

# **Programming the Cloud Dealing with Distributed Storage**

**John Kilroy  
Fleur Dragan  
EMC**

- ❑ This talk will describe how cloud-based applications interact with stored data. The traditional semantics of file systems are often not applicable or relevant. We will discuss how language-specific idioms from several common development frameworks are mapped into the stored data abstractions present in the cloud. We will also outline some of the practical implications of deploying such applications.
- ❑ Learning Objectives
  - ❑ Understand how cloud applications interact with stored data
  - ❑ Semantic differences between traditional file systems and cloud stores (blob, NoSQL and key-value)
  - ❑ Practical implications of variations in consistency models, latencies, and geographic distribution
  - ❑ Comparison of language-specific idioms for interacting with cloud stores (Ruby vs Java/Spring vs Python)
  - ❑ Lessons learned from a real-world implementation

- ❑ Terminology
- ❑ How Internet-scale application requirements are displacing traditional systems
- ❑ Cloud storage versus traditional storage
- ❑ Strategies and challenges for integrating cloud storage
- ❑ Key takeaways
  
- ❑ Q&A

# Terminology

- The Cloud...
- Blob/Cloud Storage



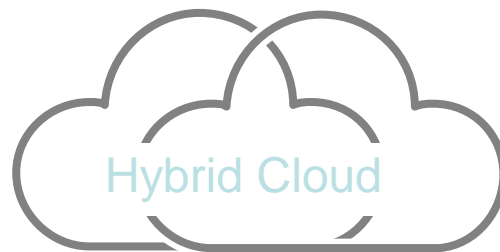
Courtesy – wordle.net

# Supporting the Shift to Cloud Inside, Outside, and Across Organizations

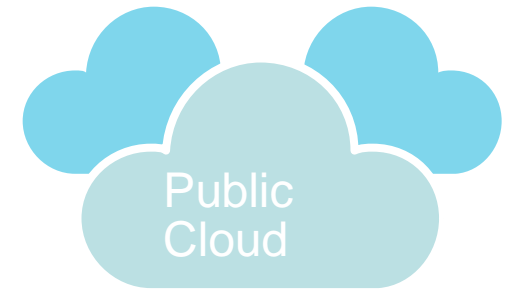
Cloud is a model for enabling **convenient, on-demand** network access to a **shared pool** of configurable computing resources (e.g. networks, servers, storage, applications) that can be **rapidly provisioned and released** with **minimal management effort** or service provider interaction



Infrastructure deployed and operated exclusively for an organization or enterprise



Composition of two or more clouds, private and/or public



Infrastructure made available to general public or many industry groups/customers

Source: \*National Institute of Standards and Technology, V15 October 2009

- ❑ How Internet-scale application requirements are displacing traditional systems
  - ❑ 10s of thousands of nodes, PBs of data
  - ❑ 100s of thousands of disks
  
- ❑ The CAP Theorem, application requirements, and the rise of no-sql db's and blob (cloud) stores

- ❑ Berkeley professor Eric Brewer's CAP theorem states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees
  - ❑ Consistency
    - ❑ All nodes see the same data at the same time
  - ❑ Availability
    - ❑ Node failures do not prevent survivors from continuing to operate
  - ❑ Partition tolerance
    - ❑ The system continues to operate despite arbitrary message loss due to network failure
- ❑ A system can satisfy only TWO of these guarantees at the same time...

