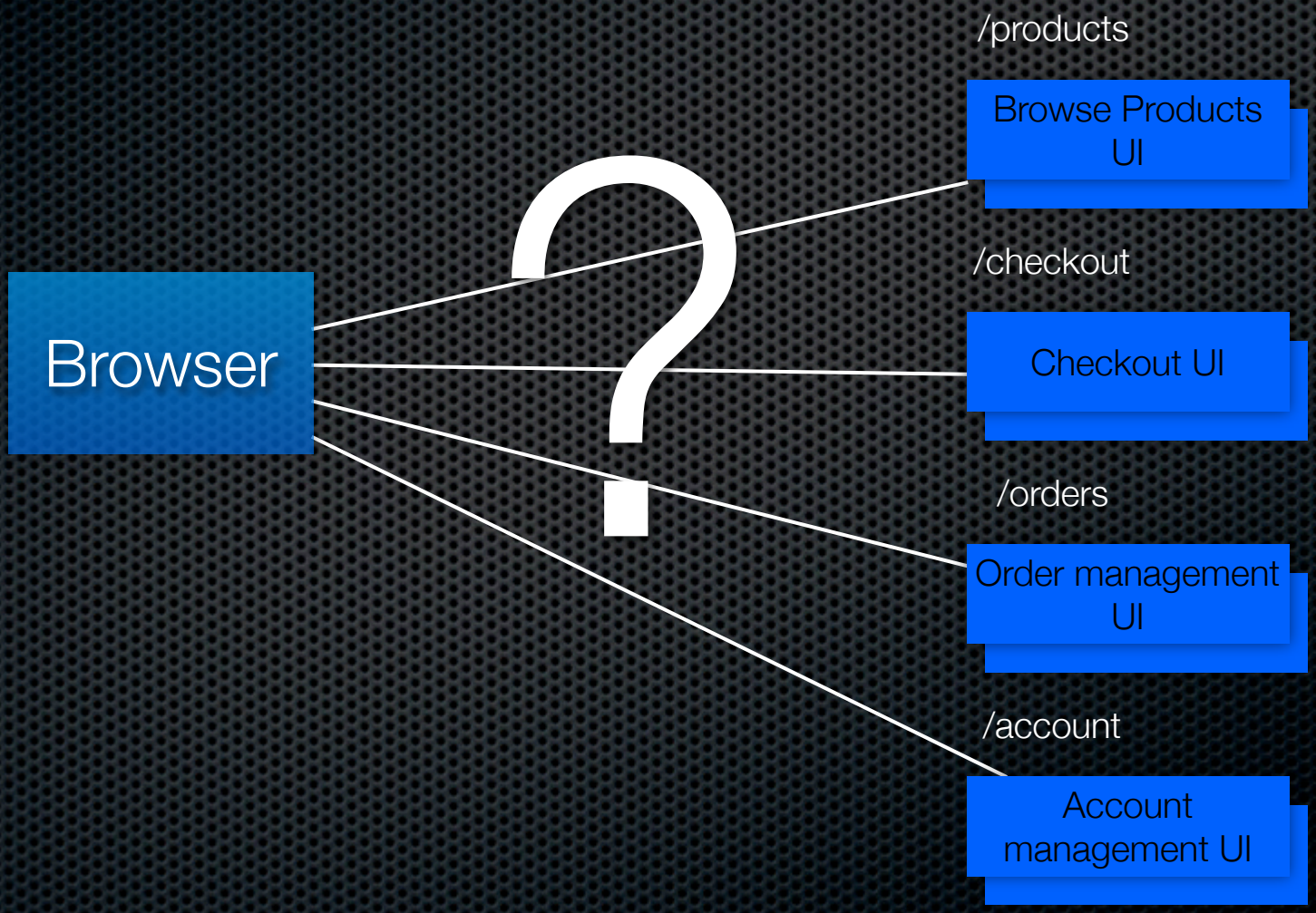
API gateway design challenges

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

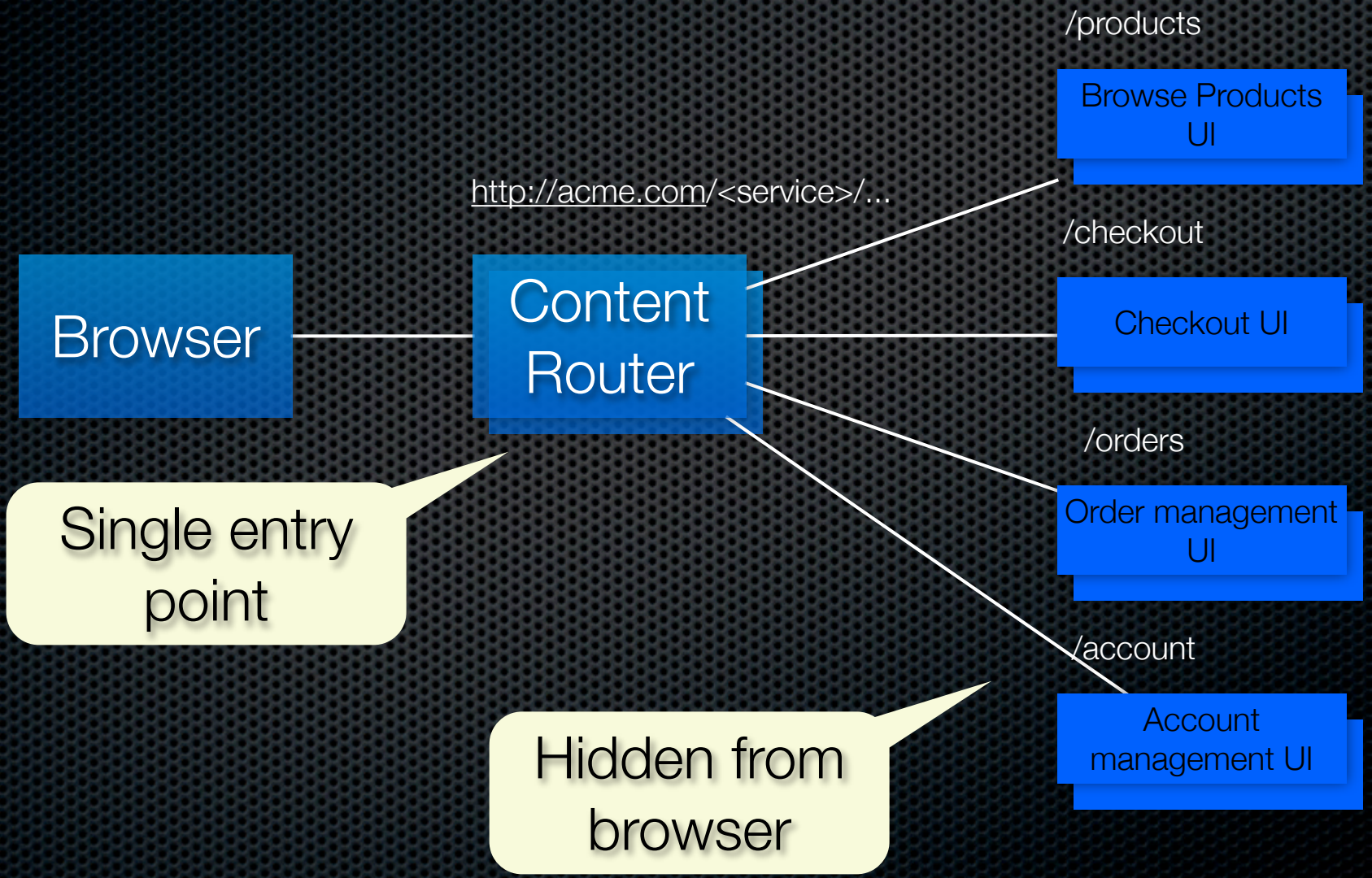
http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

