Developing applications with a microservice architecture

Chris Richardson

Author of POJOs in Action

Founder of the original CloudFoundry.com

@crichardson

chris@chrisrichardson.net

http://plainoldobjects.com

Presentation goal

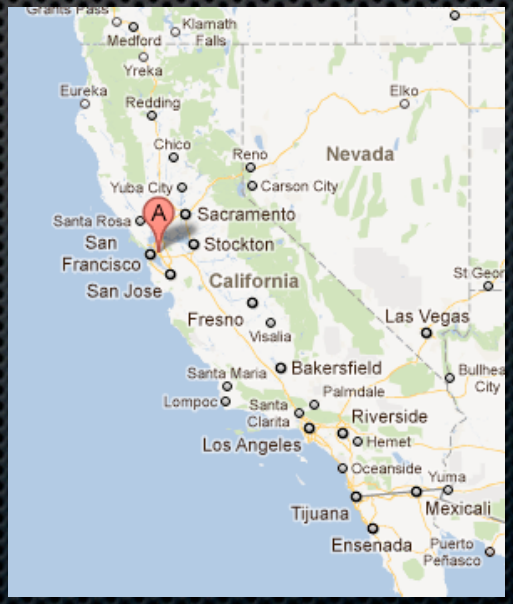
How decomposing applications into microservices

improves deployability and scalability

and

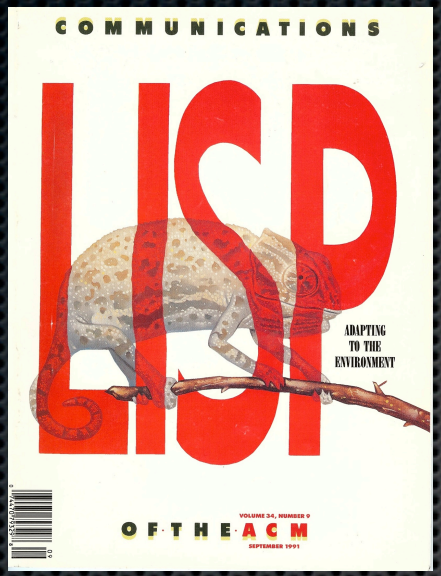
simplifies the adoption of new technologies

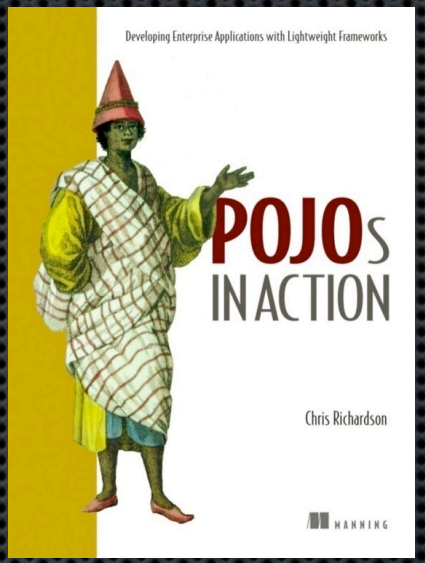
About Chris

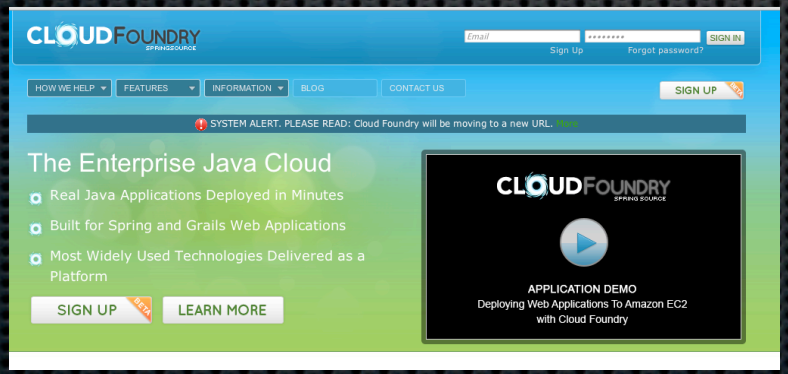


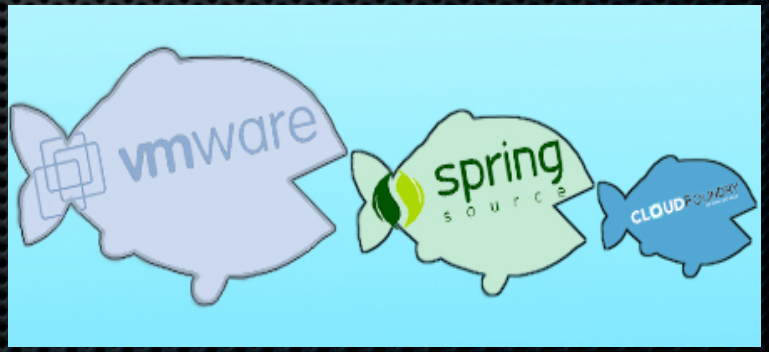
@crichardson

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

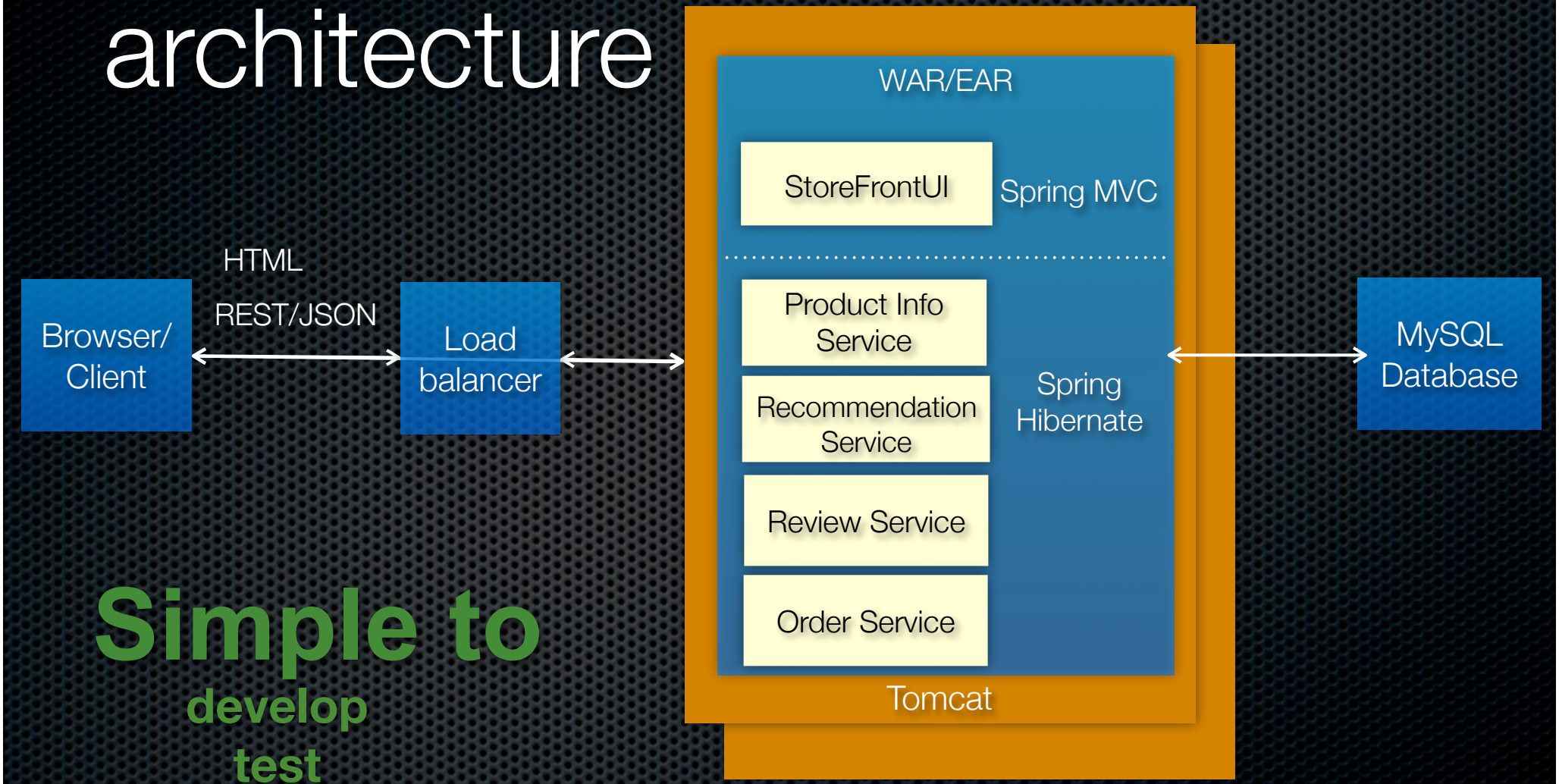
- Founder of a buzzword compliant (stealthy, social, mobile, big data, machine learning, ...) startup
- Consultant helping organizations improve how they architect and deploy applications using cloud computing, micro services, polyglot applications, NoSQL, ...