- ❑ No-SQL
- ❑ Blob stores
  
- ❑ Evolution
  - ❑ ISAM
  - ❑ RDBMS
  - ❑ Clustered/  
Sharding
  - ❑ NoSQL

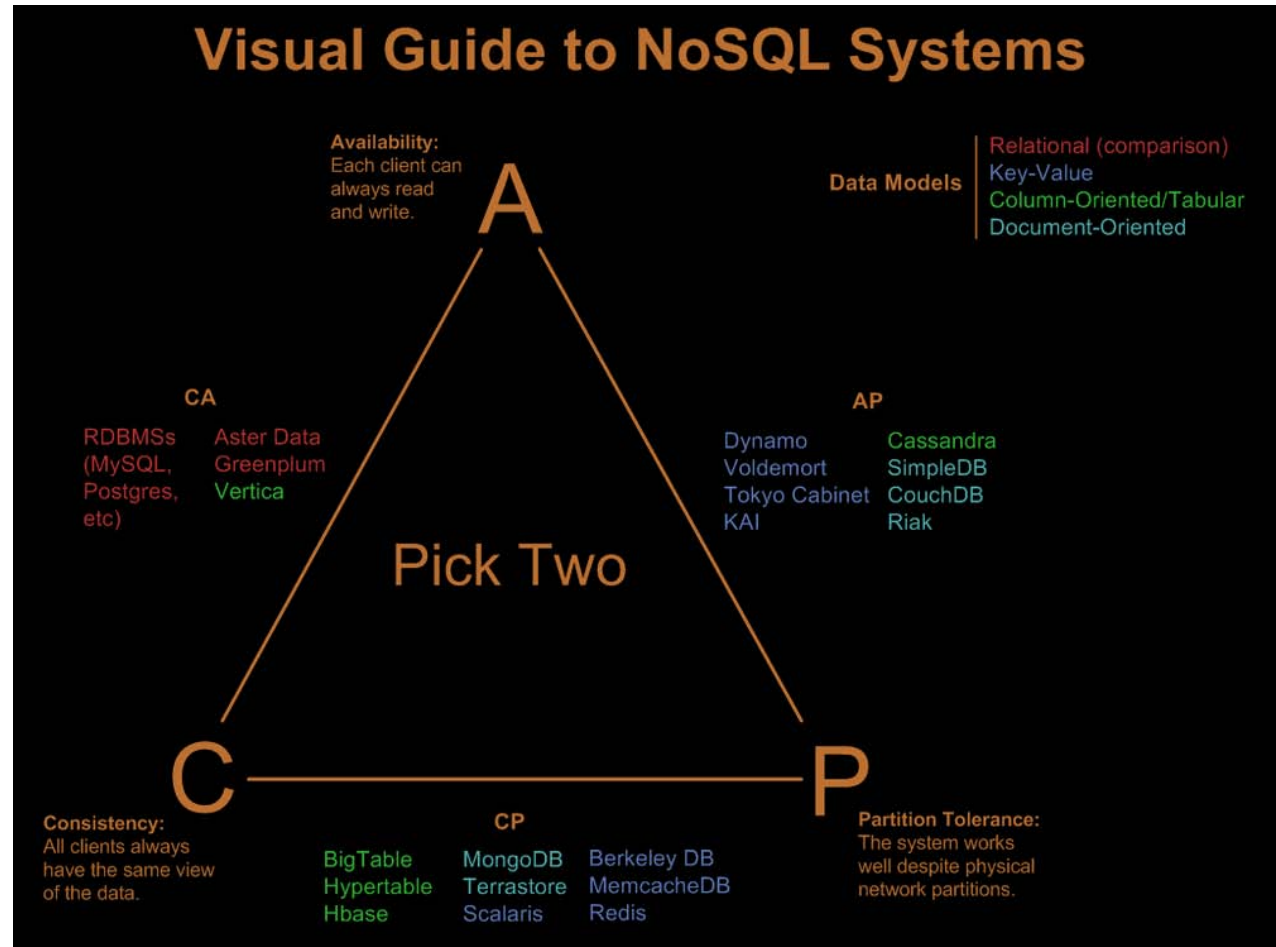


Diagram source: Nathan Hurst's blog: [blog.nahurst.com/visual-guide-to-nosql-systems](http://blog.nahurst.com/visual-guide-to-nosql-systems)