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

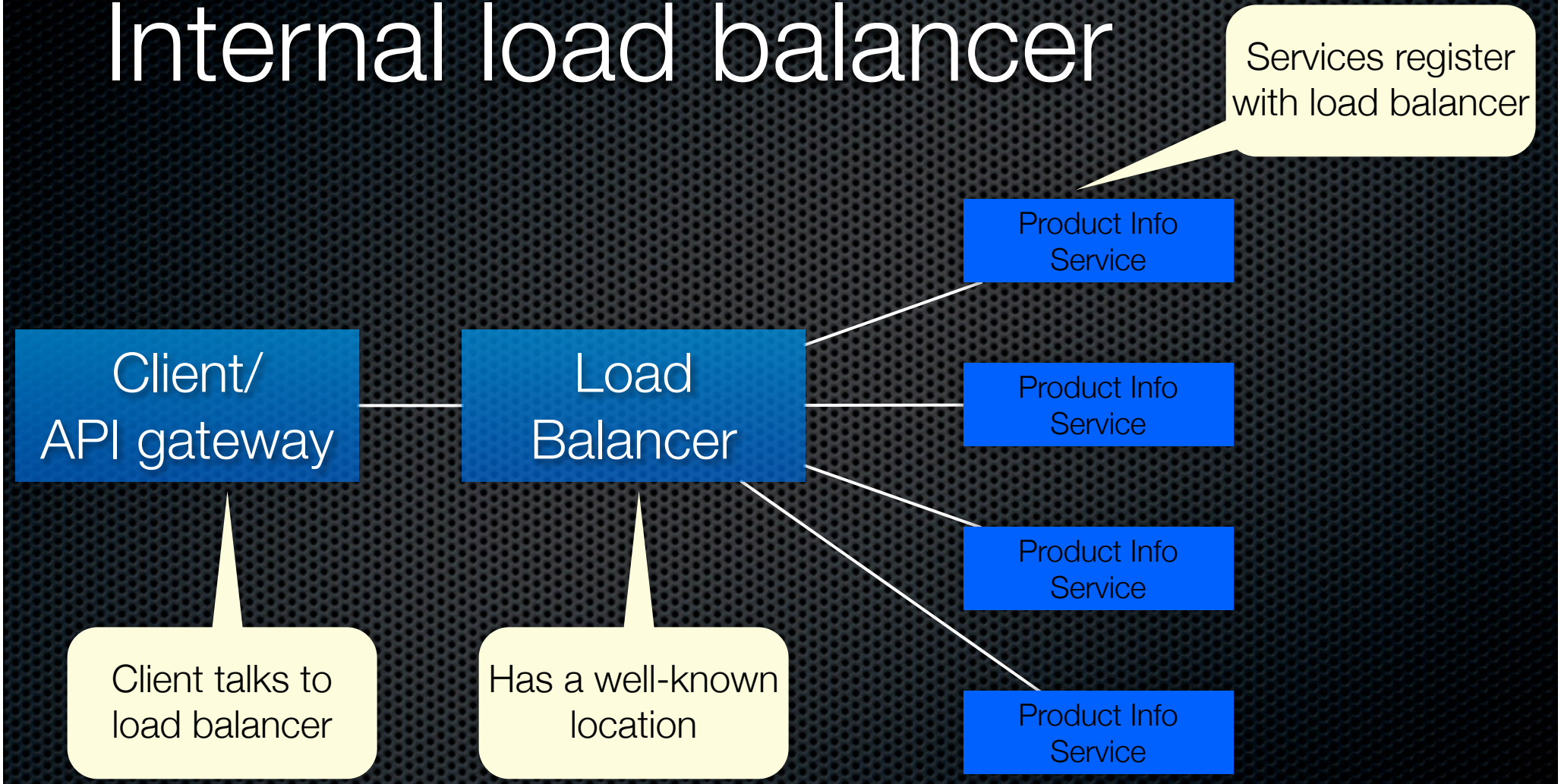
Cons

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

Pros and cons of HTTP

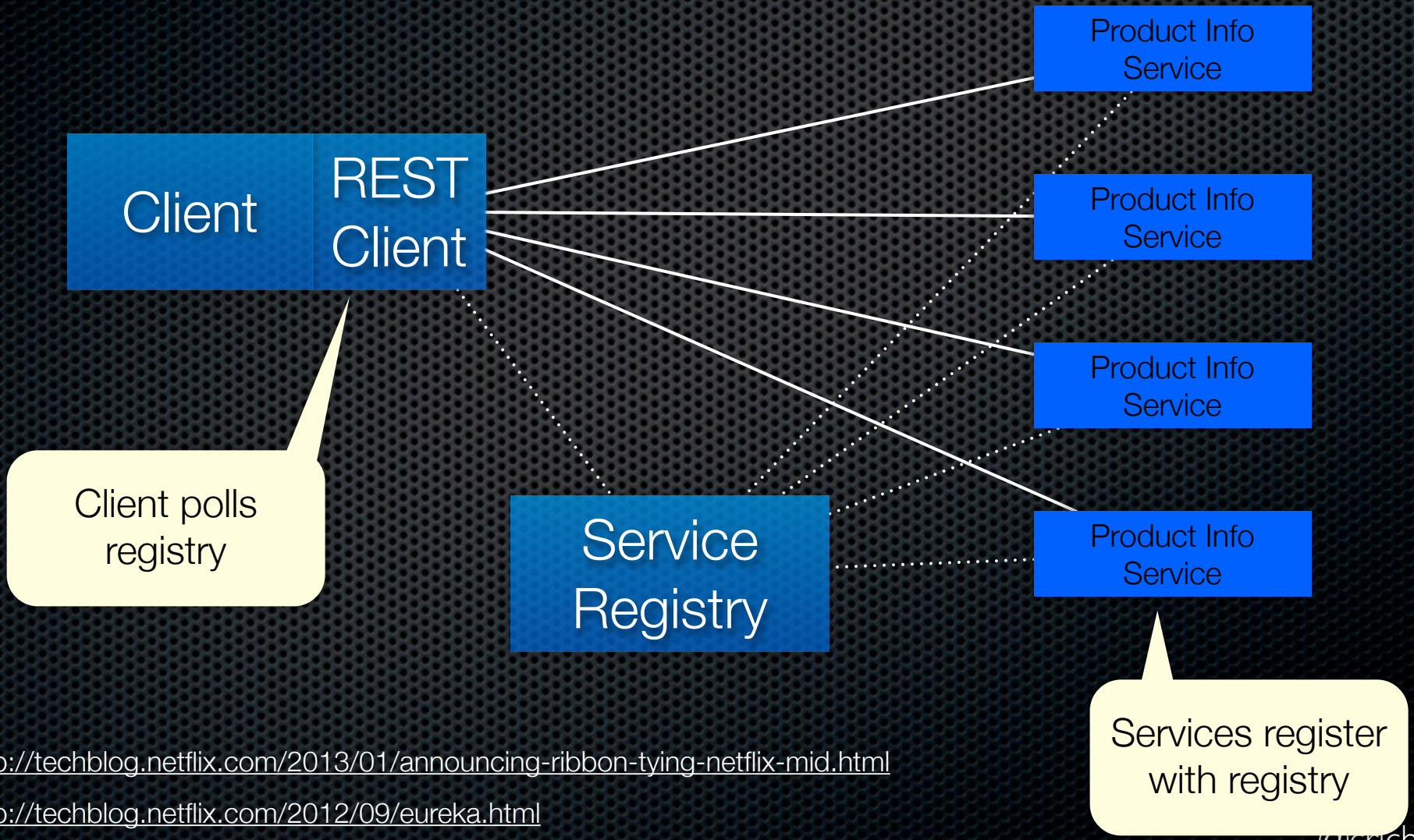
- Pros
  - Simple and familiar
  - Request/reply is easy
  - Firewall friendly
  - No intermediate broker
- Cons
  - Only supports request/reply
  - Server must be available
