Decomposing applications for deployability and scalability

Chris Richardson

Author of POJOs in Action

Founder of the original CloudFoundry.com

 @crichardson

chris.richardson@springsource.com

http://plainoldobjects.com

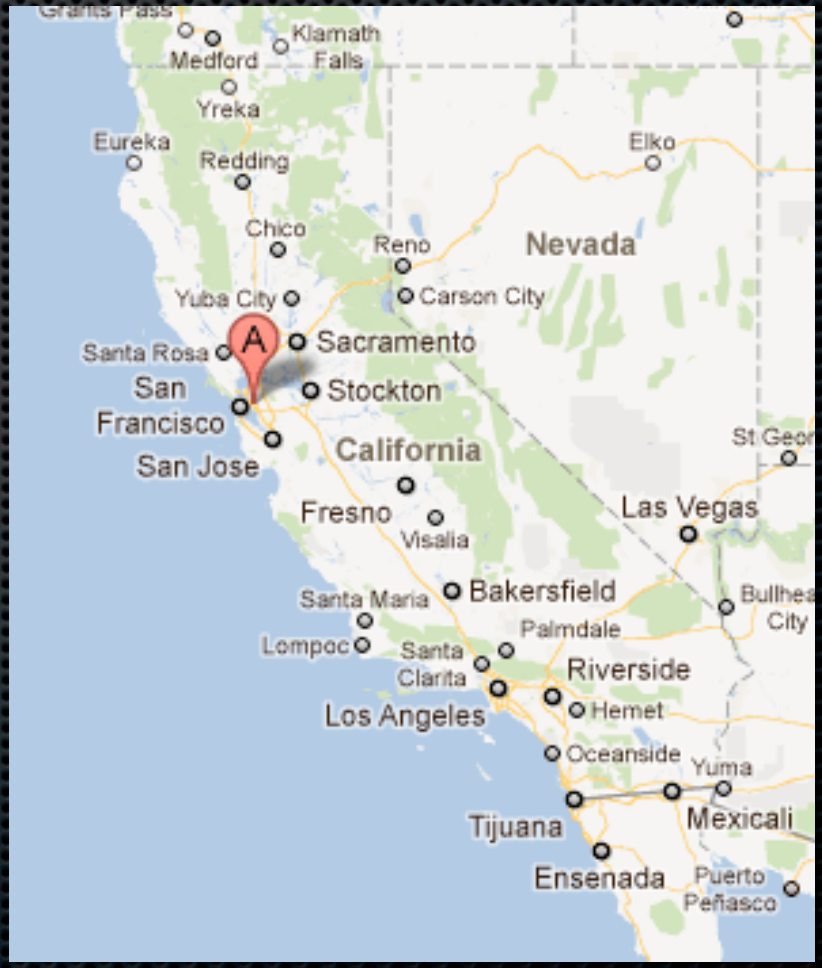
Presentation goal

How decomposing applications improves deployability and scalability

and

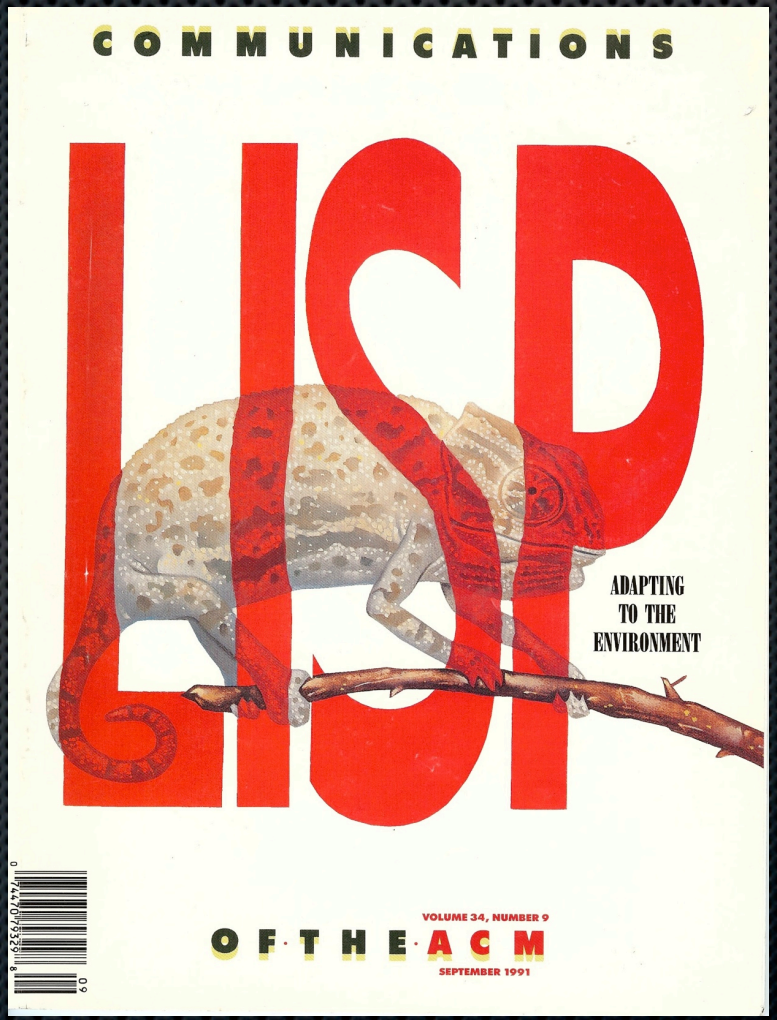
simplifies the adoption of new technologies

About Chris



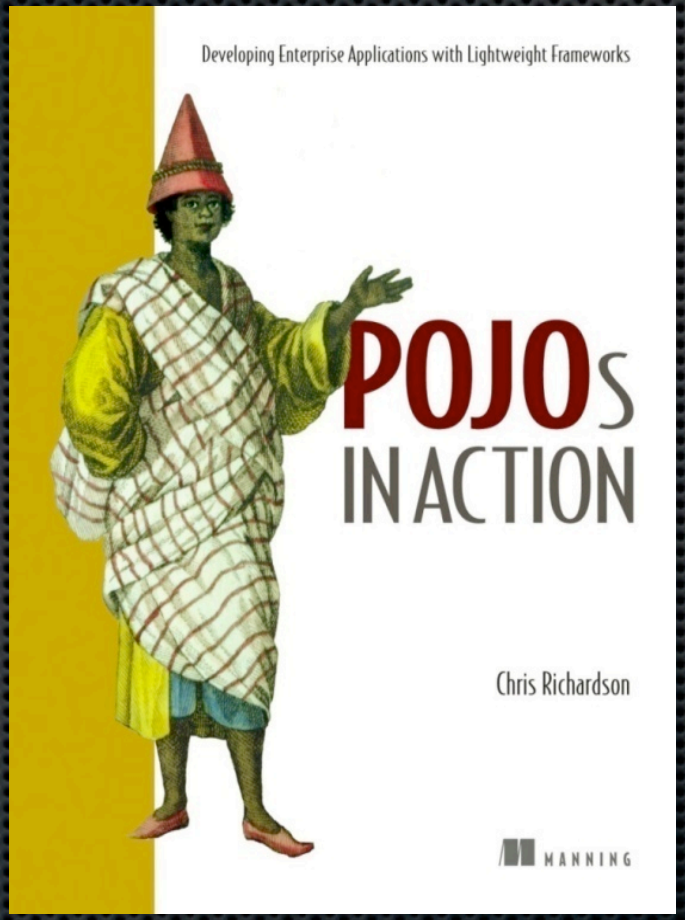
@crichardson

(About Chris)

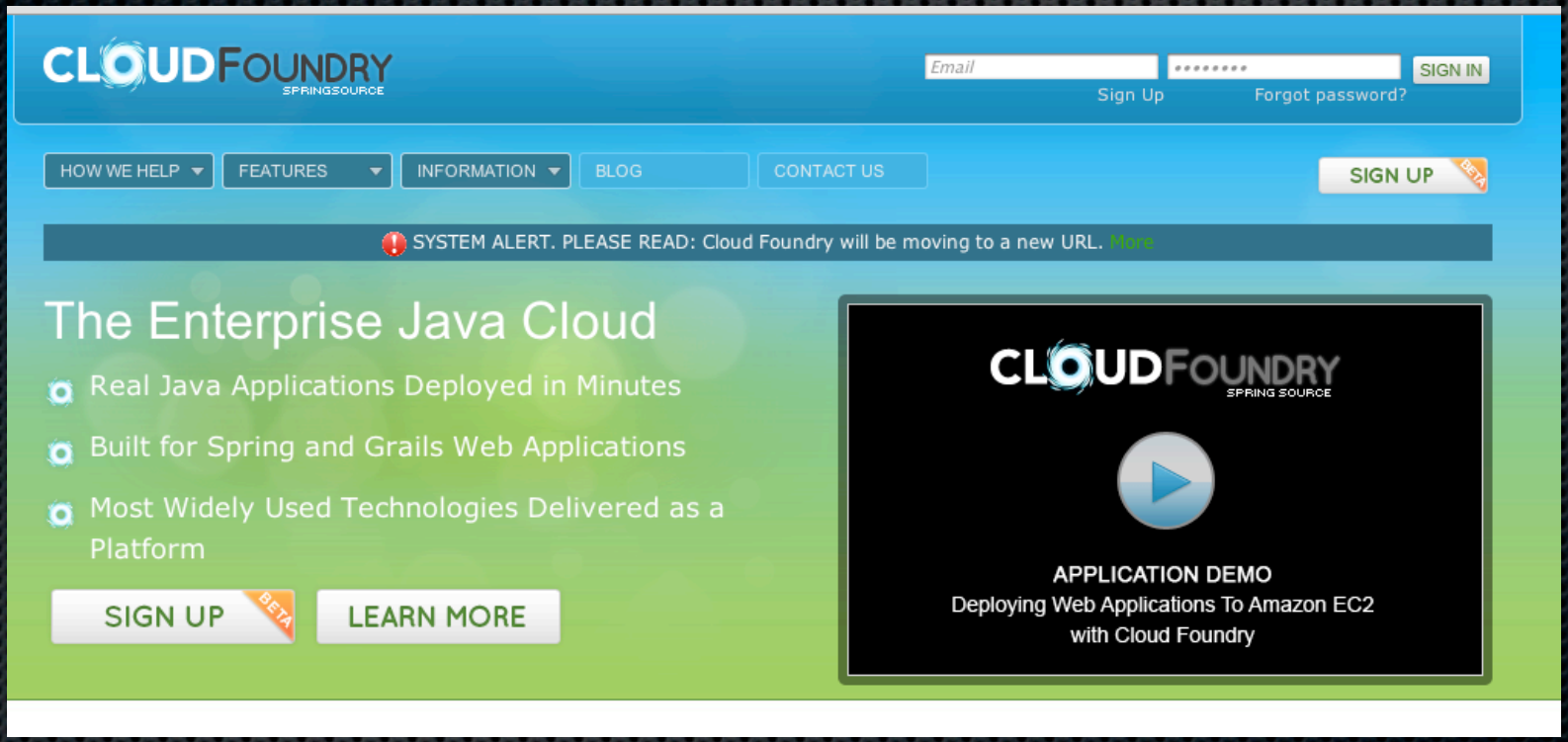


@crichardson

About Chris()