Agenda

- The (sometimes evil) monolith
- Decomposing applications into services
- Using an API gateway
- Inter-service communication mechanisms

Let’s imagine you are building an online store

Traditional application architecture



Simple to

develop

test

deploy

scale

But big, complex, monolithic applications

⇒

big 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

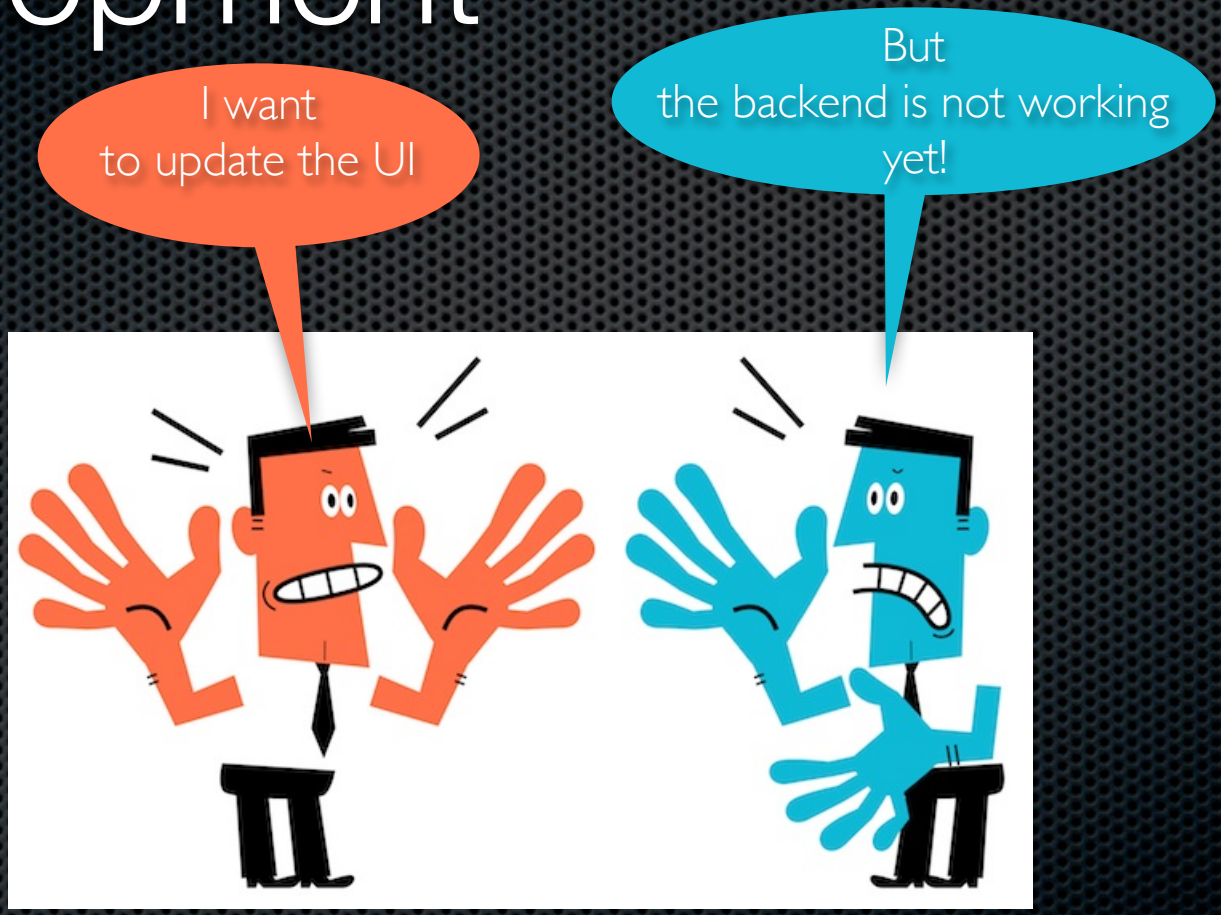
Eggs in one basket

- 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

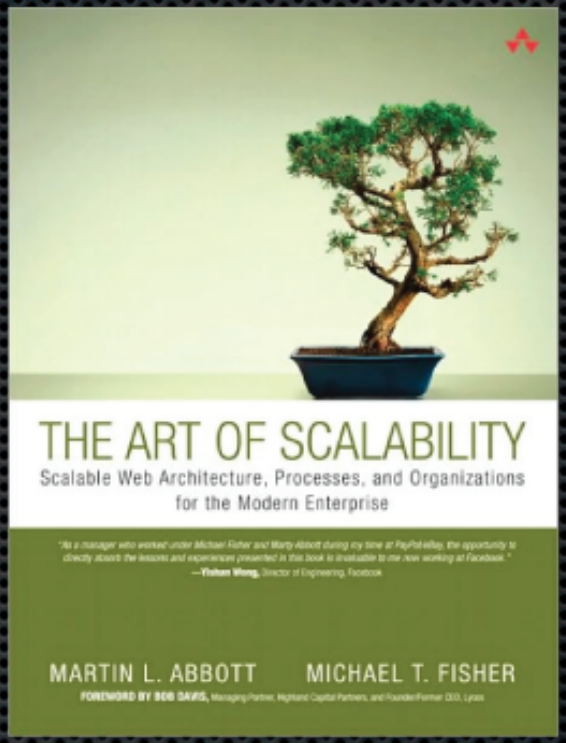


Lots of coordination and communication required

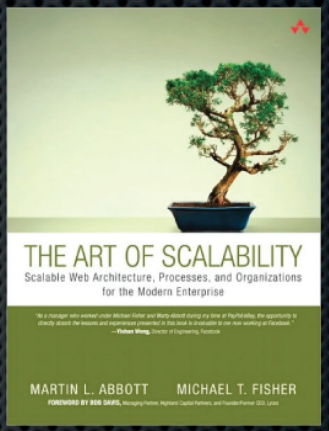
Requires long-term commitment to a technology stack

Agenda

- The (sometimes evil) monolith
- Decomposing applications into services
- Using an API gateway
- Inter-service communication mechanisms

