COMSE6998: Modern Serverless Cloud Applications

Lecture 7: S3 Websites, Queueing/SQS, Content Management Systems (Intro)

Dr. Donald F. Ferguson
Donald.F.Ferguson@gmail.com

Contents

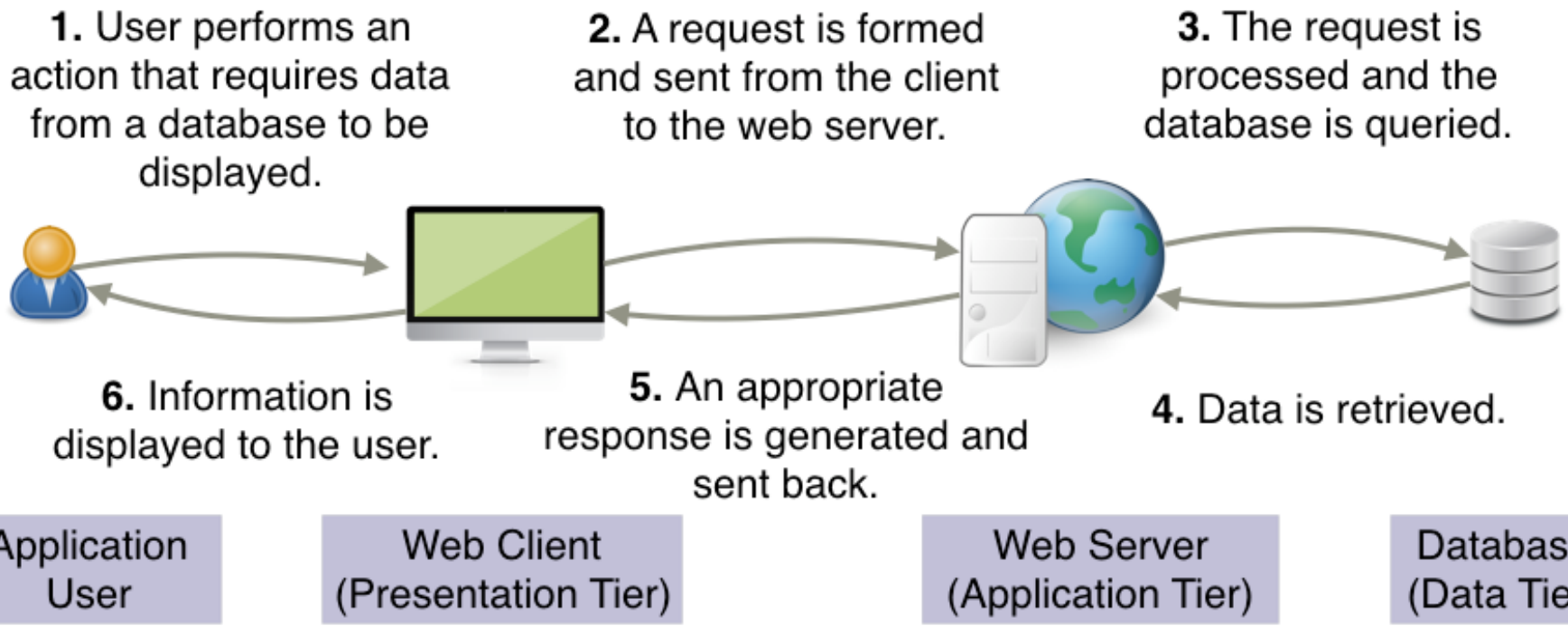
- Introduction
  - Homework status, and some questions.
  - Q&A
- New technical topics:
  - Building a Web Site – Serverless
  - Messaging
    - Asynchronous Operations
    - Message Driven Processing/Queueing
    - Amazon SQS Overview
  - Content/Asset Management (Introduction)
- Putting some pieces together.