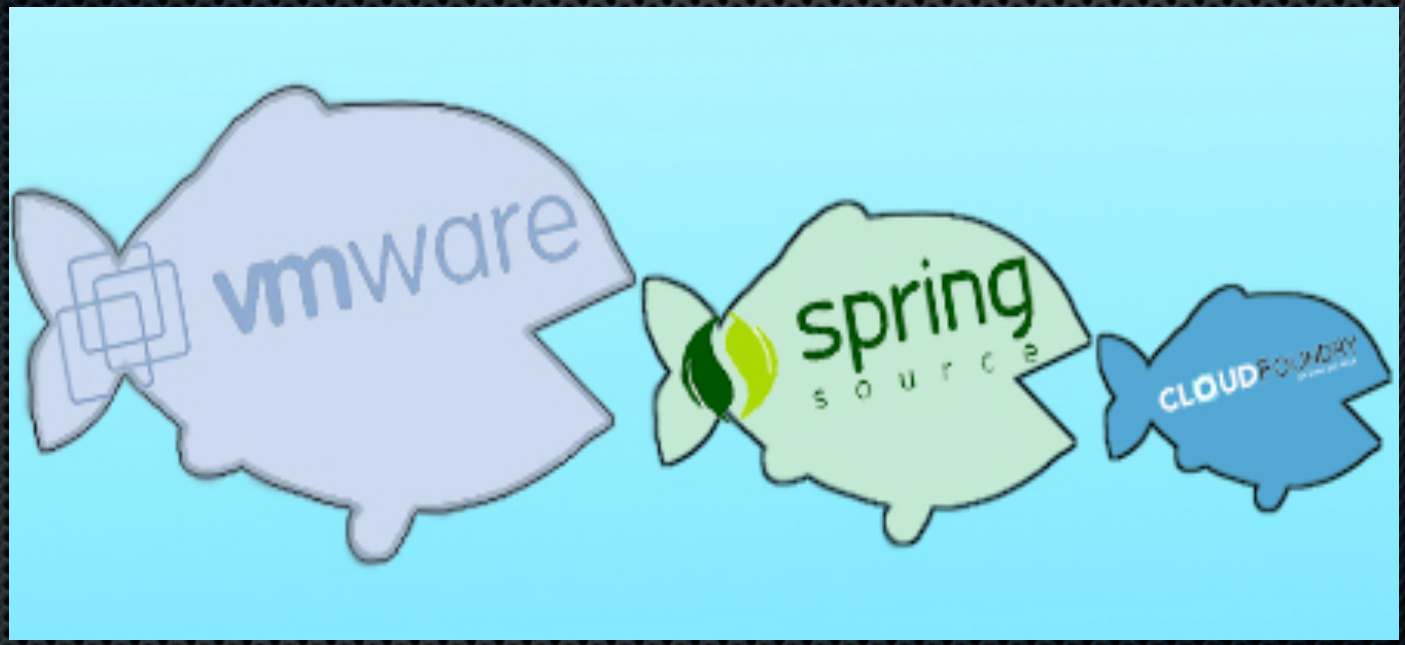

Developing Enterprise Applications with Lightweight Frameworks
POJOs
IN ACTION
Chris Richardson
MANNING

About Chris



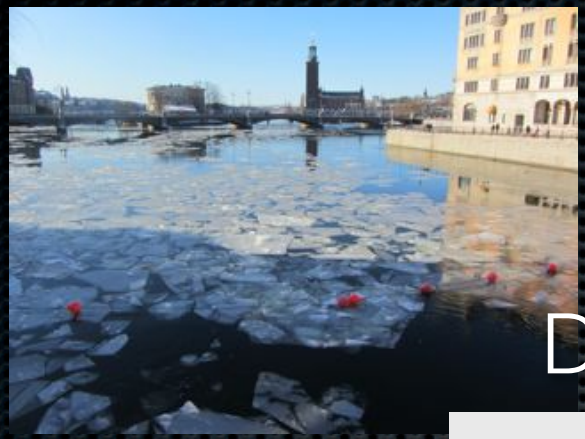
@crichardson

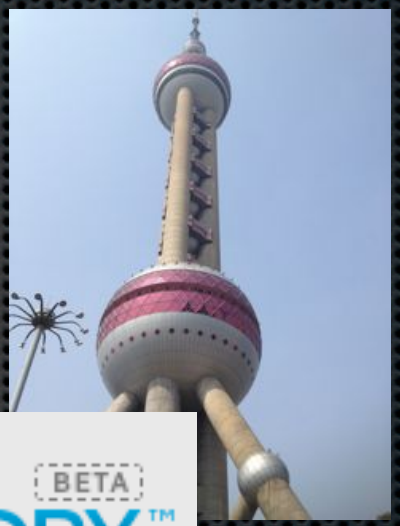
About Chris



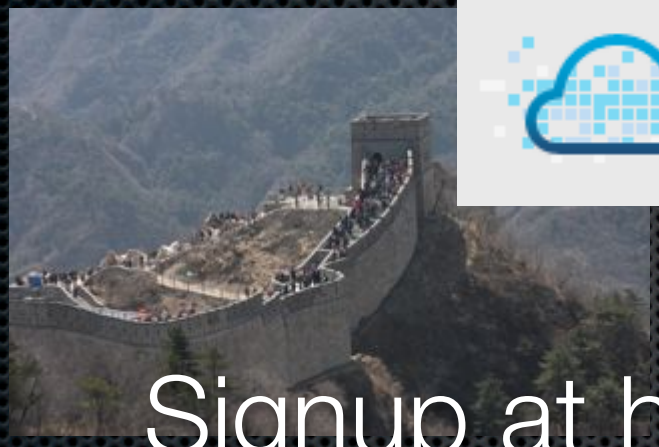
http://www.theregister.co.uk/2009/08/19/springsource_cloud_foundry/

vmc push About-Chris





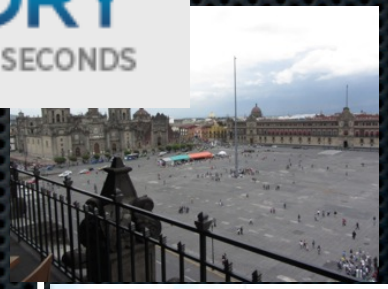
Developer Advocate



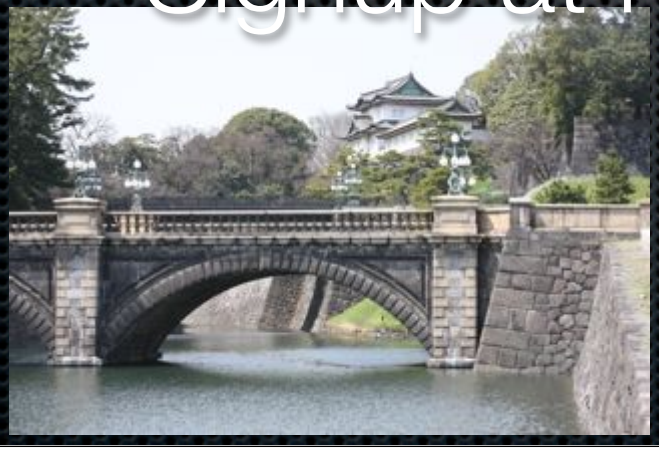
BETA

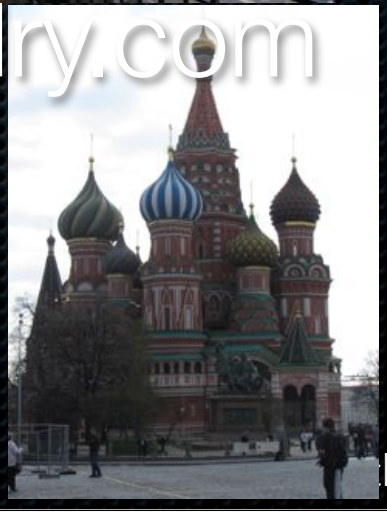
CLOUD FOUNDRY™

DEPLOY & SCALE YOUR APPLICATIONS IN SECONDS

Signup at http://cloudfoundry.com





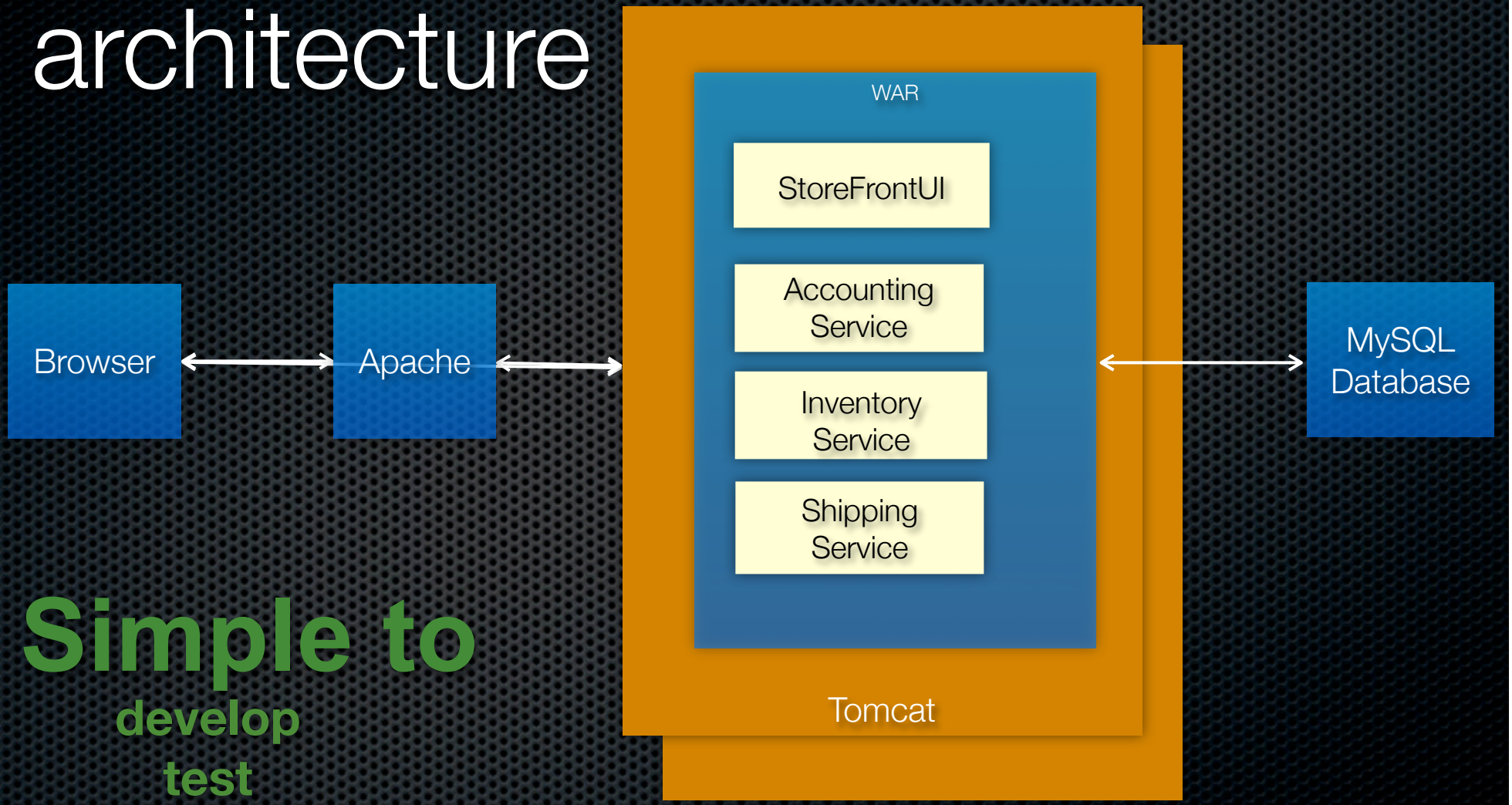
hardson

Agenda

- The (sometimes evil) monolith
- Decomposing applications into services
- How do services communicate?
- Presentation layer design