  - Client needs to discover URL(s) of server(s)

Discovery option #1: Internal load balancer



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

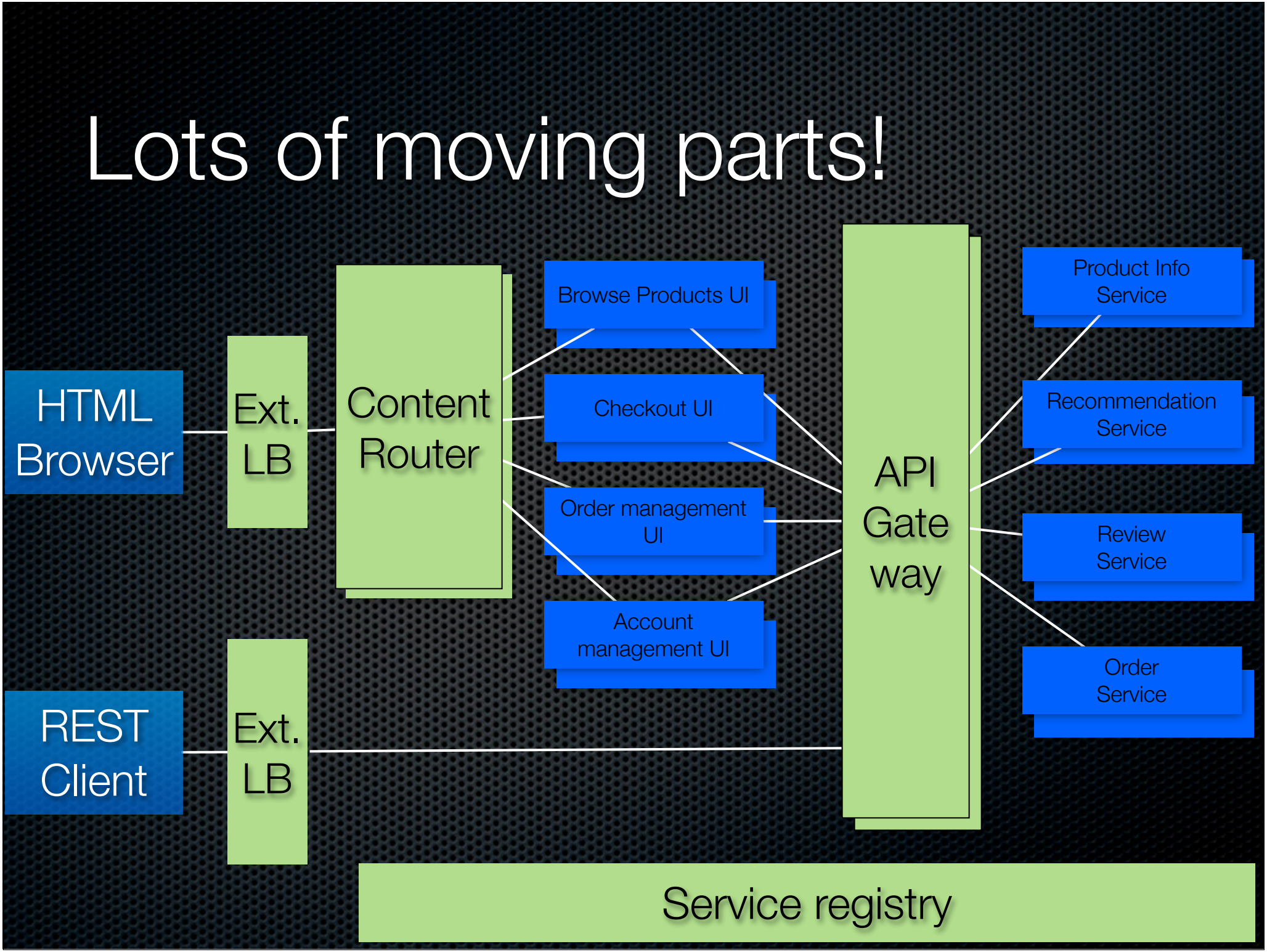
Discovery option #2: client-side load balancing



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

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

Lots of moving parts!