- ❑ Of the CAP theorem...
- ❑ The requirements of Internet scale applications exceed the capabilities of traditional structured and unstructured storage systems.
- ❑ At low transactional volumes, small latencies to allow databases to get consistent have no noticeable affect on either overall performance or user experience.
- ❑ At high x-action volume => increased latency to keep consistent
- ❑ Deploying replicas for fault-tolerance/load balancing => increased latency

# Evolving Apps to Scale

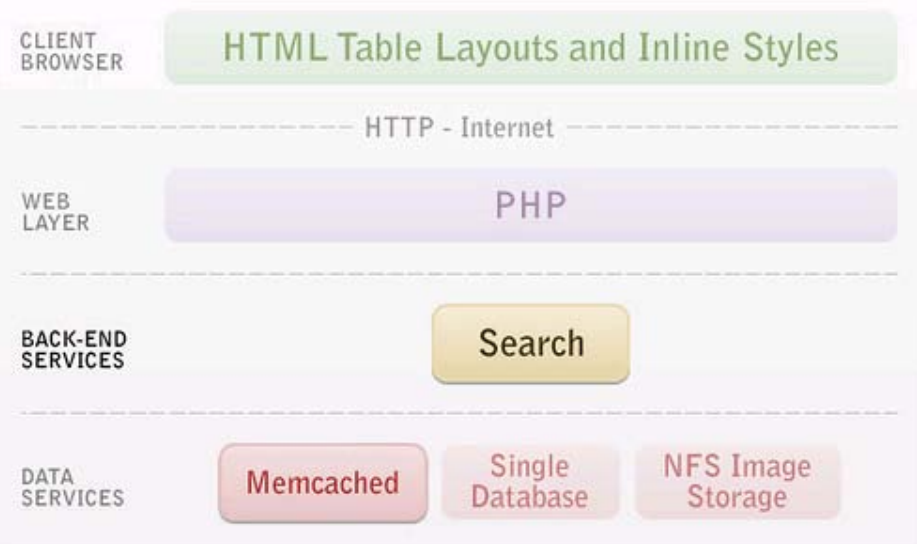
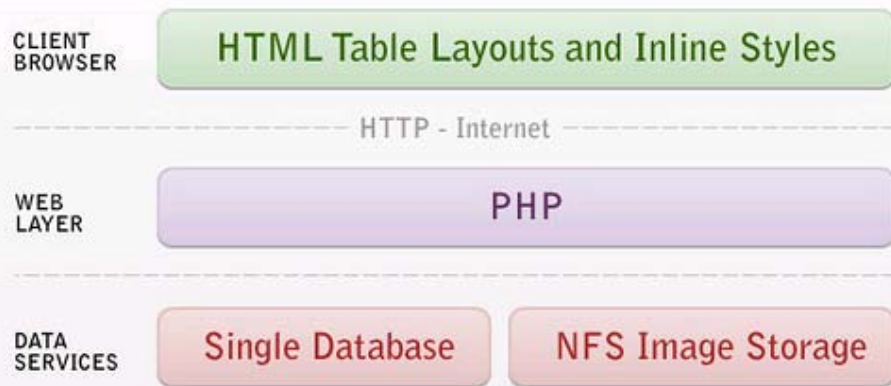
# An example...