Let's imagine you are building an e-commerce application

Traditional web application architecture

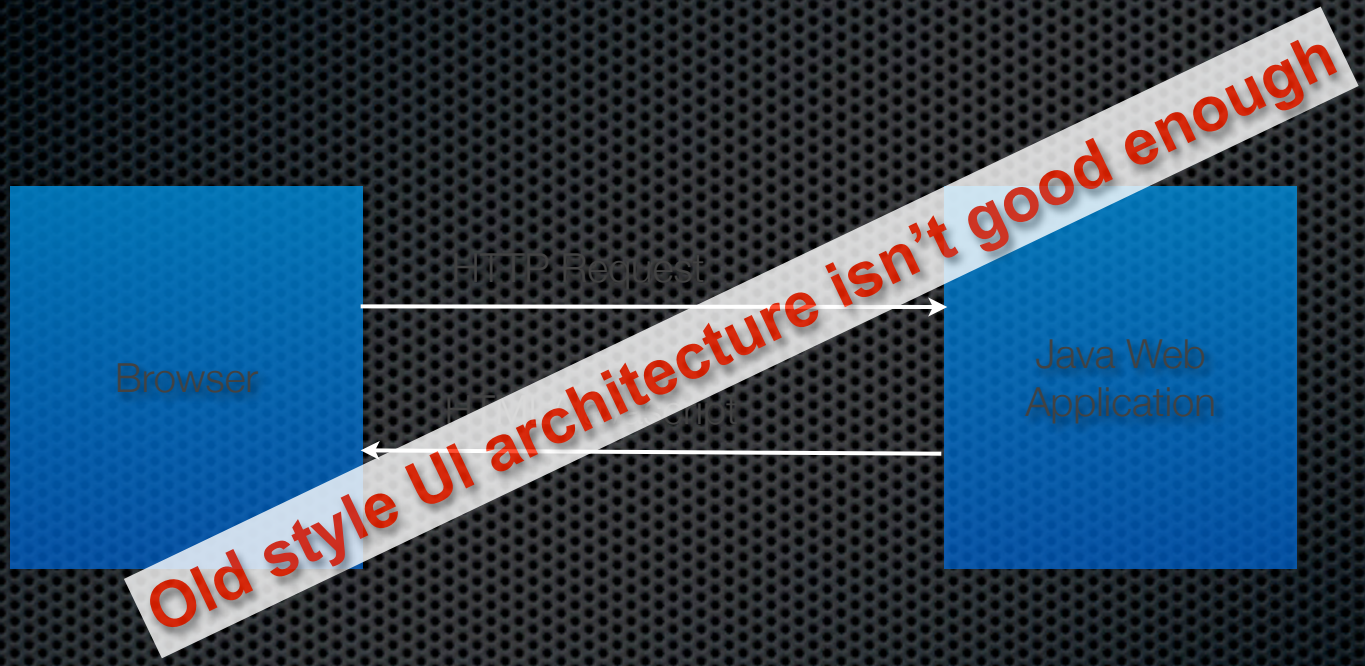


Simple to

develop
test
deploy
scale

But there are problems with a monolithic architecture

Users expect a rich, dynamic and interactive experience

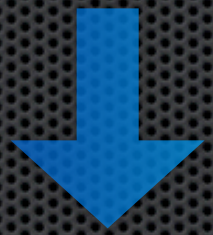


Real-time web ≅ NodeJS

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



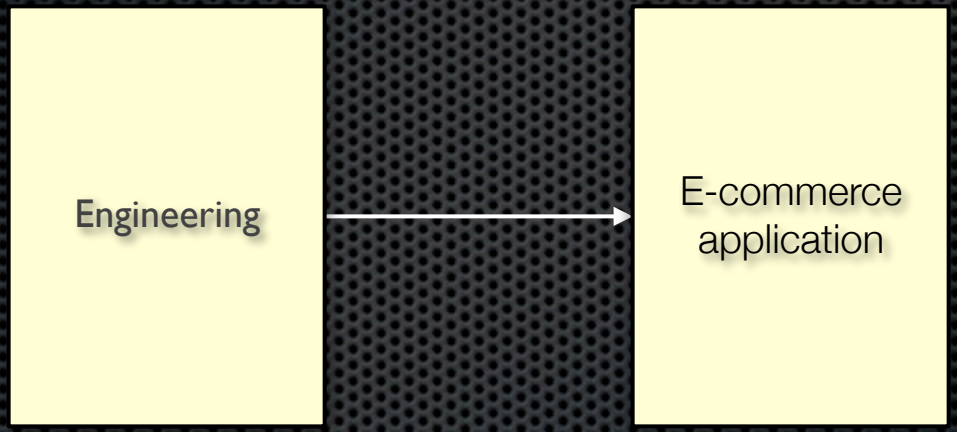
Fear of change

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

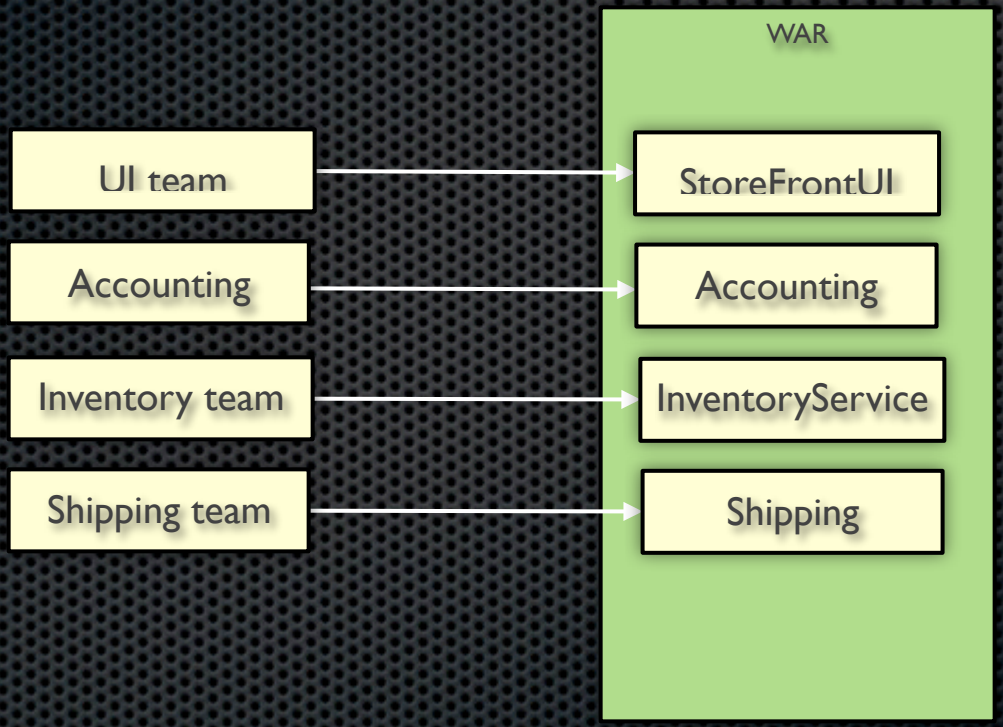
Overloads your IDE and container

Slows down development

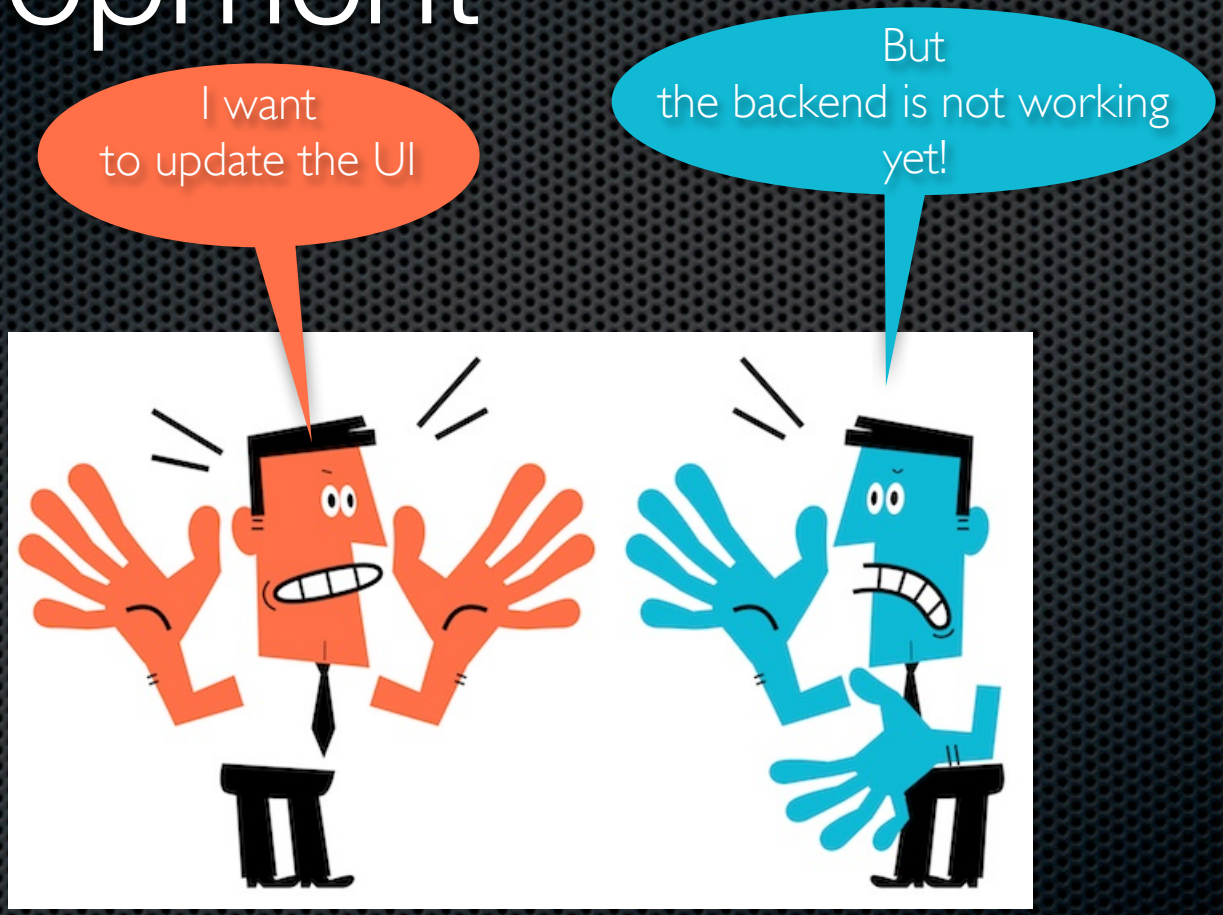
Obstacle to scaling development



Obstacle to scaling development



Obstacle to scaling development

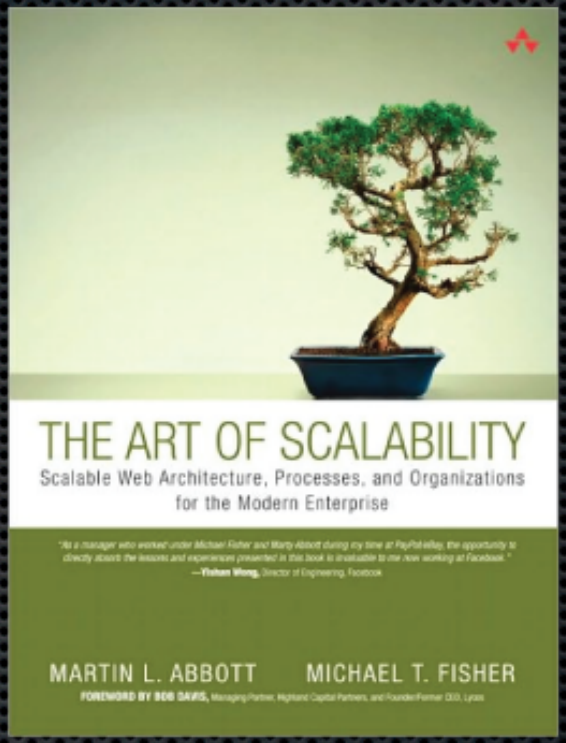


Lots of coordination and communication required

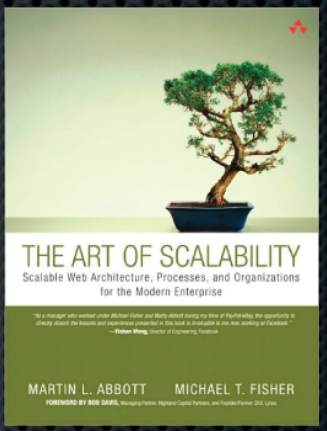
Requires long-term commitment to a technology stack

Agenda

- The (sometimes evil) monolith
- Decomposing applications into services
- How do services communicate?
- Presentation layer design