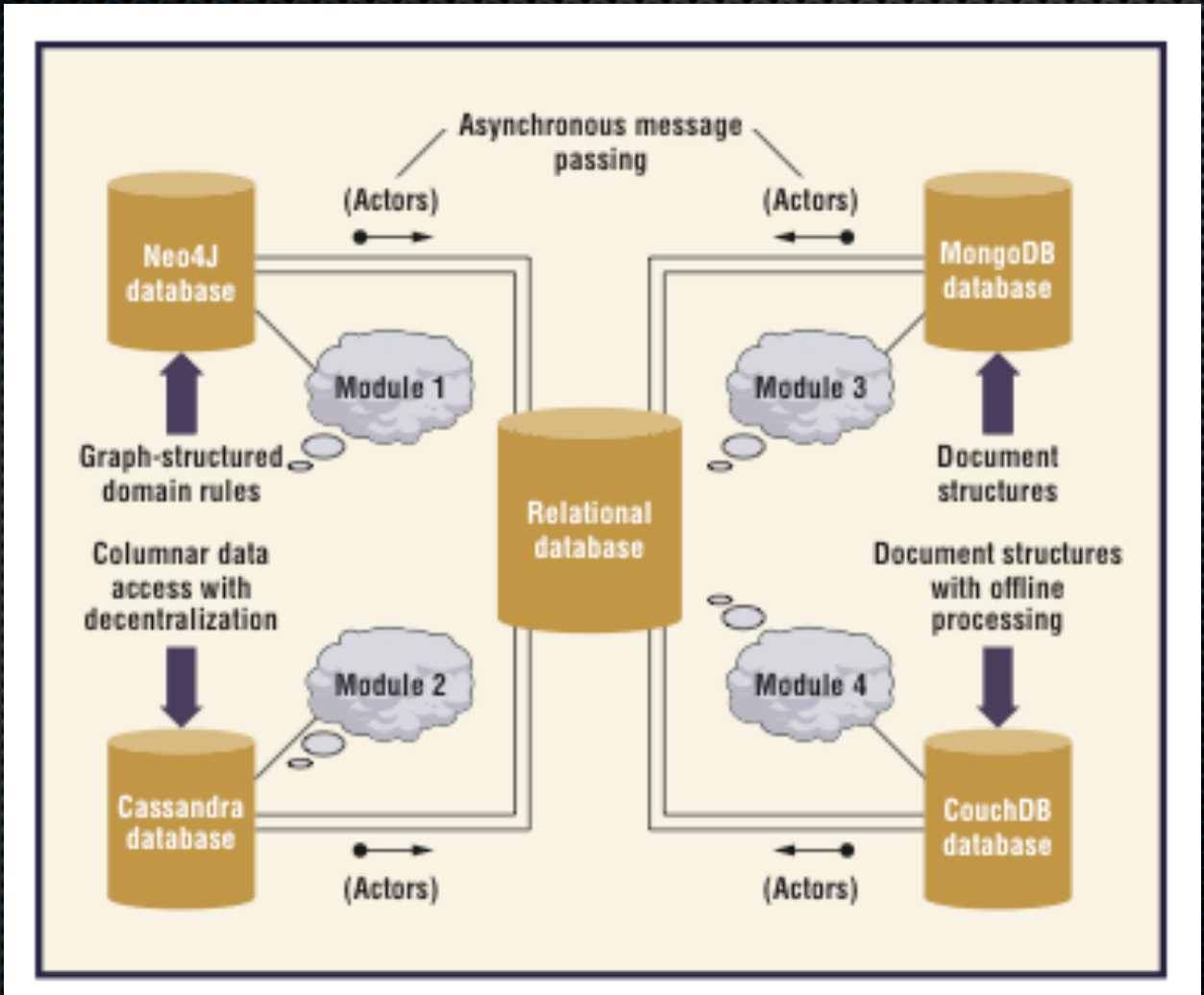
Agenda

- The (sometimes evil) monolith
- Decomposing applications into services
- Client ⇔ service interaction design
- Decentralized data management