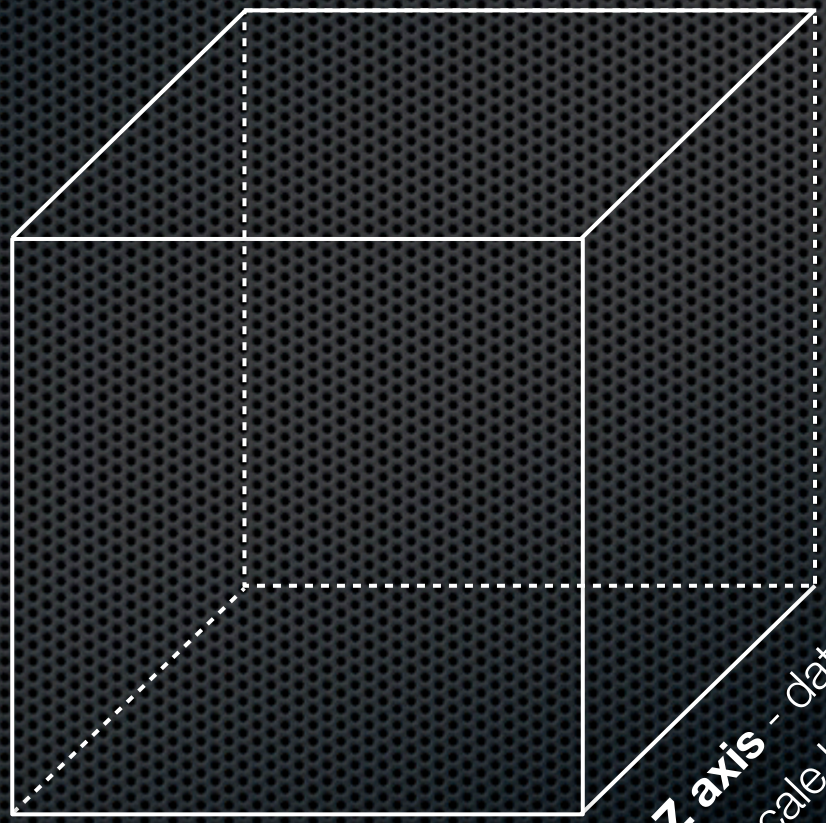

THE ART OF SCALABILITY
Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise
MARTIN L. ABBOTT
MICHAEL T. FISHER

The scale cube



Y axis - functional decomposition

Scale by splitting different things



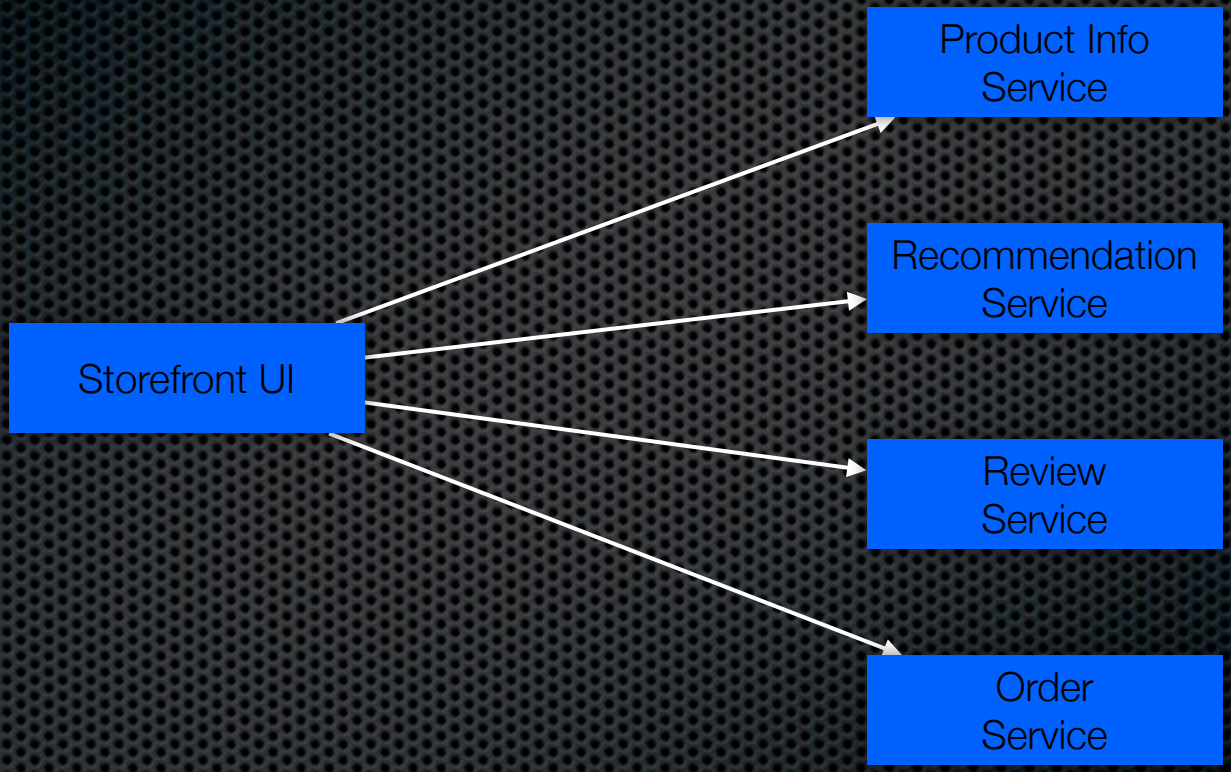
Z axis - data partitioning
Scale by splitting similar things

X axis
- horizontal duplication

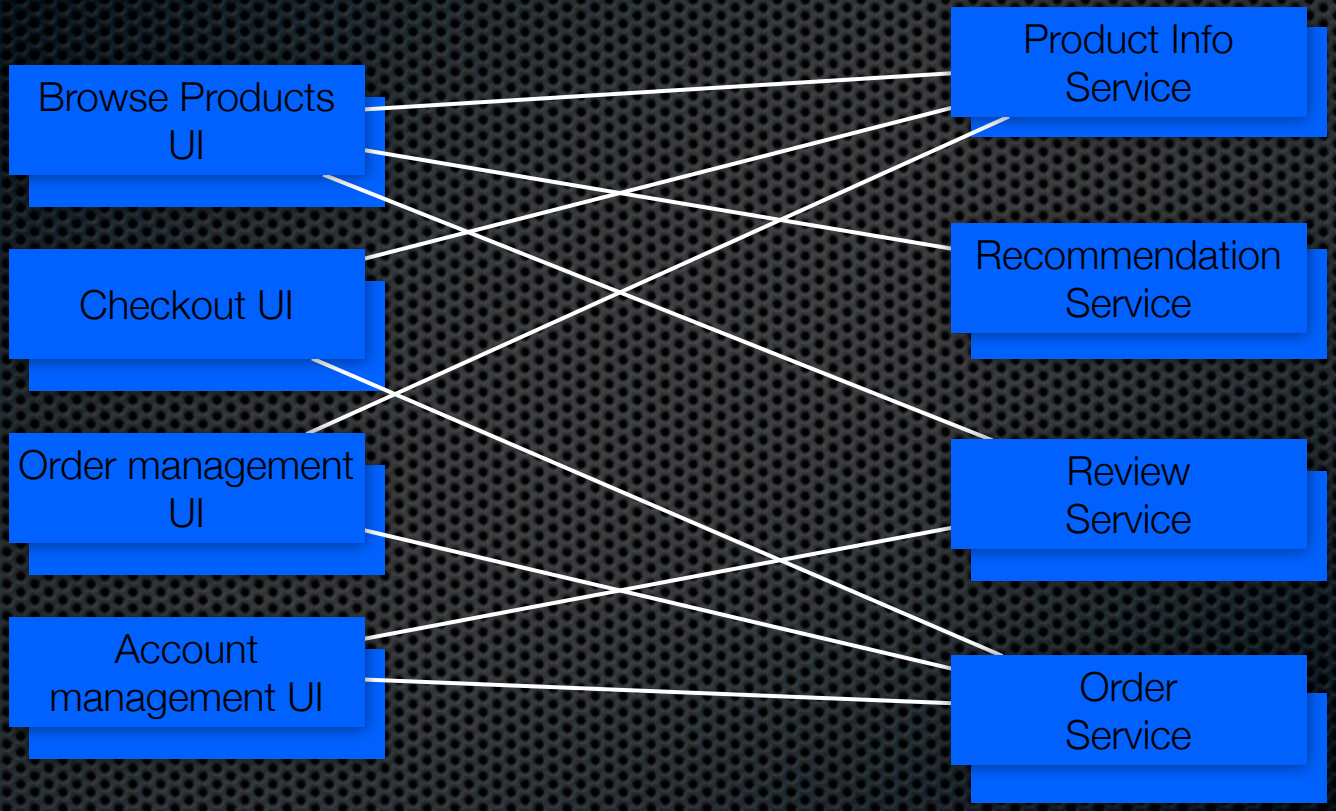
@crichardson

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

```
require 'sinatra'

post '/' do
  phone_number = params[:From]
  registration_url = "#{ENV['REGISTRATION_URL']}?phoneNumber=#{URI.encode(phone_number, "+")}"
  <<-eof
      <Response>
          <Sms>To complete registration please go to #{registration_url}</Sms>
      </Response>
eof
end
```