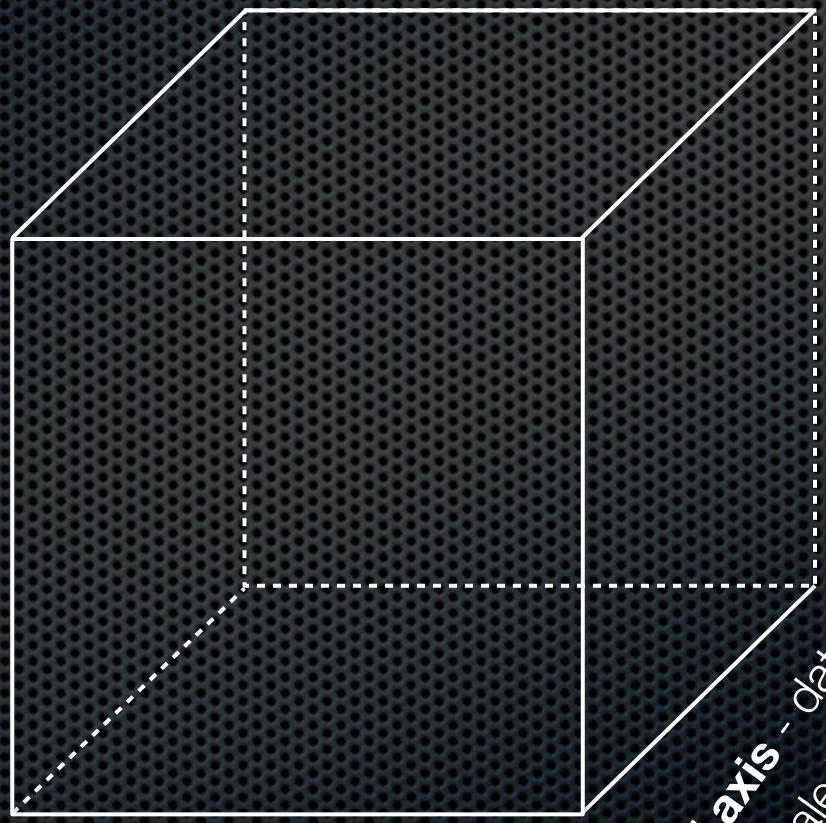

THE ART OF SCALABILITY
Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise
MARTIN L. ABBOTT
MICHAEL T. FISHER

The scale cube



Y axis - functional decomposition

Scale by splitting different things



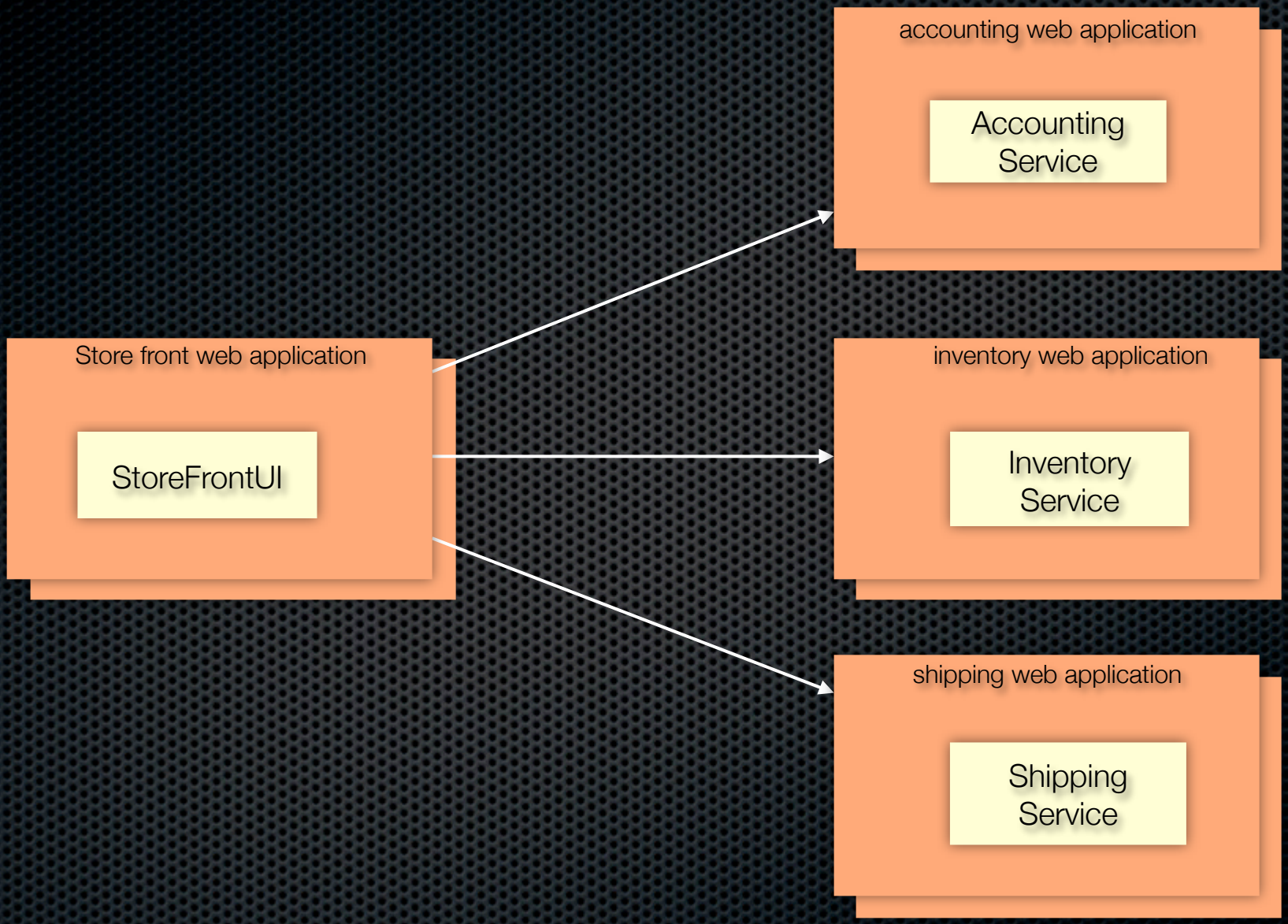
Z axis - data partitioning
Scale by splitting similar things

X axis
- horizontal duplication

@crichardson

Y-axis scaling - application level

Y-axis scaling - application level



Apply X axis cloning and/or Z axis partitioning to each service

Partitioning strategies...