## ❑ V1 (2004)

- ❑ PHP webapp
- ❑ 100k users, 15 servers

## ❑ V2 (2005)

- ❑ Cached PHP webapp
- ❑ 1m users, 20 servers

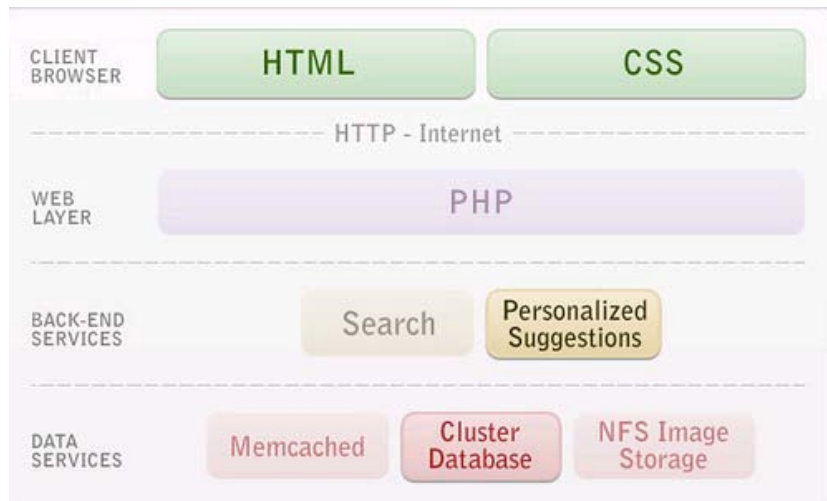


How the architecture at Tagged evolved to meet scalability requirements

# An example...

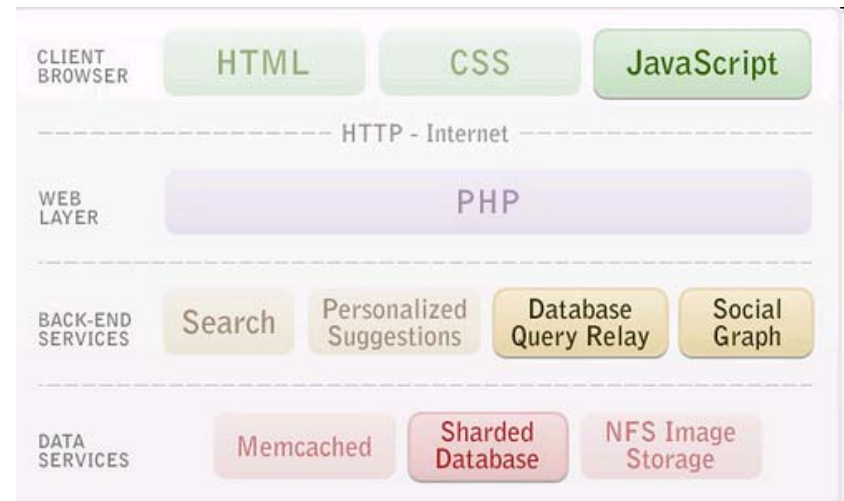
## □ V3 (2006)