Responds to incoming SMS messages via Twilio

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

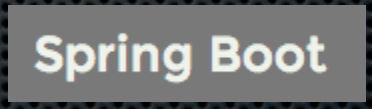
System-level

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
Come to my 11.30 am talk

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:

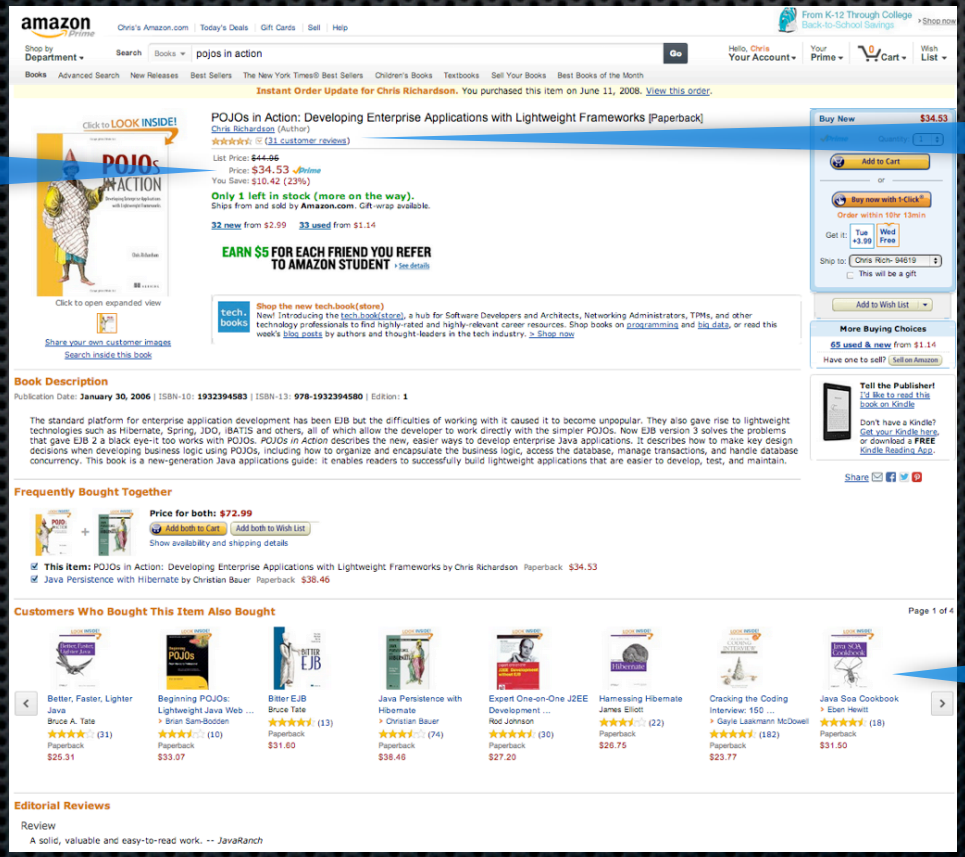
- You need it
- Refactoring is painful

Agenda

- The (sometimes evil) monolith
- Decomposing applications into services
- Using an API gateway
- Inter-service communication mechanisms