- Partition by verb, e.g. shipping service
- Partition by noun, e.g. inventory service
- 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
```

For more on micro-services see @fgeorge52

Real world examples

http://techblog.netflix.com/

http://highscalability.com/amazon-architecture

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

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

There are drawbacks

Complexity

See Steve Yegge's Google Platforms Rant re Amazon.com

Multiple databases
&
Transaction management

Implementing features that span multiple services

When to use it?

In the beginning:

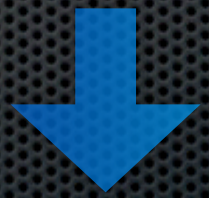
- You don't need it
- It will slow you down

Later on:

- You need it
- Refactoring is painful

But there are many benefits

- Scales development: develop, deploy and scale each service independently
- Update UI 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

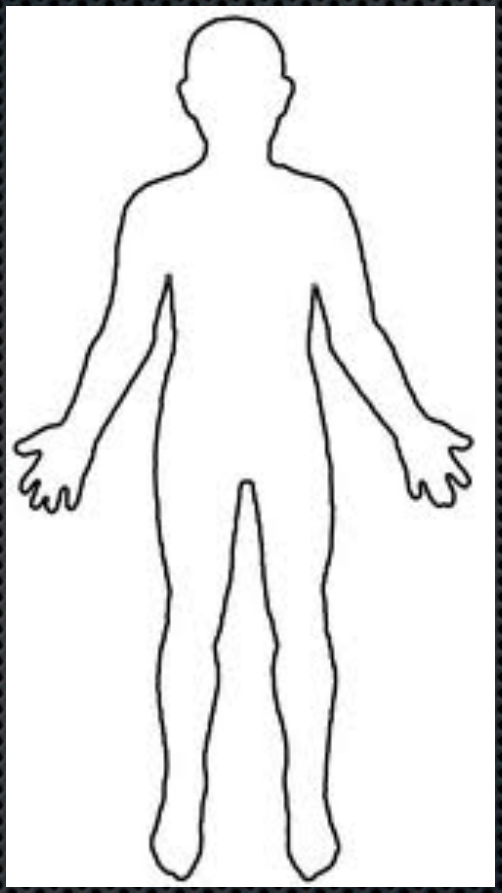
Service-level

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

If services are small...

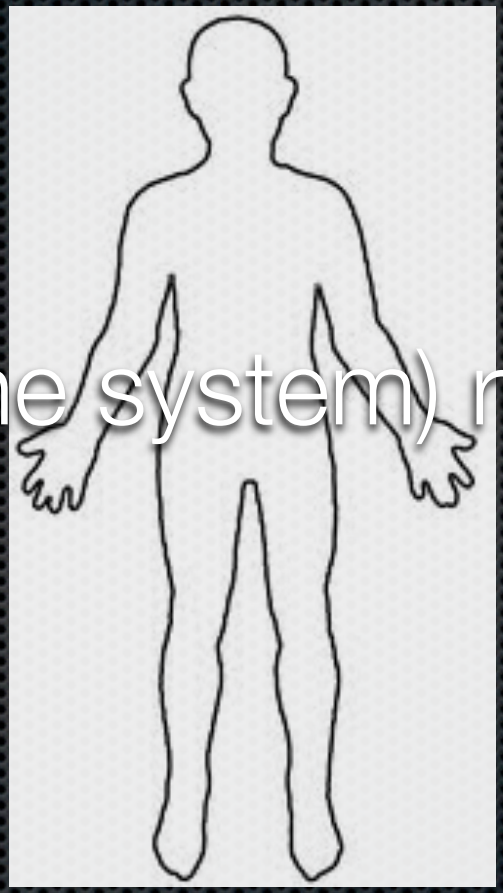
- Regularly rewrite using a better technology stack
- Adapt system to changing requirements and better technology without a total rewrite
- Pick the **best developers** rather than **best <pick a language> developers** ⇒ polyglot culture

The human body as a system



50 to 70 billion of your cells die each day

Yet you (the system) remain you



Can we build software systems with these characteristics?

http://dreamsongs.com/Files/DesignBeyondHumanAbilitiesSimp.pdf

http://dreamsongs.com/Files/WhitherSoftware.pdf

Service deployment options

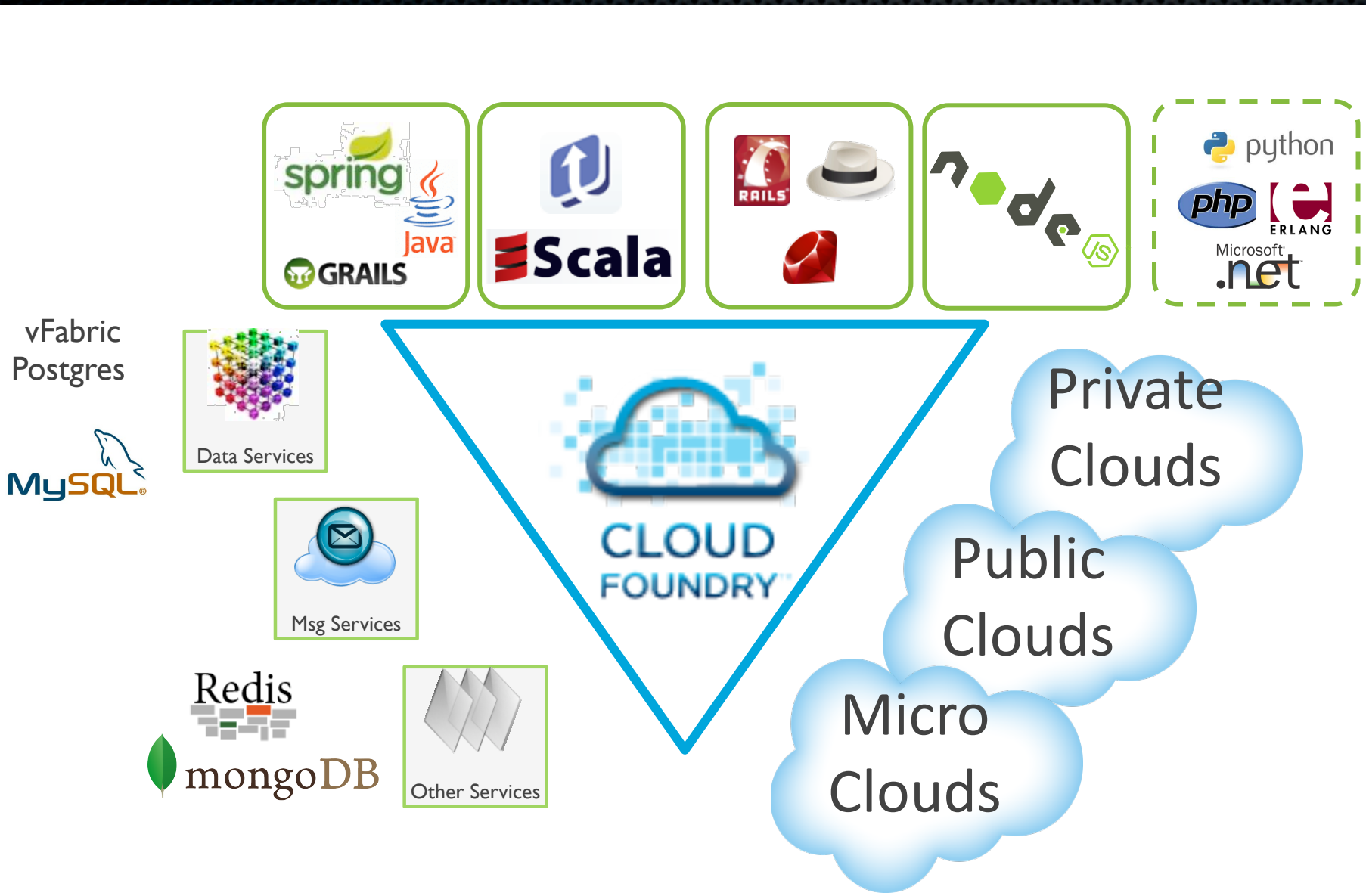
Isolation, manageability

- VM or Physical Machine
- Linux Container/LXC
- JVM
- JAR/WAR/OSGI bundle/...

Density/efficiency

You could do this yourself but ...

PaaS dramatically simplifies deployment



Cloud Foundry features

- One step deployment: vmc push
  - Single application
  - Service-oriented application
- Easy platform service provisioning: vmc create-service
- Simple scaling: vmc instances app-name +/- N
- Health monitoring and automated recovery

Benefits of PaaS

- Simplifies and automates deployment
  - Eliminates barriers to adding new service
  - Eliminates barriers to using a new platform service
- Imposes conventions: packaging, configuration and deployment
  - Enforces consistency
  - Eliminates accidental complexity

Agenda

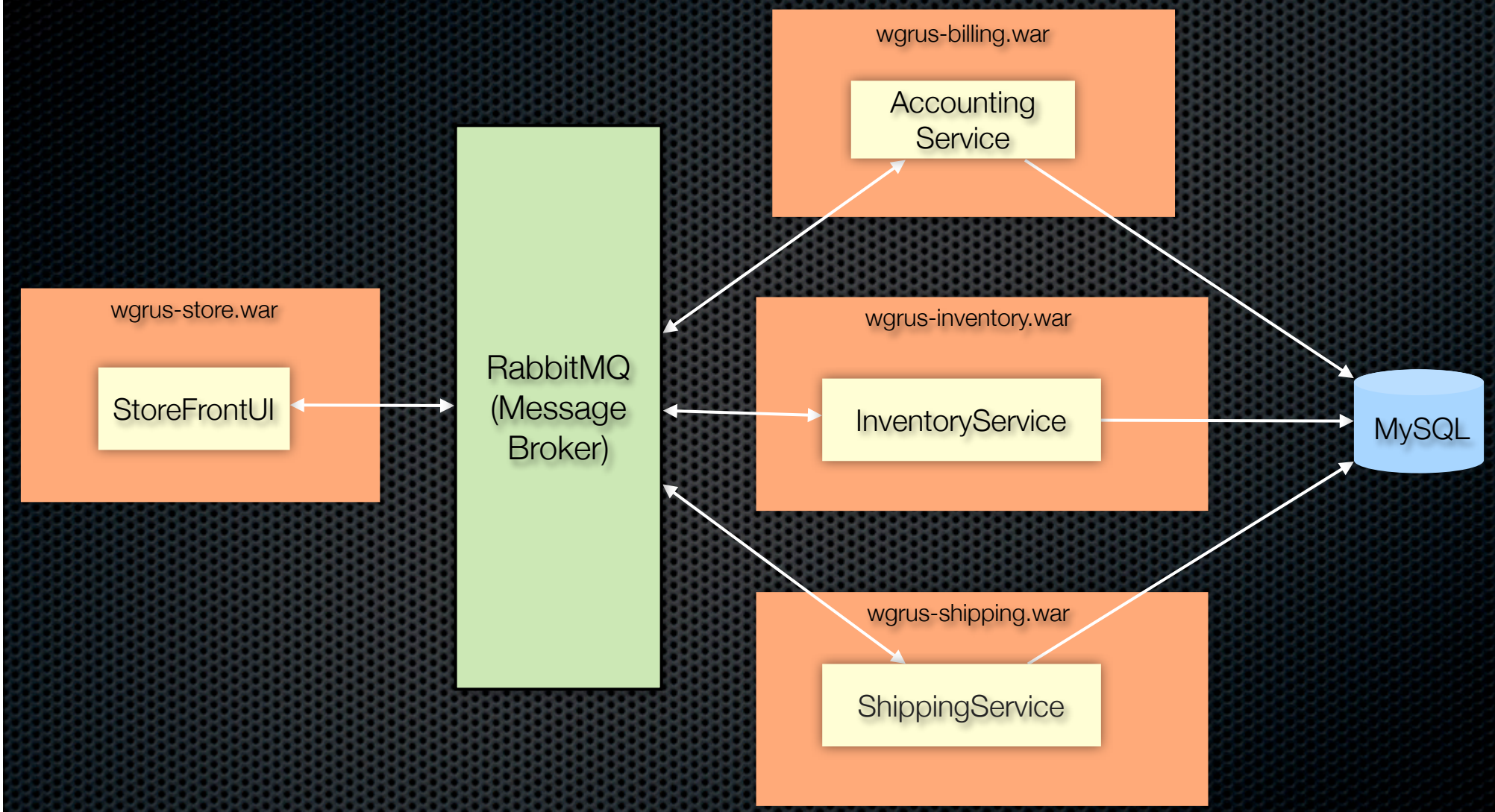
- The (sometimes evil) monolith
- Decomposing applications into services
- How do services communicate?
- Presentation layer design

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

Asynchronous message-based communication



Benefits

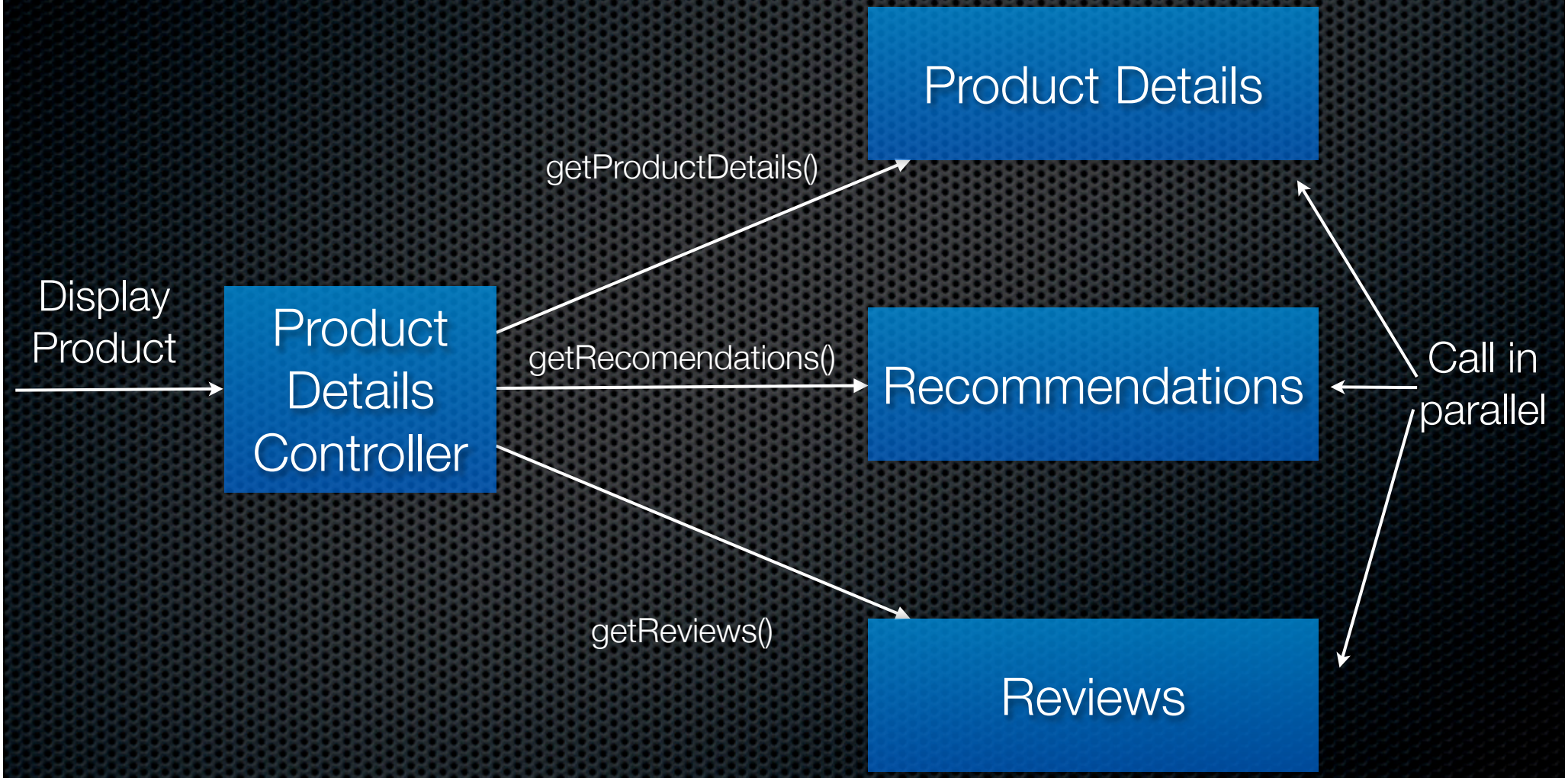
- Decouples caller from server: caller unaware of server's coordinates (URL)
- Message broker buffers message when server is down/slow
- Supports a variety of communication patterns, e.g. point-to-point, pub-sub, ...

Drawbacks

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

Writing code that calls services

The need for parallelism



Futures are a great concurrency abstraction

An object that will contain the result of a concurrent computation

http://en.wikipedia.org/wiki/Futures_and_promises

```
Future<Integer> result =
    executorService.submit(new Callable<Integer>() {... });
```

Java has basic futures. We can do much better....

Better: Futures with callbacks

```
val f : Future[Int] = Future { ... }
f onSuccess {
  case x : Int => println(x)
}
f onFailure {
  case e : Exception => println("exception thrown")
}
```

Guava ListenableFutures,
Java 8 CompletableFuture, **Scala Futures**

Even better: Composable Futures

```
val f1 = Future {  ... ; 1 }
val f2 = Future {  ... ; 2 }

val f4 = f2.map(_ * 2)
assertEquals(4, Await.result(f4, 1 second))

val fzip = f1 zip f2
assertEquals((1, 2), Await.result(fzip, 1 second))


def asyncOp(x : Int) = Future { x * x}
val f = Future.sequence((1 to 5).map { x => asyncOp(x) })
assertEquals(List(1, 4, 9, 16, 25),
             Await.result(f, 1 second))
```

Transforms Future

Combines two futures

Transforms list of futures to a future containing a list

Using Scala futures