- Databases scaling
- 10m users, 100 servers



## □ V4 (2007)

- Database sharding
- 50m users, 500 servers

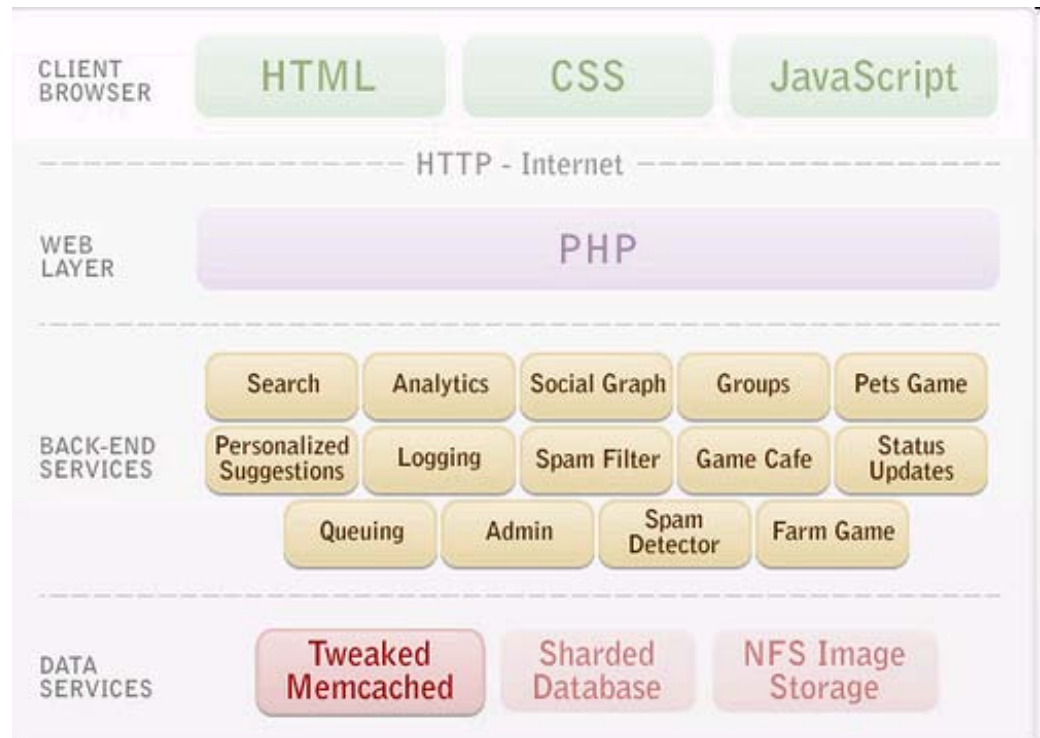


How the architecture at Tagged evolved to meet scalability requirements

# An example...

- V5 (2010)
  - Refinements and extensions
  - 80m users, 1,000 servers

How the architecture at Tagged evolved to meet scalability requirements



**Will the NFS/NAS store easily scale?**

# Why do we need cloud storage?

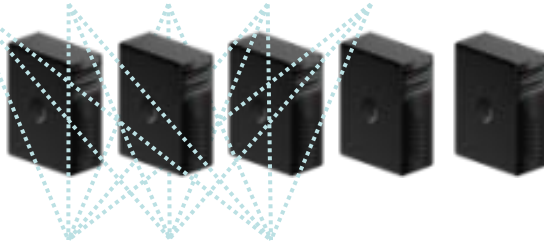
## Scalability issues w/ NAS and mount points

### Application

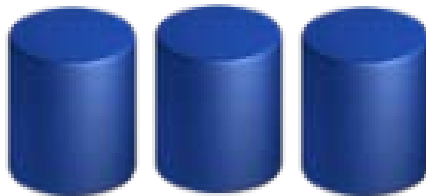


### Application servers

- Nas1:/mnt1
- Nas2:/mnt2
- Nasn:/mntn



### Storage



Single site

*Active*

- ❑ Single points of failure that you have to manage
  - ❑ App has 20-30 servers
  - ❑ 5 mount points \* 30 servers = 150 single points of failure
- ❑ Location and server problems
  - ❑ //corpusmx32/somedir/ ???
- ❑ Unnecessarily complex semantics
  - ❑ Create/read/write/close/remove
- ❑ Need to know path and maintain state for file handles
  - ❑ State doesn't scale

# Why do we need cloud storage?

## NAS challenges multiply with two sites

### Application



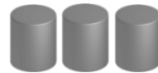
### Application servers

- Nas1:/mnt1
- Nas2:/mnt2
- Nasn:/mntn



**Site #1**  
*Active*

rsync scripts



**Site #2**  
*Passive*

### Storage

- ❑ Even more single points of failure
- ❑ Homegrown replication software is a lot of overhead
  - ❑ Scripted rsync anyone?
  - ❑ FTP maybe?
  - ❑ Managing consistency is a challenge
- ❑ Passive failover resources are sitting around idle
- ❑ Layering active/passive instances in alternate directions adds further complexity
- ❑ Two sites is bad, three+ sites is worse

# Why do we need cloud storage?

## NAS challenges to overcome

- ❑ Eliminate single points of failure
- ❑ Simplify semantics – create/update/delete
- ❑ Eliminate file handles and mount points
- ❑ Eliminate dependencies on path, state, location
- ❑ Get rid of sizing / provisioning constraints
- ❑ Maximize utilization of resources