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

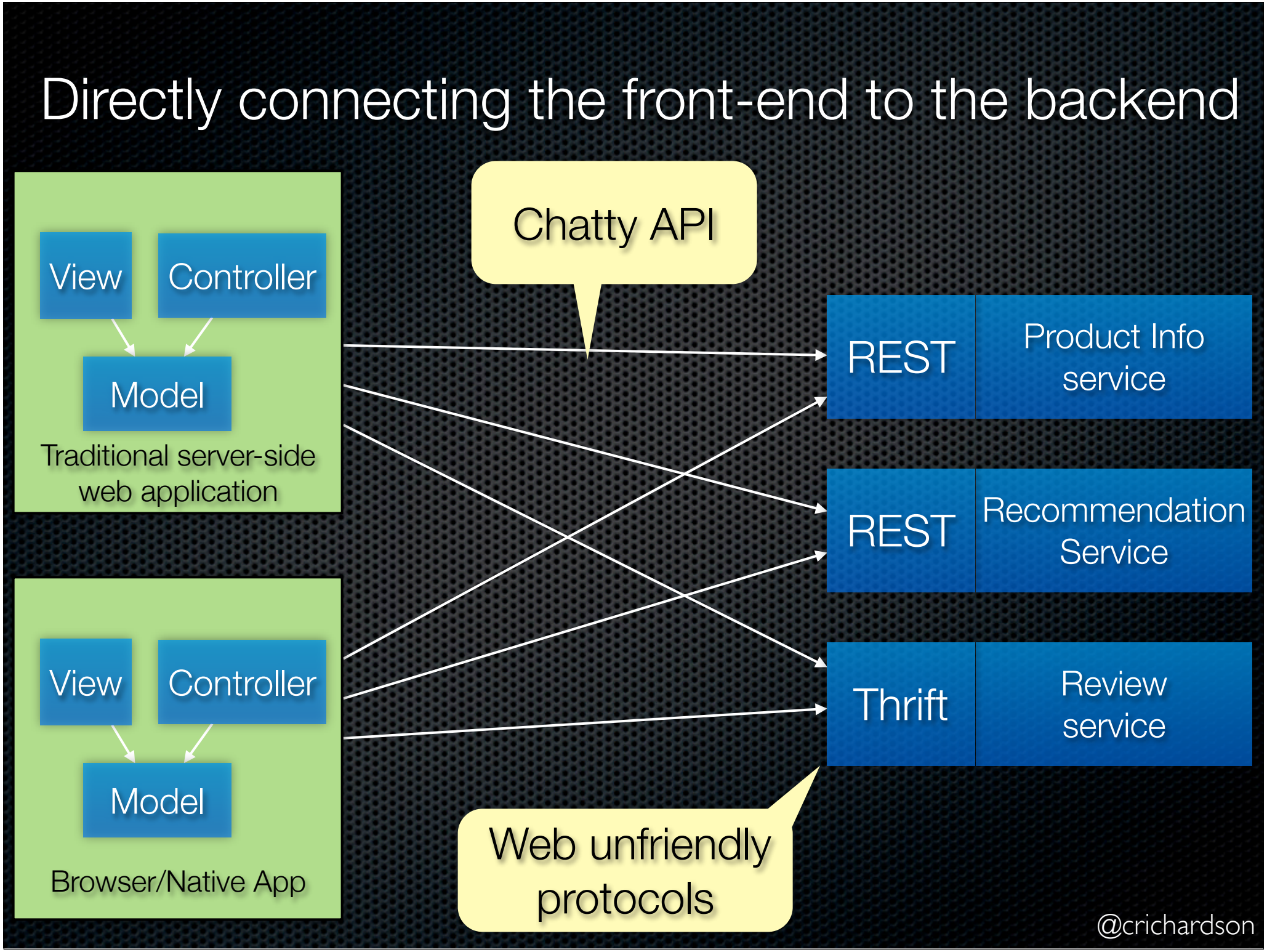
Product Info



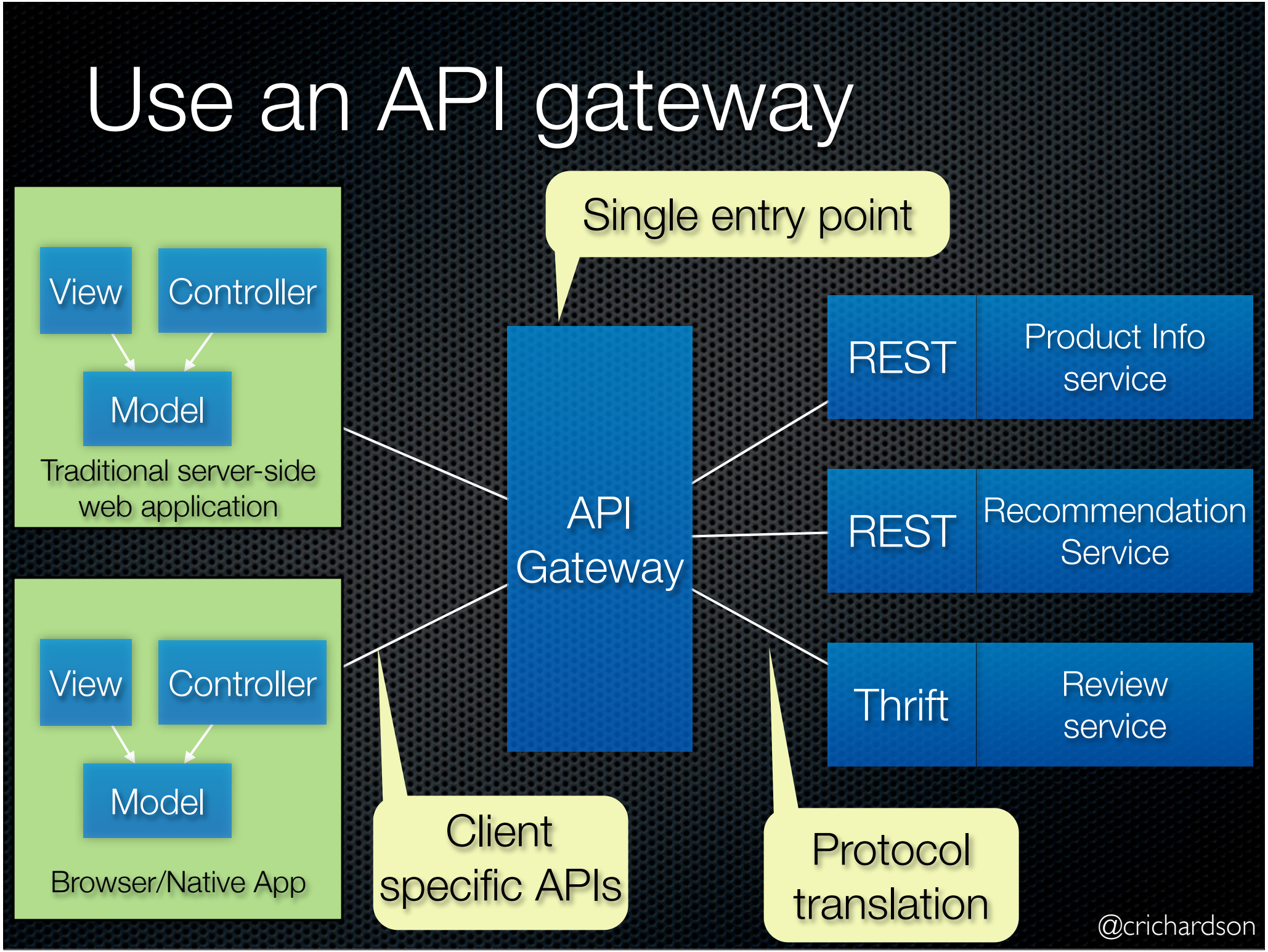
Reviews

Recommendations

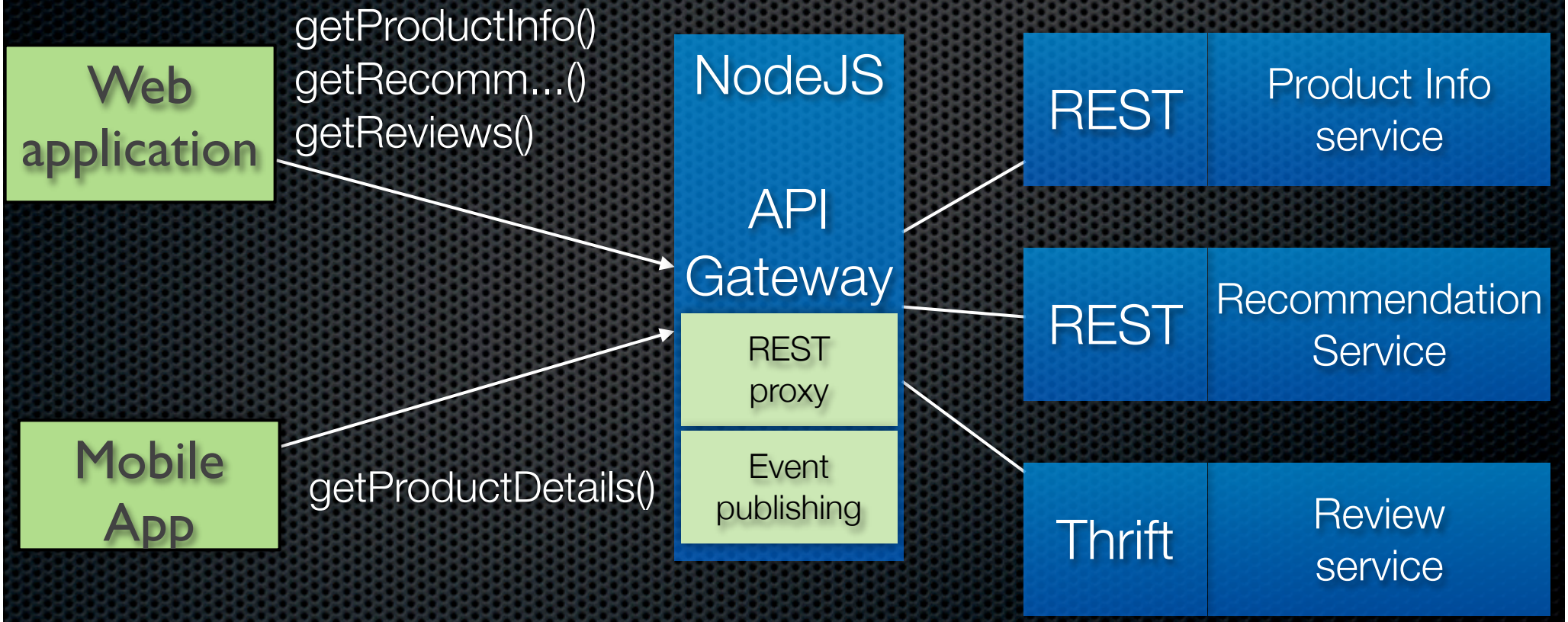
Directly connecting the front-end to the backend