```
def callB() : Future[...] = ...
def callC() : Future[...] = ...
def callD() : Future[...] = ...

val future = for {
  (b, c) <- callB() zip callC();
  d <- callD(b, c)
 } yield d

val result = Await.result(future, 1 second)
```

Two calls execute in parallel

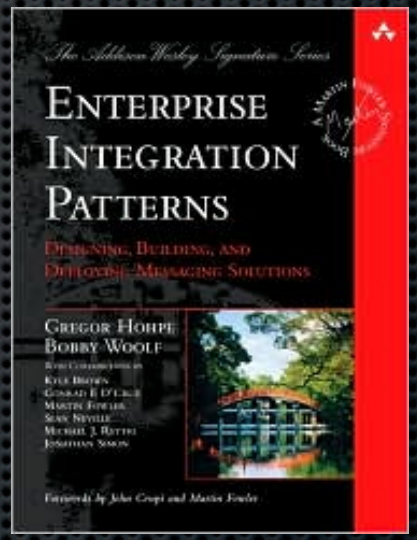
And then invokes D

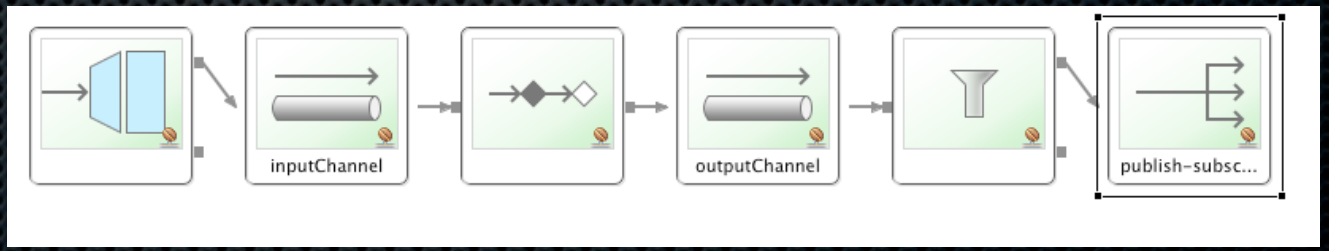
Get the result of D

Scala Futures

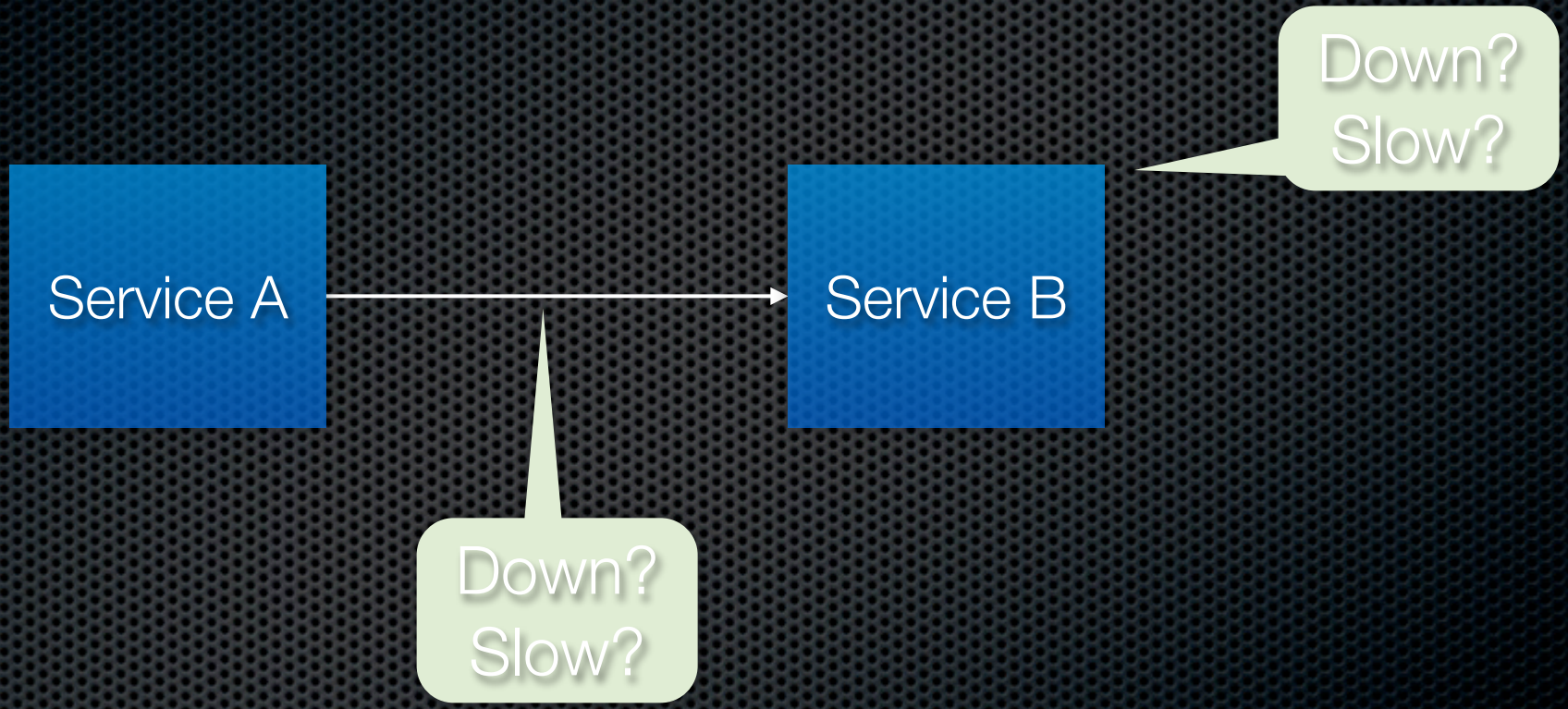
Spring Integration

- Provides the building blocks for a pipes and filters architecture
- Enables development of application components that are
  - loosely coupled
  - insulated from messaging infrastructure
- Messaging defined declaratively





Handling partial failures

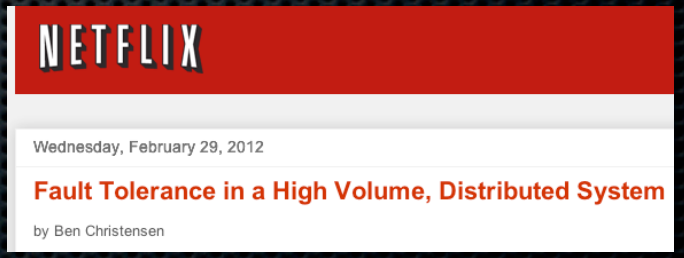


About Netflix

> 1B API calls/day

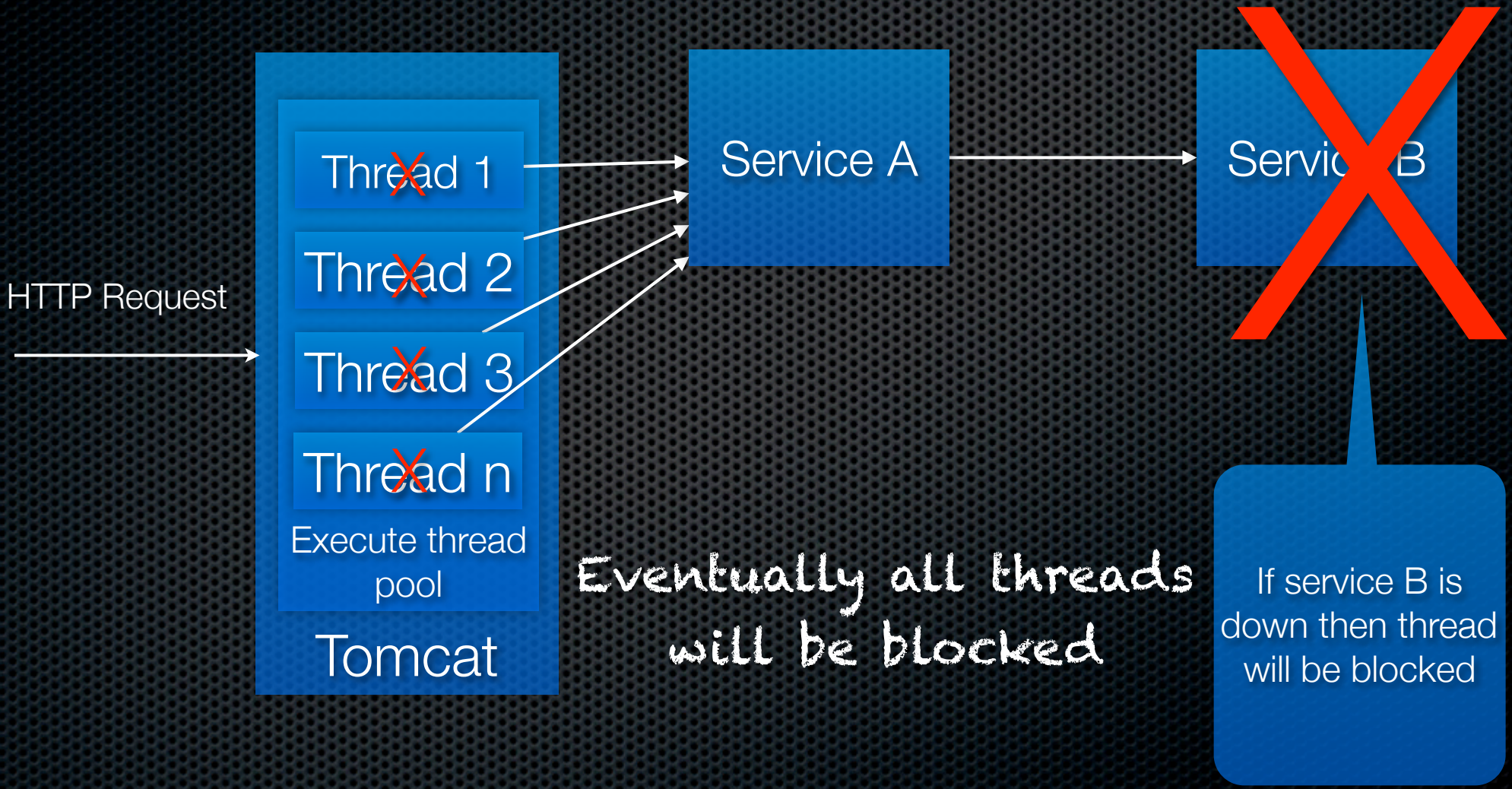
1 API call ⇒ average 6 service calls

Fault tolerance is essential

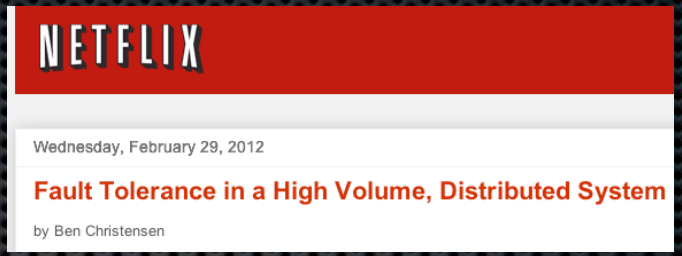


http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

How to run out of threads



Their approach

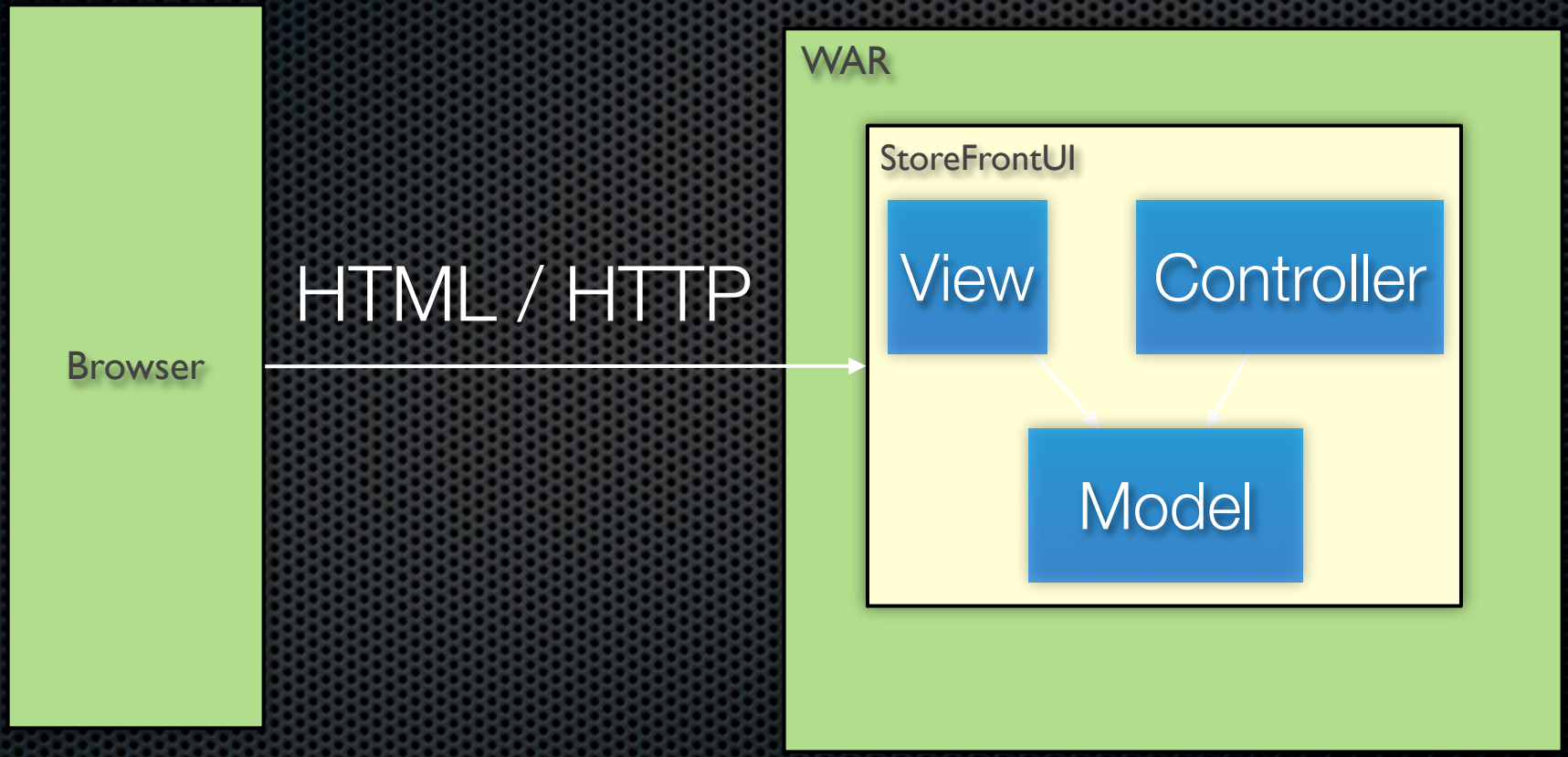


- Network timeouts and retries
- Invoke remote services via a bounded thread pool