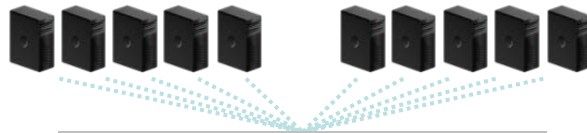
# Cloud Storage

## Application

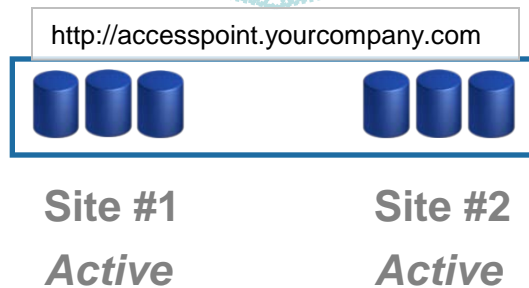


## Application servers

•NasA:/mntA



## Storage



- ❑ Single URL across all access nodes – no SPOF, failover
- ❑ Simple – create/update/delete
- ❑ No need to understand handles, paths, state or location
- ❑ Once access is permitted, all capacity is available
- ❑ All sites active for reads and writes – no wasted resources
- ❑ PLUS
  - ❑ Policy-based management for content protection, retention and positioning

# Integrating cloud storage: Strategies and development challenges

- ❑ Is there a standard/consistent interface shared by cloud providers?
  - ❑ No... (close, but no....)
- ❑ What drives the differences across system layers?
  - ❑ Data layer – system-level implementation choices
  - ❑ Service layer – what boundary surface is exposed
  - ❑ Application layer – fit language and app paradigms
- ❑ Is it advisable to wrap vendor API's
  - ❑ Provide an isolation layer, portability
  - ❑ Adapt the API to be more friendly to language paradigms
- ❑ Two example language bindings and frameworks available
  - ❑ Spring and Ruby