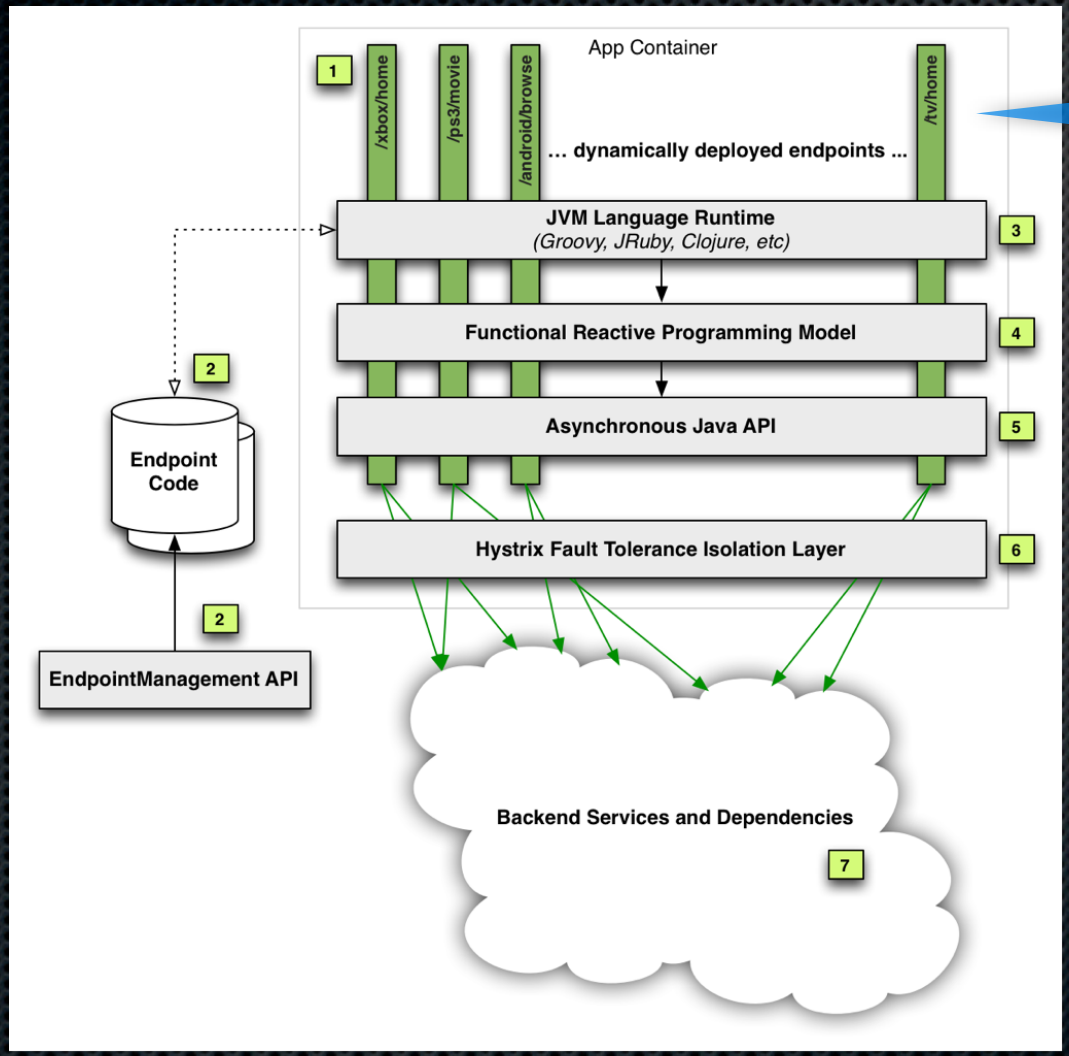
Use an API gateway



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

Agenda

- The (sometimes evil) monolith
- Decomposing applications into services
- Using an API gateway
- Inter-service communication mechanisms

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

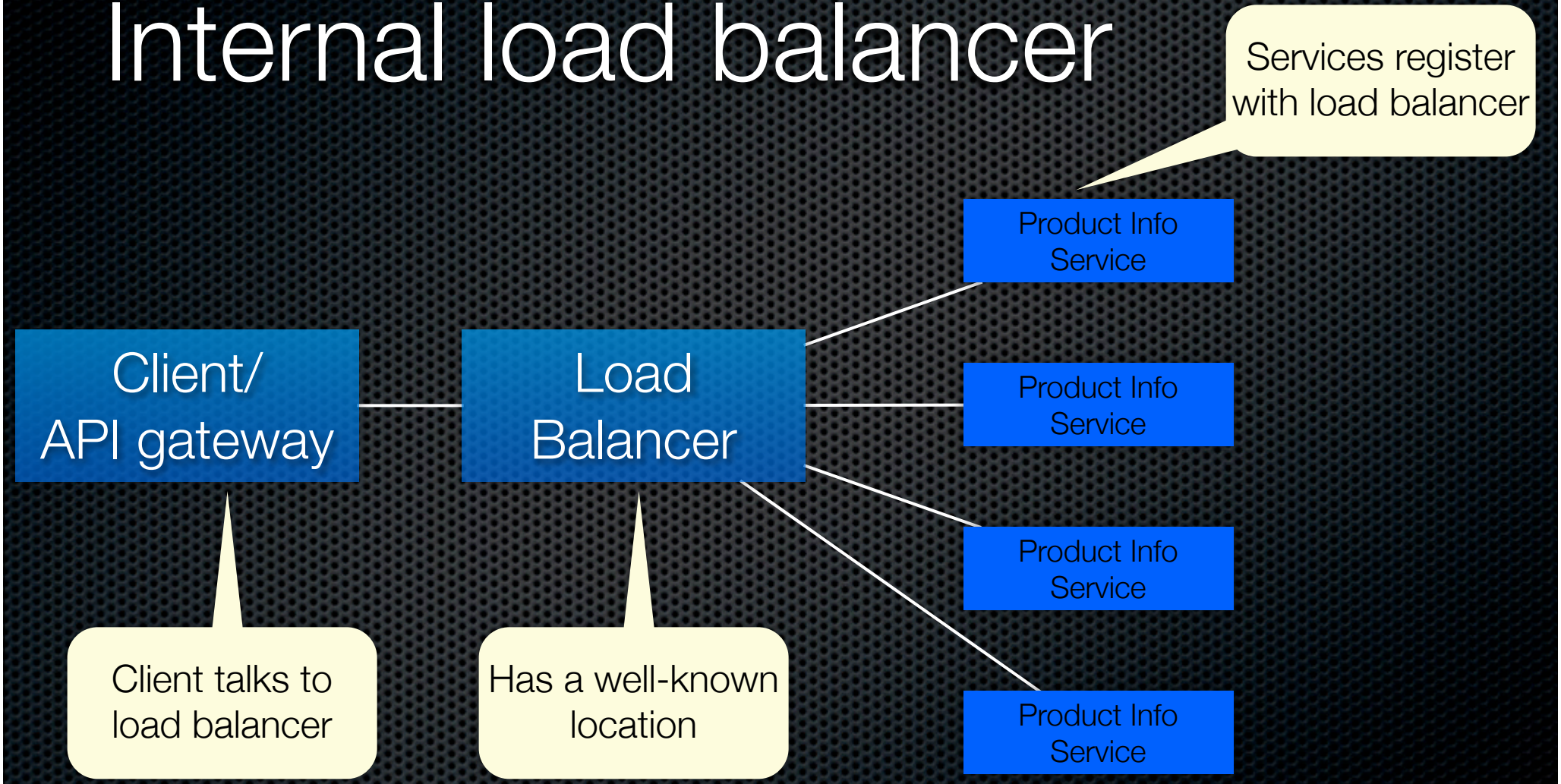
Cons

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

Pros and cons of HTTP

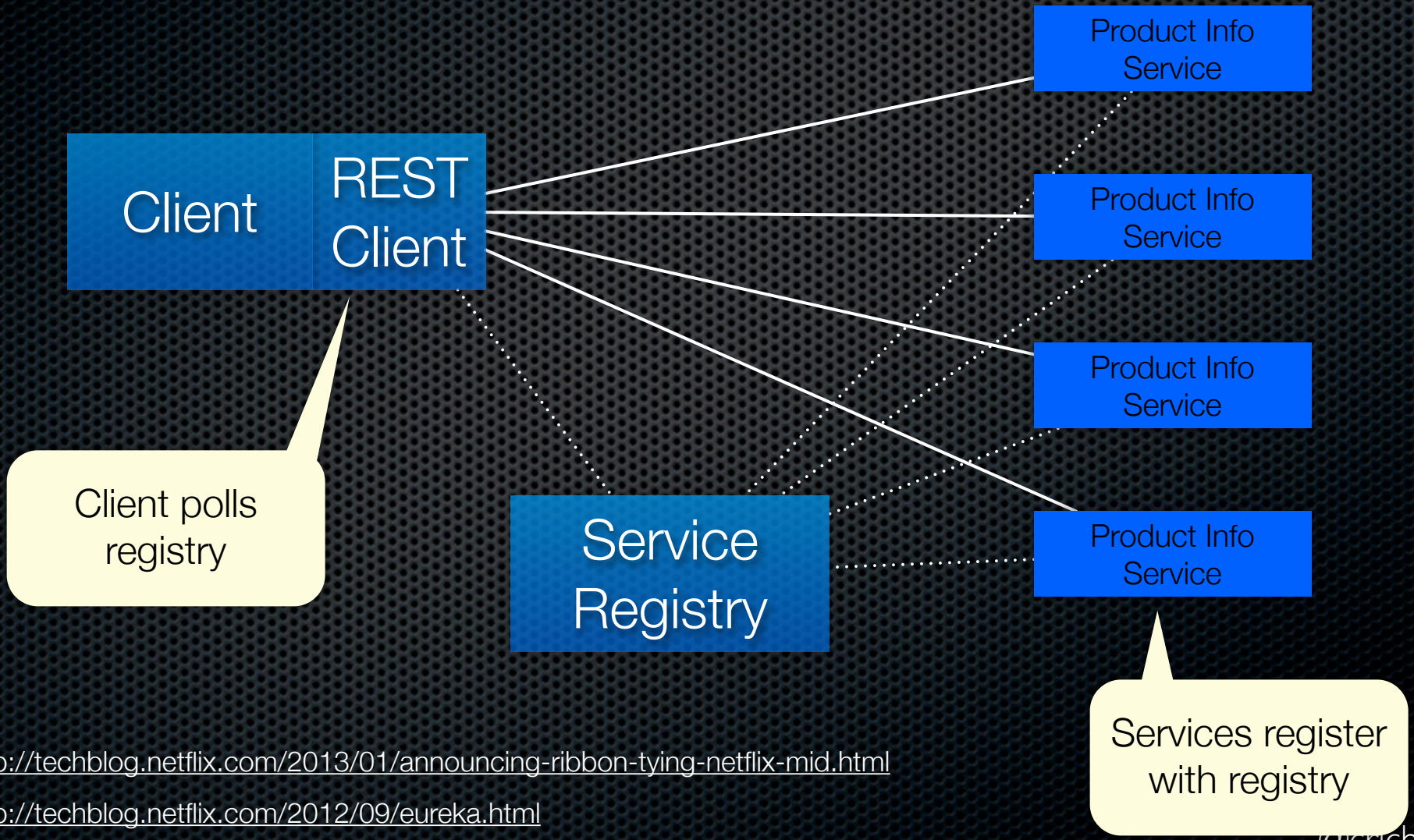
- Pros
  - Simple and familiar
  - Request/reply is easy
  - Firewall friendly
  - No intermediate broker
- Cons
  - Only supports request/reply
  - Server must be available
  - Client needs to discover URL(s) of server(s)