- Use the Circuit Breaker pattern
- On failure:
  - return default/cached data
  - return error to caller
- https://github.com/Netflix/Hystrix

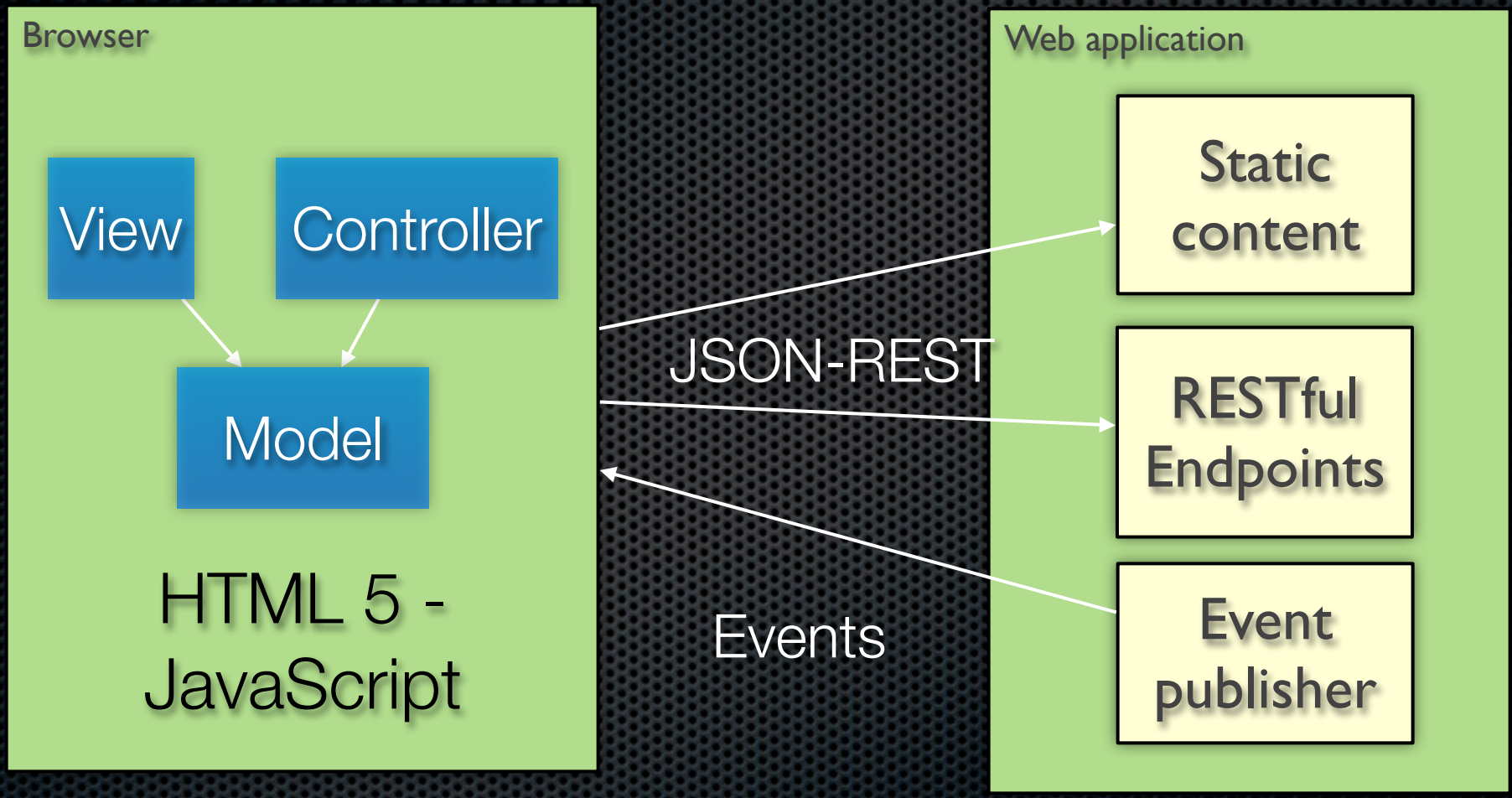
Agenda

- The (sometimes evil) monolith
- Decomposing applications into services
- How do services communicate?
- Presentation layer design

Presentation layer evolution....



...Presentation layer evolution



No elaborate, server-side web framework required

How to publish events to browser?

WebSockets might not be supported and lacks features

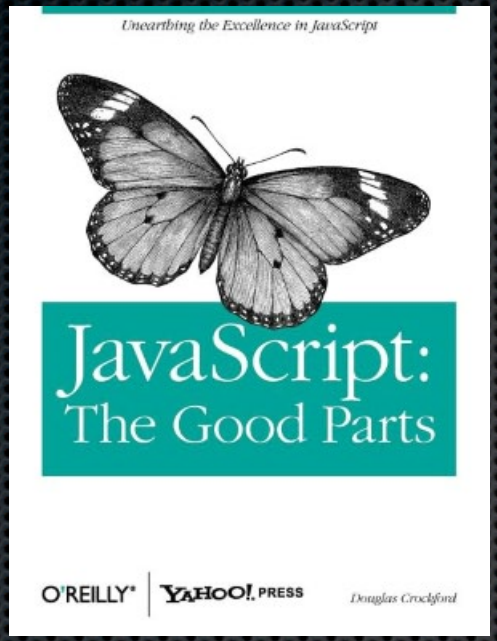
NodeJS is the fashionable technology

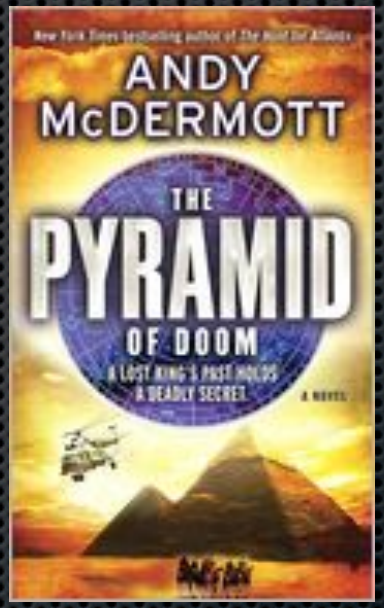
@crichardson

Why NodeJS?

- Familiar Javascript
- High-performance, scalable event-driven, non-blocking I/O model
- Compact runtime
- Over 26,000 modules developed by the community
- Simple event publishing - with or without websockets - using socket.io or SockJS

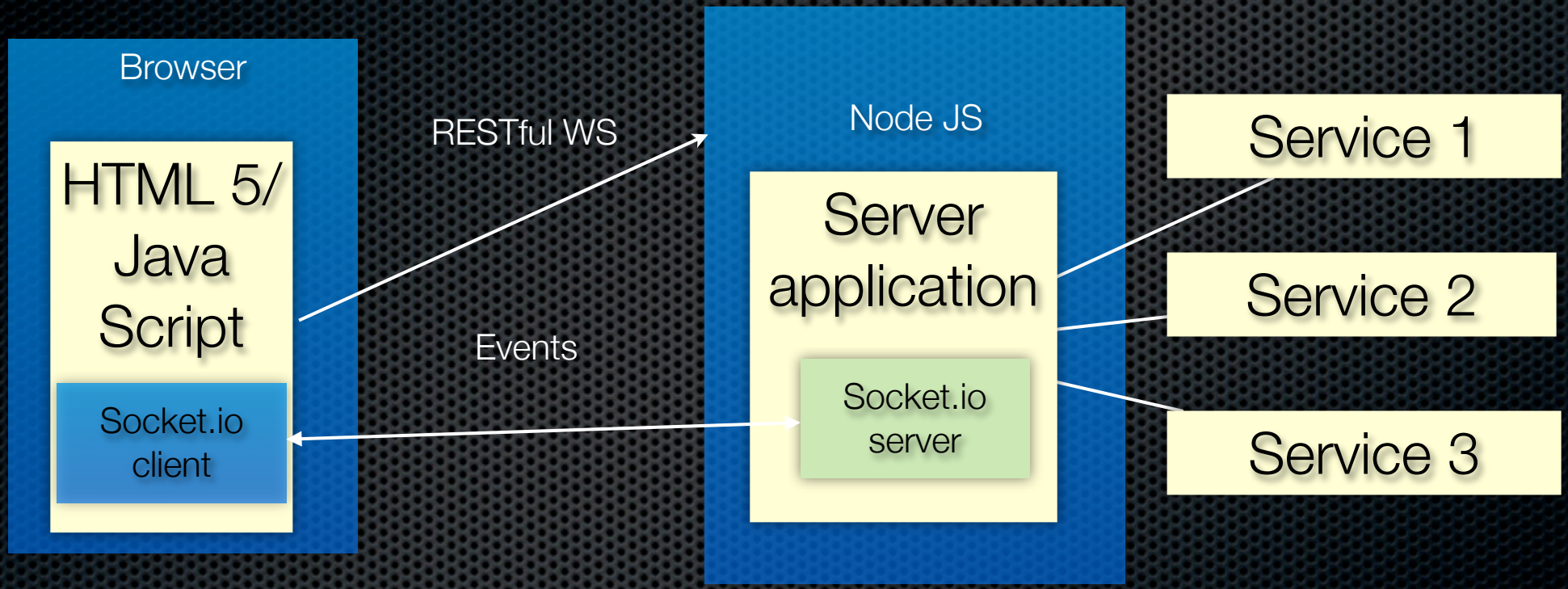
Why not NodeJS?



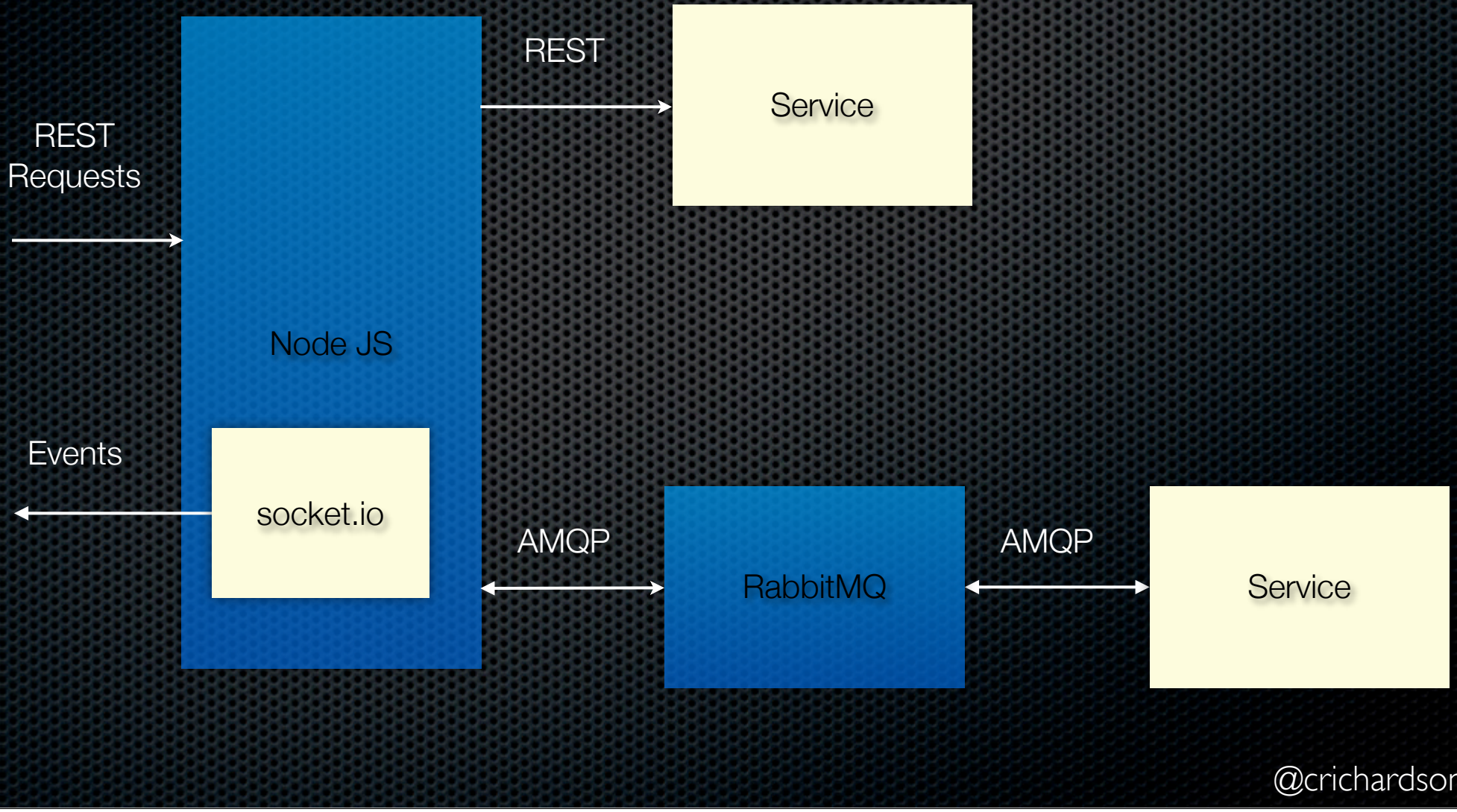


a.k.a. callback hell

A modern web application



NodeJS - using RESTful WS and AMQP



Socket.io server-side