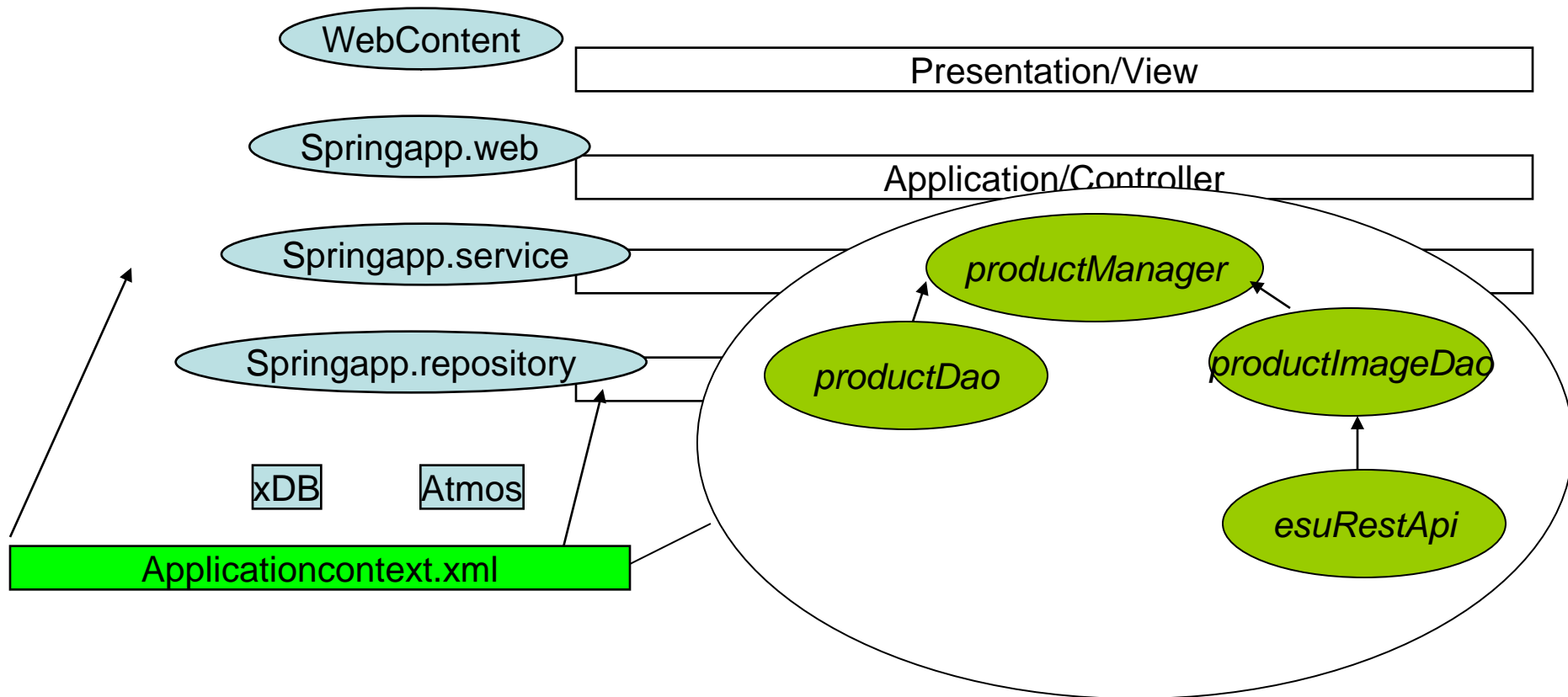
# Example – Spring

- VI “Developing a Spring Framework MVC Application Step-by-step”
  - <http://static.springsource.org/docs/Spring-MVC-step-by-step/index.html>

- V1 “Developing a Spring Framework MVC Application Step-by-step”
  - <http://static.springsource.org/docs/Spring-MVC-step-by-step/index.html>
- V2 “Spring into Atmos” Parts 1 and 2
  - <https://community.emc.com/docs/DOC-7655>
  - <https://community.emc.com/docs/DOC-8197>
  - Xdb & Atmos

- V1 “Developing a Spring Framework MVC Application Step-by-step”
  - <http://static.springsource.org/docs/Spring-MVC-step-by-step/index.html>
- V2 “Spring into Atmos” Parts 1 and 2
  - <https://community.emc.com/docs/DOC-7655>
  - <https://community.emc.com/docs/DOC-8197>
  - Xdb & Atmos
- V3 Today’s app...
  - Re-factored structure so that Spring directly instantiates and injects the Atmos API
  - CSS v3 styling

# Spring Artifacts



# Atmos Java API – EsuAPI

<a href="#">ObjectId</a>	<a href="#">createObject</a> ( <a href="#">Acl</a> acl, <a href="#">MetadataList</a> metadata, byte[] data, java.lang.String mimeType) Creates a new object in the cloud.
<a href="#">ObjectId</a>	<a href="#">createObjectFromSegment</a> ( <a href="#">Acl</a> acl, <a href="#">MetadataList</a> metadata, <a href="#">BufferSegment</a> data, java.lang.String mimeType) Creates a new object in the cloud using a BufferSegment.
<a href="#">ObjectId</a>	<a href="#">createObjectFromSegmentOnPath</a> ( <a href="#">ObjectPath</a> path, <a href="#">Acl</a> acl, <a href="#">MetadataList</a> metadata, <a href="#">BufferSegment</a> data, java.lang.String mimeType) Creates a new object in the cloud using a BufferSegment on the given path.
<a href="#">ObjectId</a>	<a href="#">createObjectOnPath</a> ( <a href="#">ObjectPath</a> path, <a href="#">Acl</a> acl, <a href="#">MetadataList</a> metadata, byte[] data, java.lang.String mimeType) Creates a new object in the cloud on the specified path.
void	<a href="#">deleteObject</a> ( <a href="#">Identifier</a> id) Deletes an object from the cloud.
void	<a href="#">deleteUserMetadata</a> ( <a href="#">Identifier</a> id, <a href="#">MetadataTags</a> tags) Deletes metadata items from an object.
<a href="#">Acl</a>	<a href="#">getAcl</a> ( <a href="#">Identifier</a> id) Returns an object's ACL
<a href="#">ObjectMetadata</a>	<a href="#">getAllMetadata</a> ( <a href="#">Identifier</a> id) Returns all of an object's metadata and its ACL in one call.
<a href="#">MetadataTags</a>	<a href="#">getListableTags</a> ( <a href="#">MetadataTag</a> tag) Returns a list of the tags that are listable the current user's tennant.
<a href="#">MetadataTags</a>	<a href="#">getListableTags</a> (java.lang.String tag) Returns a list of the tags that are listable the current user's tennant.
<a href="#">MetadataList</a>	<a href="#">getSystemMetadata</a> ( <a href="#">Identifier</a> id, <a href="#">MetadataTags</a> tags) Fetches the system metadata for the object.
<a href="#">MetadataList</a>	<a href="#">getUserMetadata</a> ( <a href="#">Identifier</a> id, <a href="#">MetadataTags</a> tags) Fetches the user metadata for the object.