Discovery option #1: Internal load balancer



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

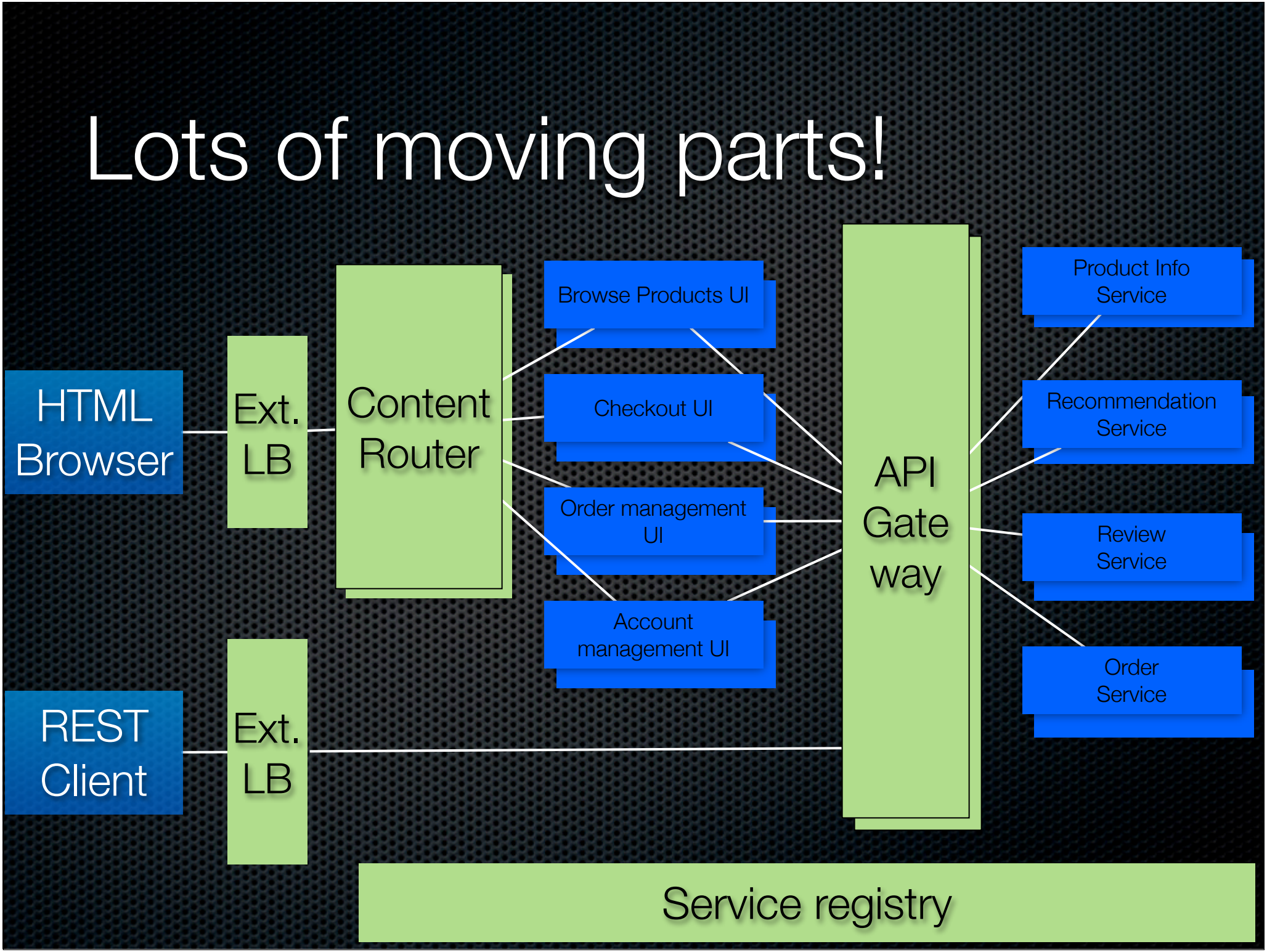
Discovery option #2: client-side load balancing



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

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

Lots of moving parts!