Decomposed services ⇒ decomposed databases

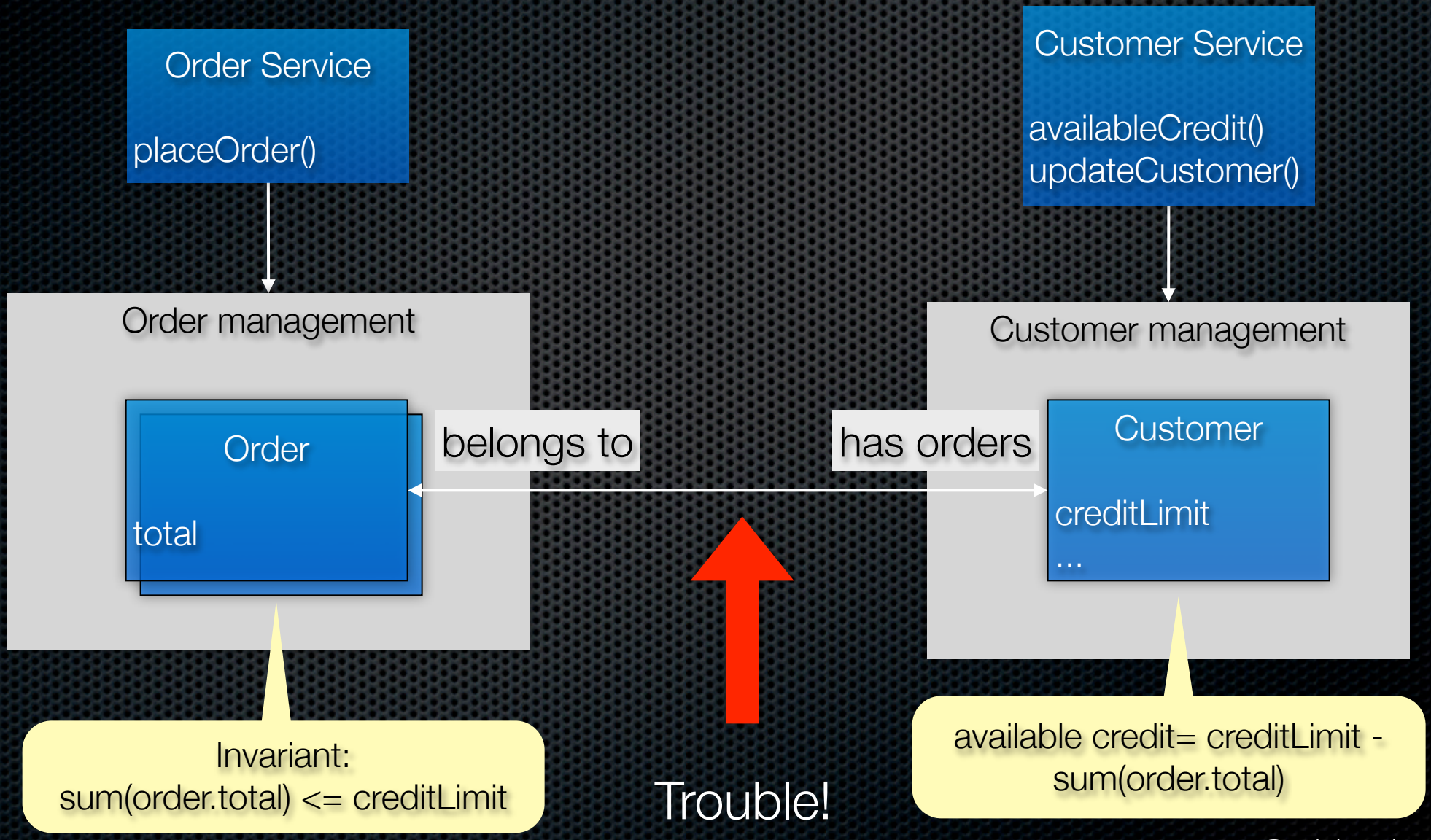
Separate databases ⇒ less coupling

Decomposed databases ⇒ polyglot persistence



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

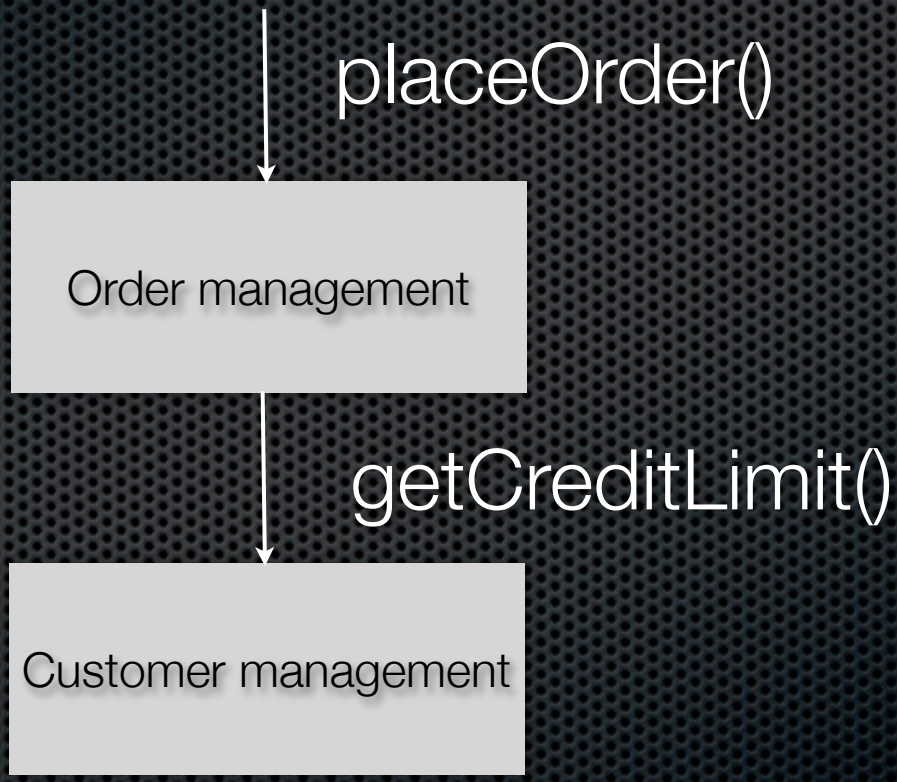
Untangling orders and customers



Problems

- Reads
  - Service A needs to read data owned by service B
- Updates
  - Transaction must update data owned by multiple services