# Example – Ruby

## □ Create/Read/Update/Delete in EsuApi

```
require 'rubygems'
require 'EsuApi'

esu = EsuApi::EsuRestApi.new('accesspoint.atmosonline.com', 80,
    'b245ad26f5c94bcbbaffa315a833f27a/A79755852684b2e05d00',
    'D7qsp4j16PBHWSiUbc/bt3lbPBX=' )
id = esu.create_object( nil, nil, nil, nil )

content = esu.read_object( id, nil, nil )
esu.update_object( id, nil, nil, "Hello World", "text/plain" )
esu.delete_object(id)
```

# Ruby aws/s3 Example

## □ Create/Read/Update/Delete in aws/s3

```
require 'rubygems'
require 'aws/s3'
AWS::S3::Base.establish_connection!(
  :access_key_id      => 'abc',
  :secret_access_key => '123'
)
Bucket.create('bucket')
S3Object.store("file.txt", 'Hello World', 'bucket')
S3Object.value('file.txt', 'bucket')
obj = S3Object.find 'file.txt', 'bucket'
obj.data = 'Hello World!'
obj.store
S3Object.delete 'file.txt', 'bucket'
```

## □ Create/Read/Update/Delete in Atmos::Store

```
require 'rubygems'
require 'atmos'

print "about to start\n"
s = Atmos::Store.new(
  :url => 'https://accesspoint.atmosonline.com',
  :uid  => 'b245ad26f5c94bcbbaffa315a833f27a/A79755852684b2e05d00',
  :secret => 'sM6oN/k6e03iW7lNSUzFav9rwck=' )

print "made connection\n"
obj = s.create(:data => "Hello World")
contents = obj.data
obj.update(".")
obj.delete
```

## □ Create/Read/Update-in-place/Delete in Atmos::Store

```
require 'rubygems'
require 'atmos'

print "about to start\n"
s = Atmos::Store.new(
  :url => 'https://accesspoint.atmosonline.com',
  :uid => 'b245ad26f5c94bcbbaffa315a833f27a/A79755852684b2e05d00',
  :secret => 'sM6oN/k6e03iW7lNSUzFav9rwck=' )

print "made connection\n"
obj = s.create(:data => "Hello World", :mimetype => 'text/plain')
contents = obj.data
obj.update(".")
obj.update("Mad ", 1..5)
obj.delete
```

# Key takeaways

- ❑ Distributed storage is mandatory for internet-scale apps
  - ❑ Applications need to scale orders of magnitude bigger than they used to. You cannot fit your persistent data in a single machine or even a cluster of machines anymore.
- ❑ Distributed stores behaves differently than traditional storage
  - ❑ Latencies. Consistency guarantees. Network reliability
- ❑ App developers who are aware of these differences can adapt.
  - ❑ The difference is not transparent. Traditional file system semantics represent an impedance mismatch. Relaxed consistency constraints aid scalability – do you really need that level of consistency?
  - ❑ Expect things to fail.

# Q & A