```
var express = require('express')
   , http = require('http')
   , amqp = require('amqp')
   ....;

server.listen(8081);
...
var amqpCon = amqp.createConnection(...);

io.sockets.on('connection', function (socket) {

  function amqpMessageHandler(message, headers, deliveryInfo) {
     var m = JSON.parse(message.data.toString());
     socket.emit('tick', m);
   };
   amqpCon.queue("", {},
      function(queue) {
         queue.bind("myExchange", "");
         queue.subscribe(amqpMessageHandler);
   });
});
```

Handle socket.io connection

Republish as socket.io event

Subscribe to AMQP queue

https://github.com/cer/nodejs-clock

Socket.io - client side

```
<html>
<body>

The event is <span data-bind="text: ticker"></span>

<script src="/socket.io/socket.io.js"></script>
<script src="/knockout-2.0.0.js"></script>
<script src="/clock.js"></script>

</body>
</html>
```

Bind to model

Connect to socket.io

clock.js

```
var socket = io.connect(location.hostname);

function ClockModel() {
  self.ticker = ko.observable(1);
  socket.on('tick', function (data) {
   self.ticker(data);
  });
};

ko.applyBindings(new ClockModel());
```

Subscribe to tick event

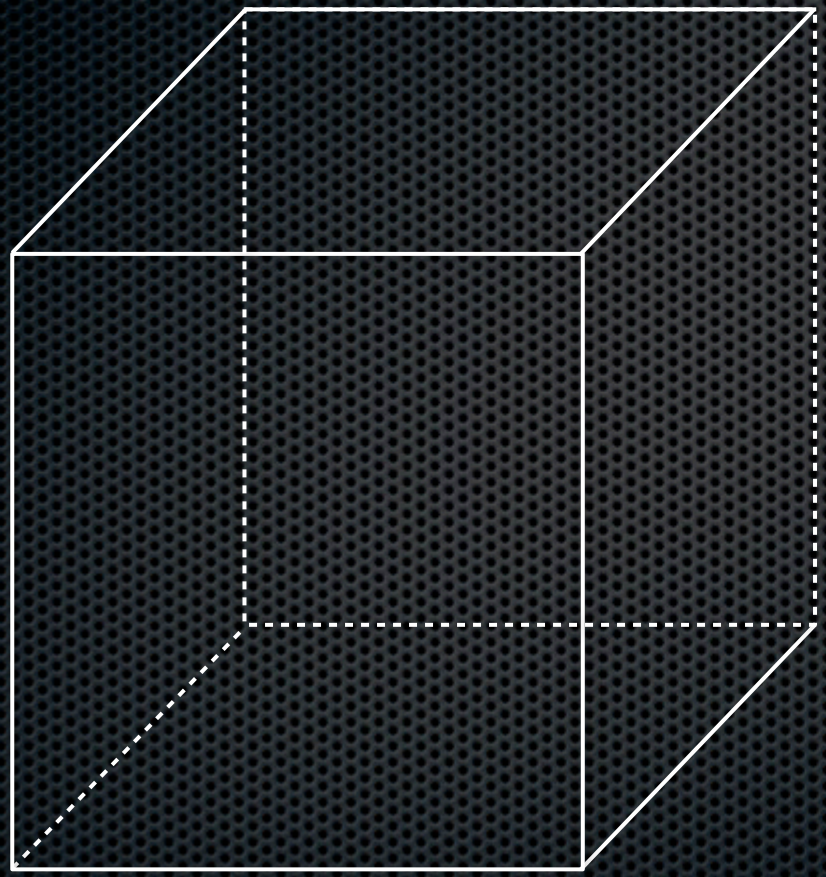
Update model

Summary

Monolithic applications are simple to develop and deploy

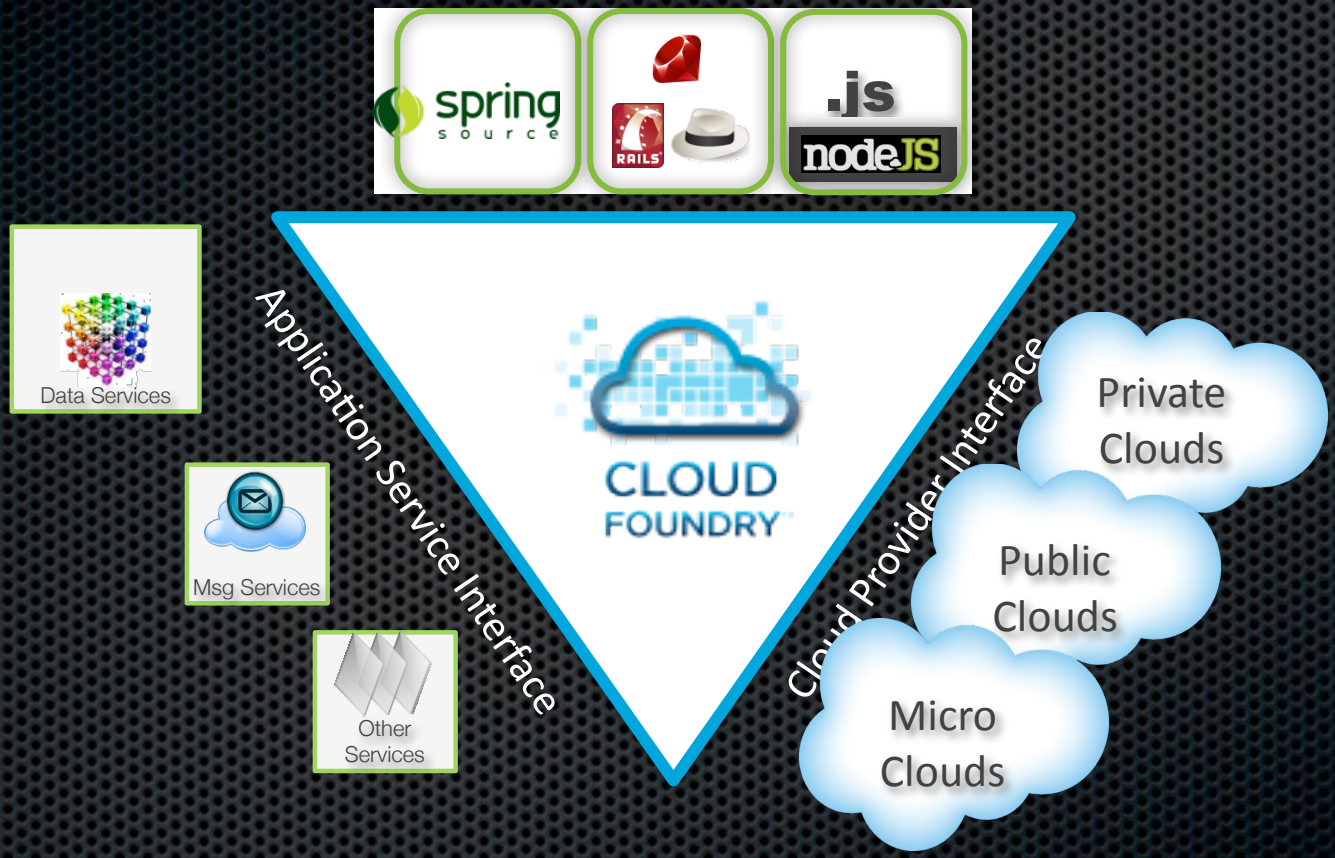
BUT have significant drawbacks

Apply the scale cube



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

Cloud Foundry helps



@crichardson chris.richardson@springsource.com

http://plainoldobjects.com - code and slides

Questions?

www.cloudfoundry.com