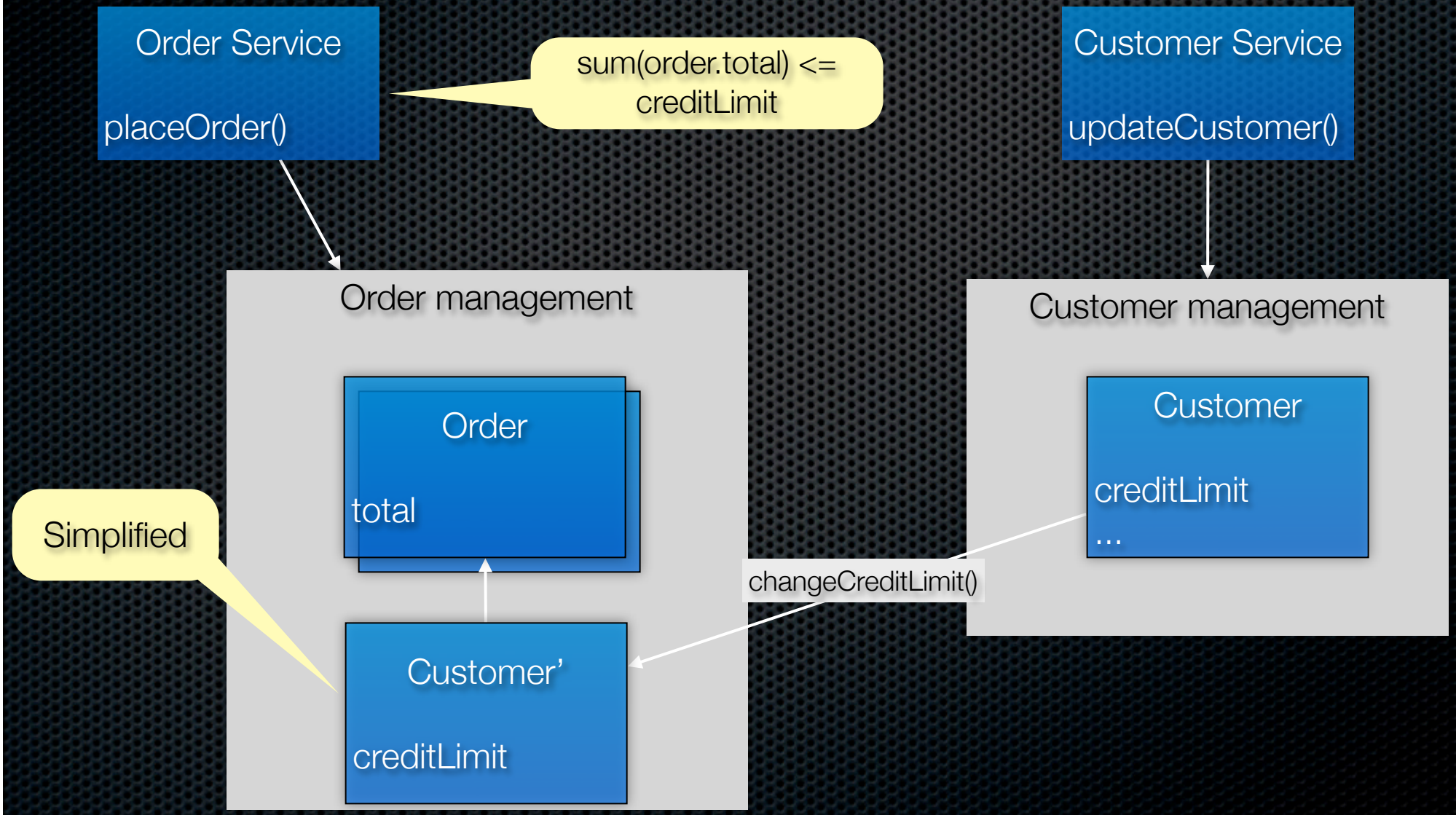
Handling reads: requesting credit limit



Pulling data

- Benefits
  - Simple to implement
  - Ensures data is fresh
- Drawbacks
  - Reduces availability
  - Increases response time

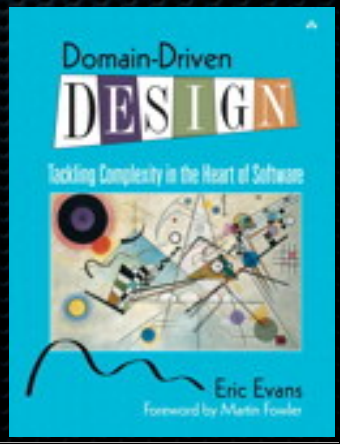
Handling reads: replicating the credit limit



Useful idea: Bounded context

- Different services have a different view of a domain object, e.g.
  - User Management = complex view of user
  - Rest of application: User = PK + ACL + Name

- Different services can have a different domain model



Replicating data

- Benefits
  - Improved availability for reads
  - Improves latency
- Drawbacks
  - Additional complexity of replication mechanism

How to handle updates (including of replicated data)?

Use distributed transactions

- Benefits
  - Guarantees consistency
- Drawbacks
  - Complex
  - Reduced availability

Use eventual consistency

- How
  - Services publish events when data changes
  - Subscribing services update their data
- Benefits:
  - Simpler
  - Better availability
- Drawbacks:
  - Application has to handle inconsistencies

How do services publish events?