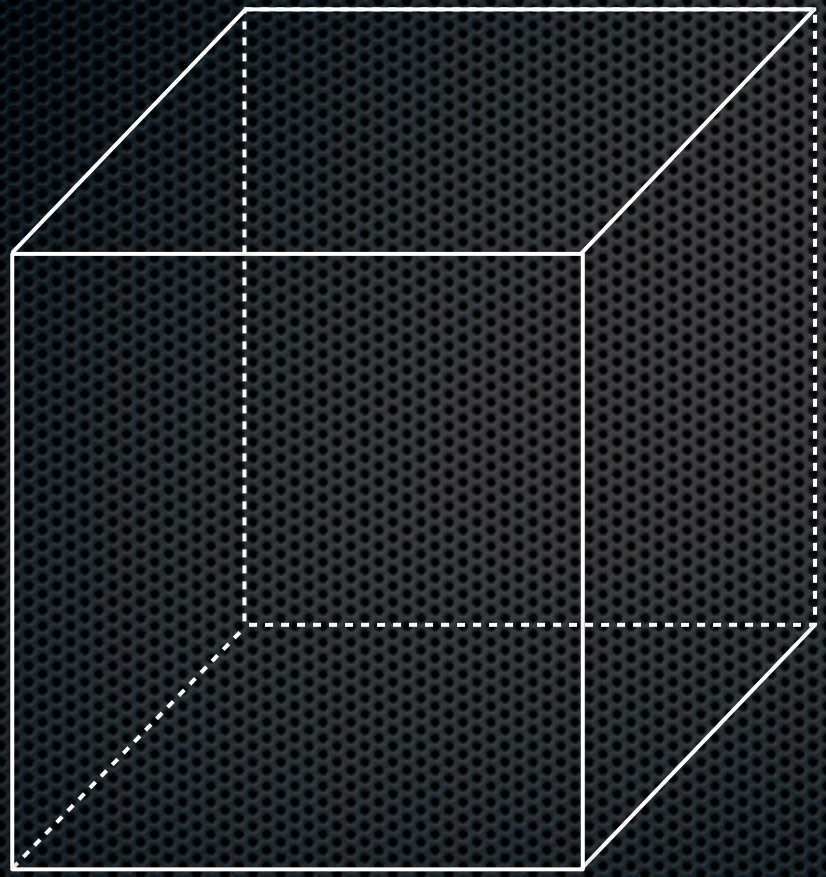


Summary

Monolithic applications are simple to develop and deploy

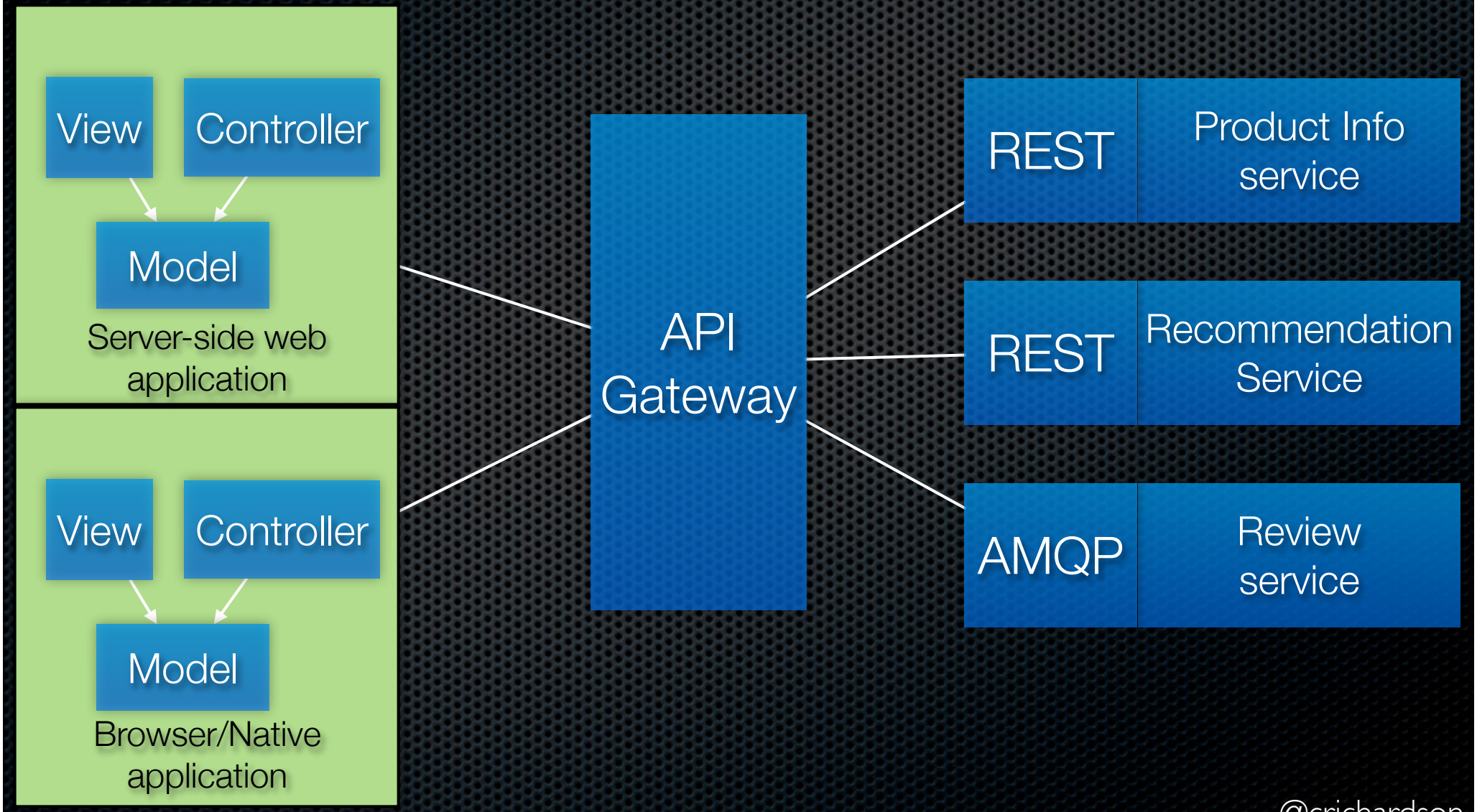
BUT have significant drawbacks

Apply the scale cube

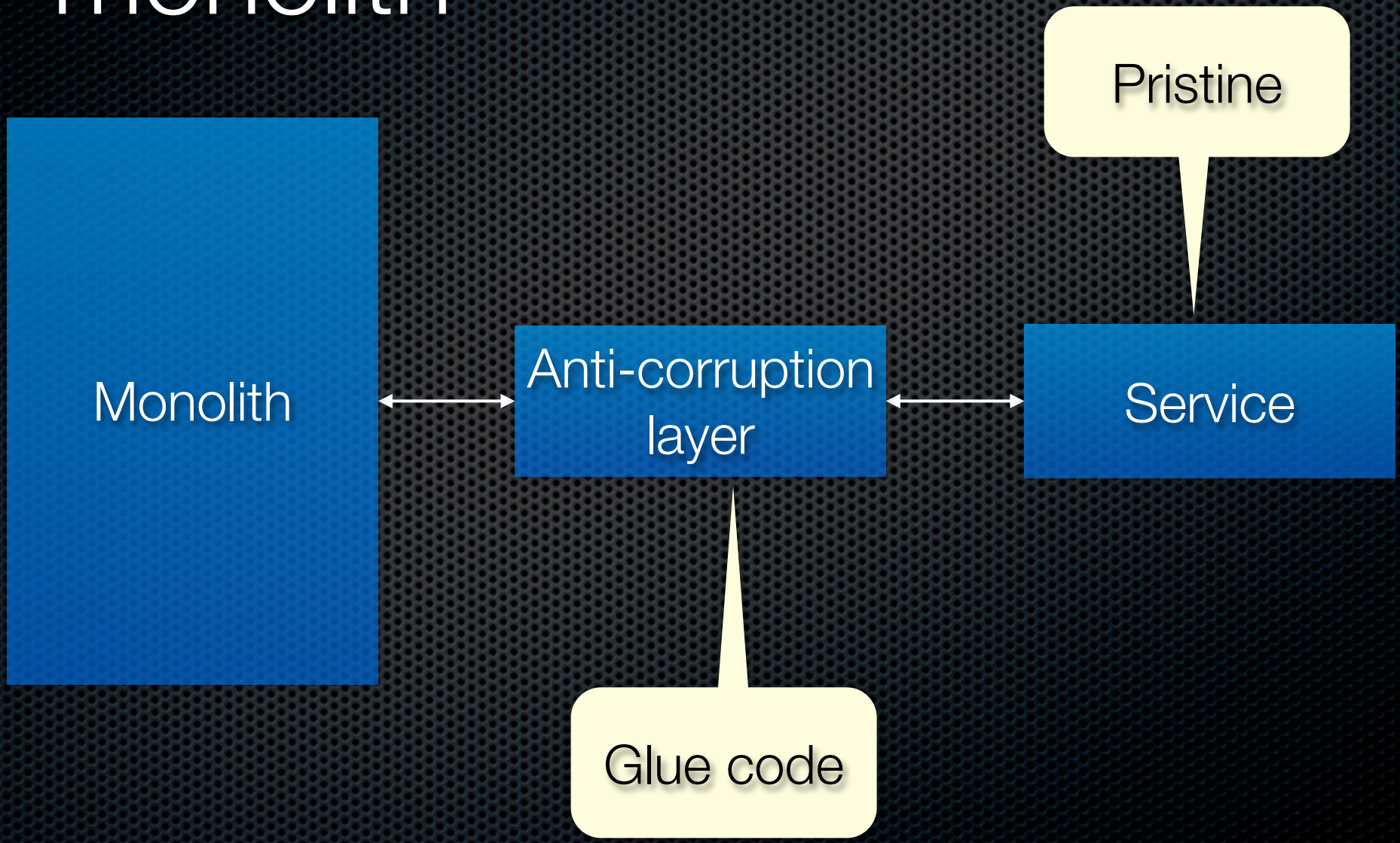


- Modular, polyglot, and scalable applications
- Services developed, deployed and scaled independently

Use a modular, polyglot architecture



Start refactoring your monolith



Come to my 11.30 am talk to learn more....

Building microservices with Scala, functional domain models and Spring Boot


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