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

Change tracking options

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

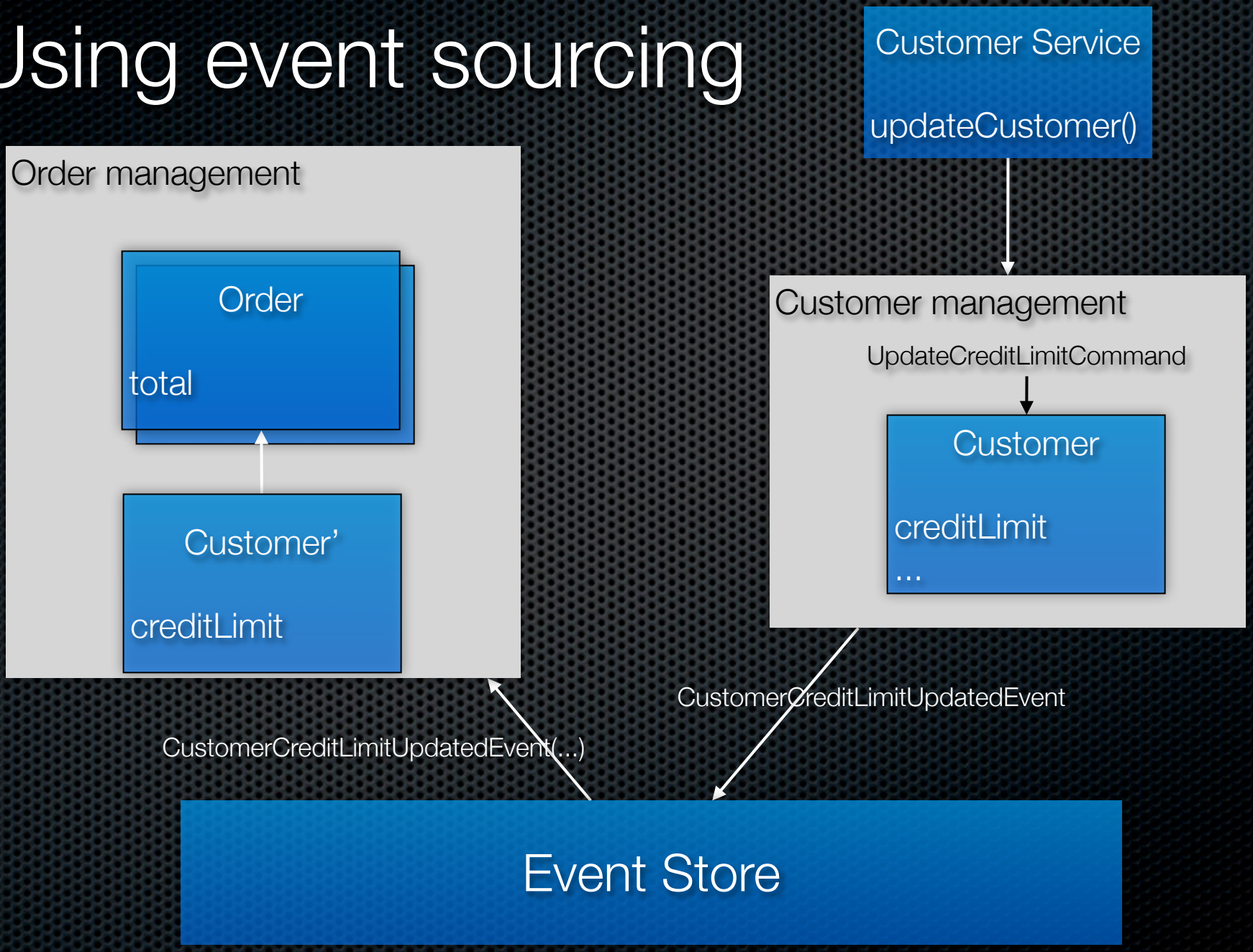
Event sourcing

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

EventStore API

```
trait EventStore {

  def save[T](entityId: Id, events: Seq[Event]): T

  def update[T](entityId: Id,
                version: EntityVersion, events: Seq[Event]): T

  def load[T](entityType: Class[T], entityId: EntityId): T

  def subscribe(...) : ...
   ..
}
```

Using event sourcing



Customer aggregate

```
case class Customer(customerId: String, creditLimit: BigDecimal)
  extends ValidatingAggregate[Customer, CustomerCommands.CustomerCommand] {


  def this() = this(null, null)

  override def validate = {
    case CreateCustomerCommand(customerId, creditLimit) =>
          Seq(CustomerCreatedEvent(customerId, creditLimit))

    case UpdateCreditLimitCommand(newLimit) if newLimit >= 0 =>
        Seq(CustomerCreditLimitUpdatedEvent(newLimit))
  }

  override def apply = {

    case CustomerCreatedEvent(customerId, creditLimit) =>
        copy(customerId=customerId, creditLimit=creditLimit)

    case CustomerCreditLimitUpdatedEvent(newLimit) =>
        copy(creditLimit=newLimit)

  }
}
```

Command ⇒ Events

Event ⇒ Updated state

Unfamiliar but it solves many problems

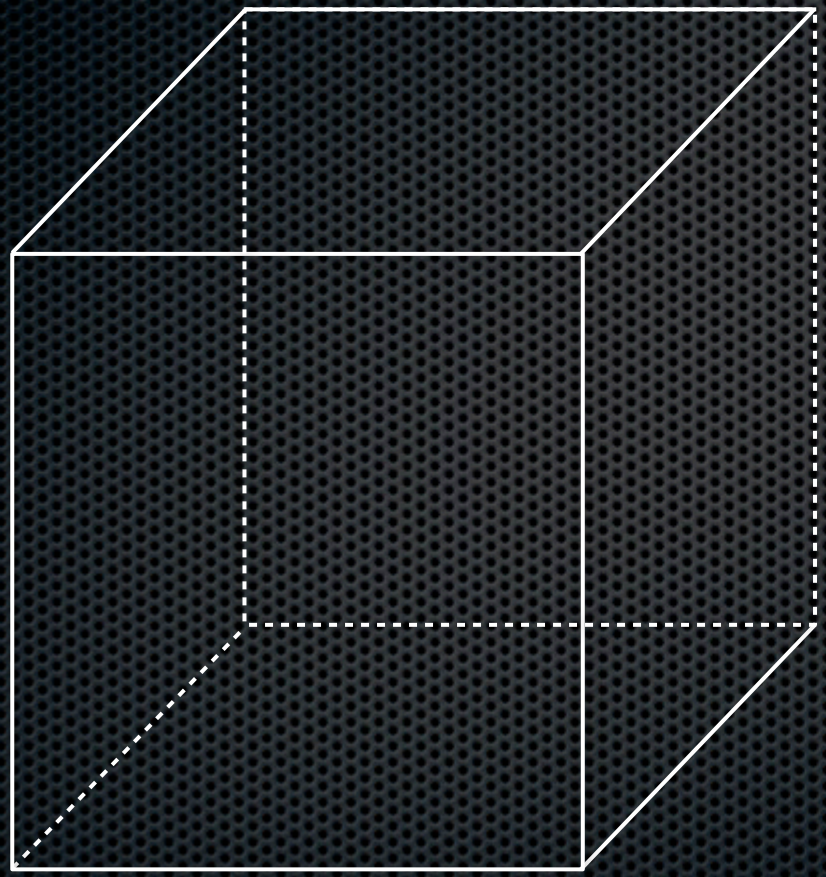
- Eliminates O/R mapping problem
- Supports both SQL and NoSQL databases
- Publishes events reliably
- Reliable eventual consistency framework
- ...

Summary

Monolithic applications are simple to develop and deploy

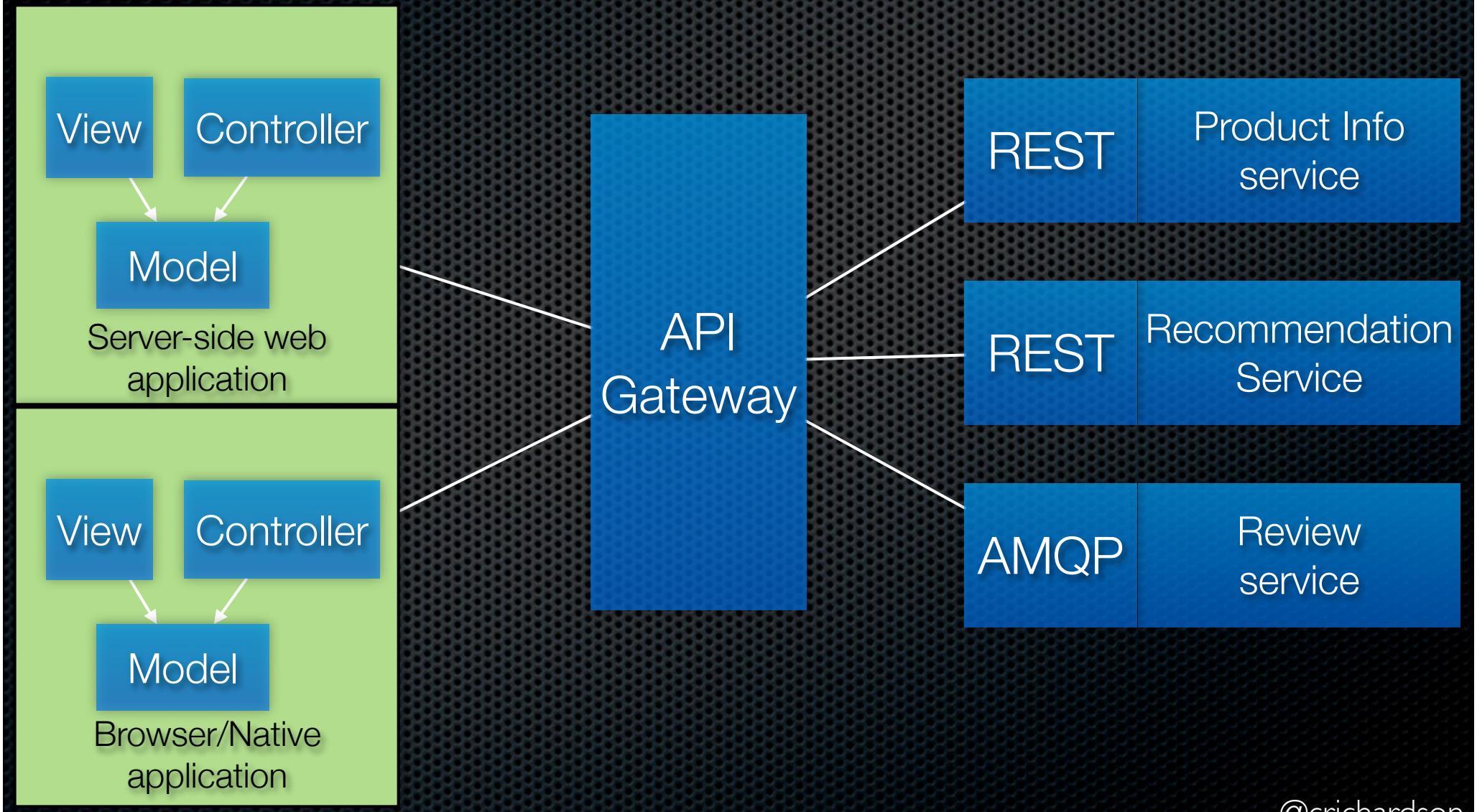
BUT have significant drawbacks

Apply the scale cube

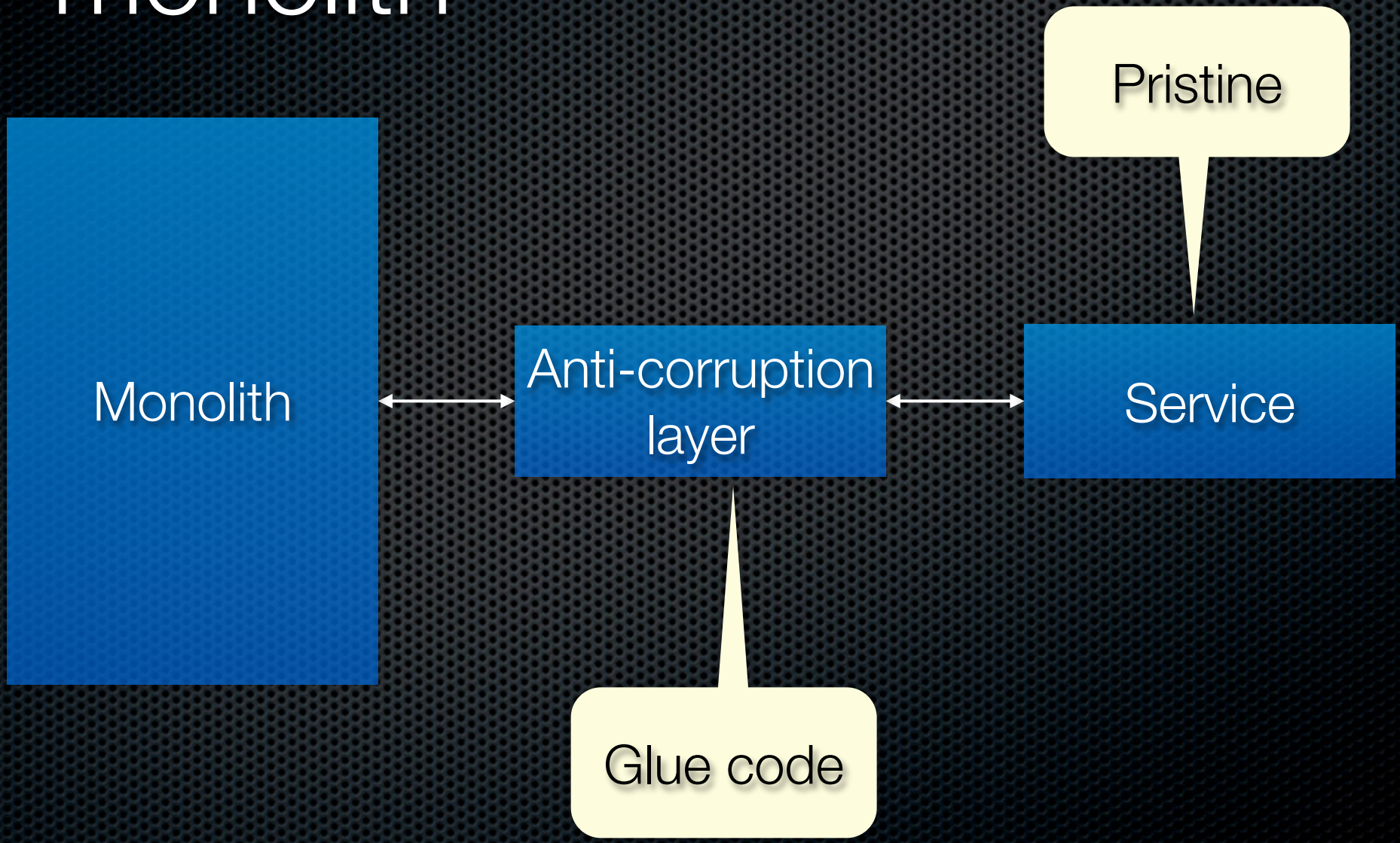


- Modular, polyglot, and scalable applications
- Services developed, deployed and scaled independently

Use a modular, polyglot architecture



Start refactoring your monolith




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