Introduction

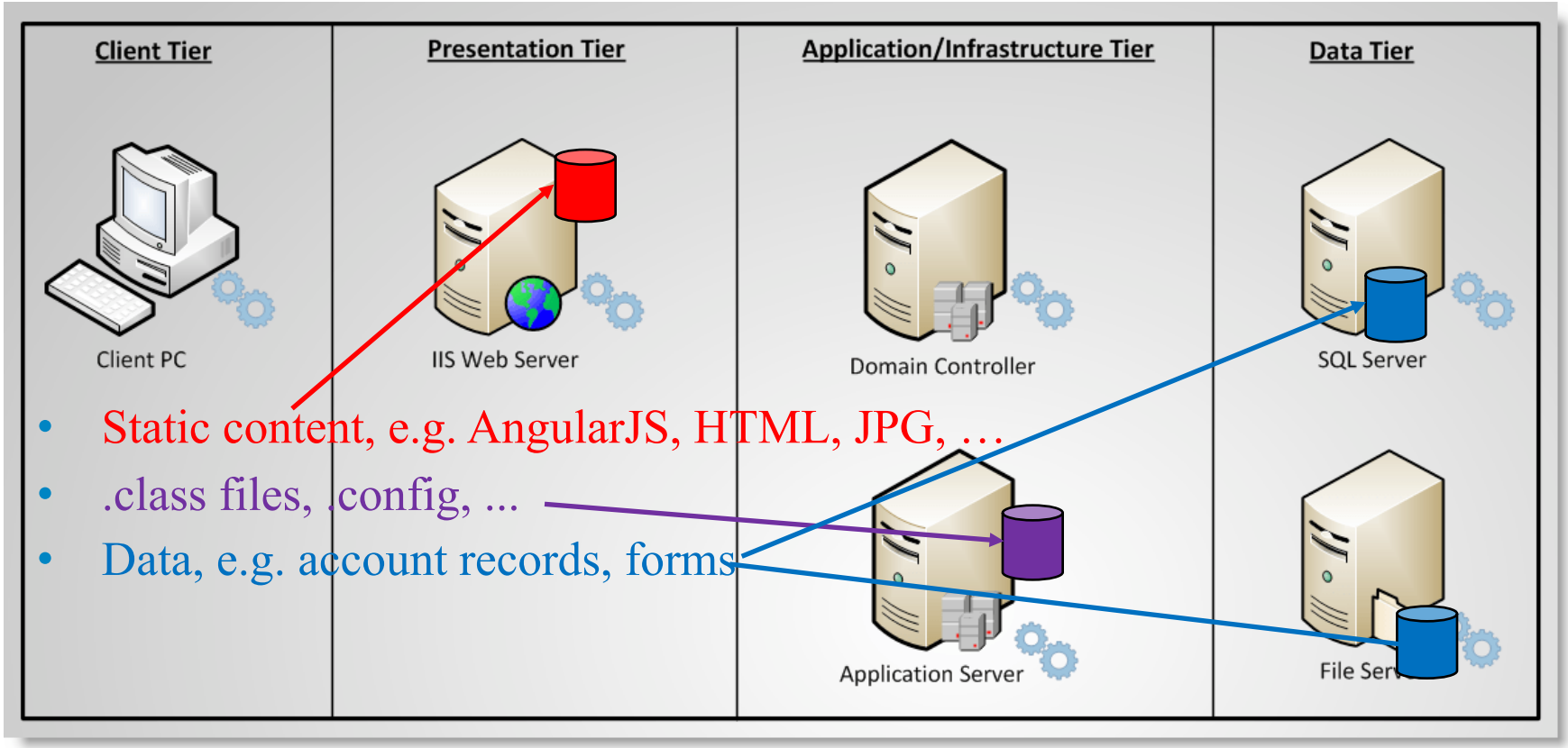
New Technical Topics

Static Website

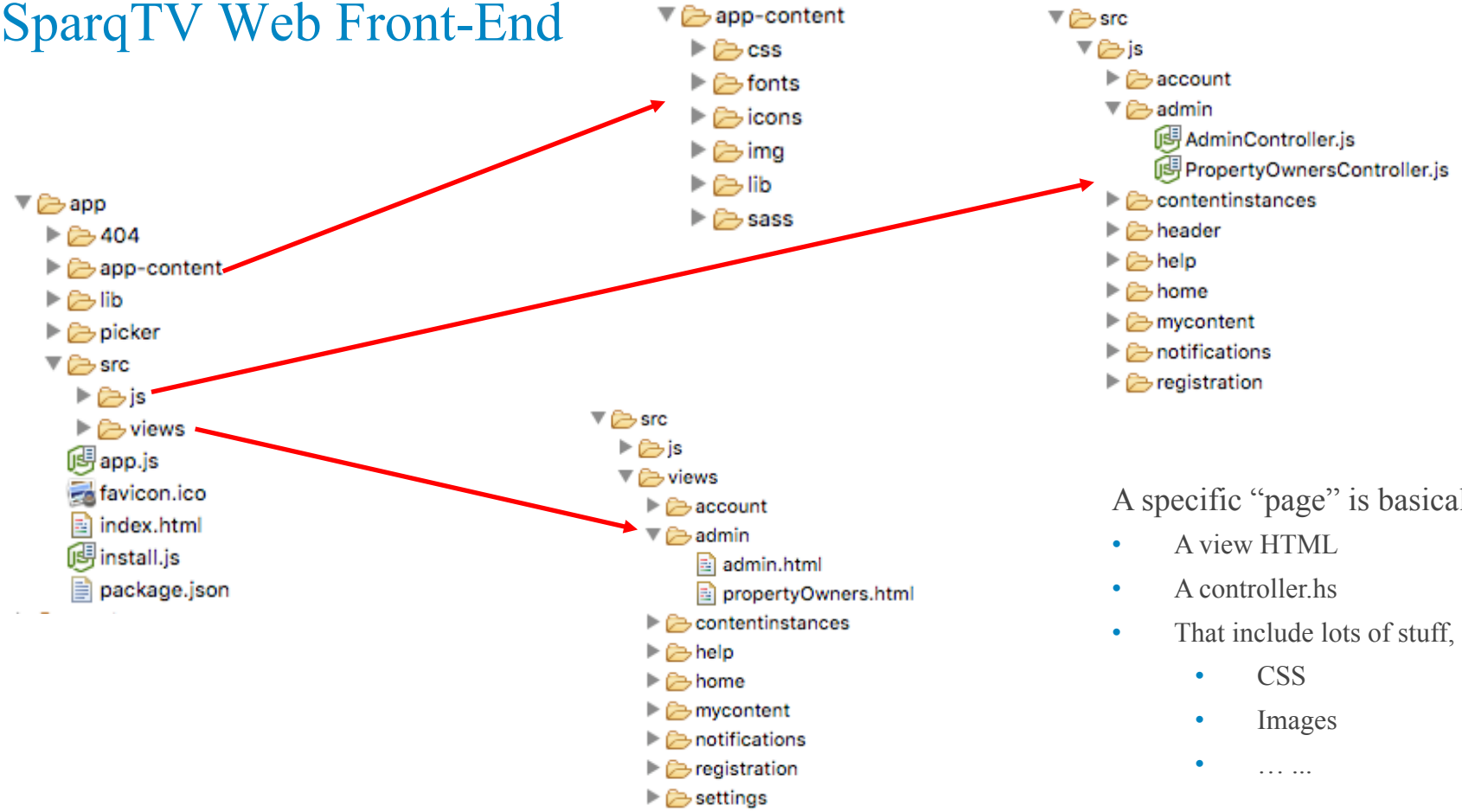
Web Application Basic Concepts



Physical Topology



SparqTV Web Front-End



A specific "page" is basically

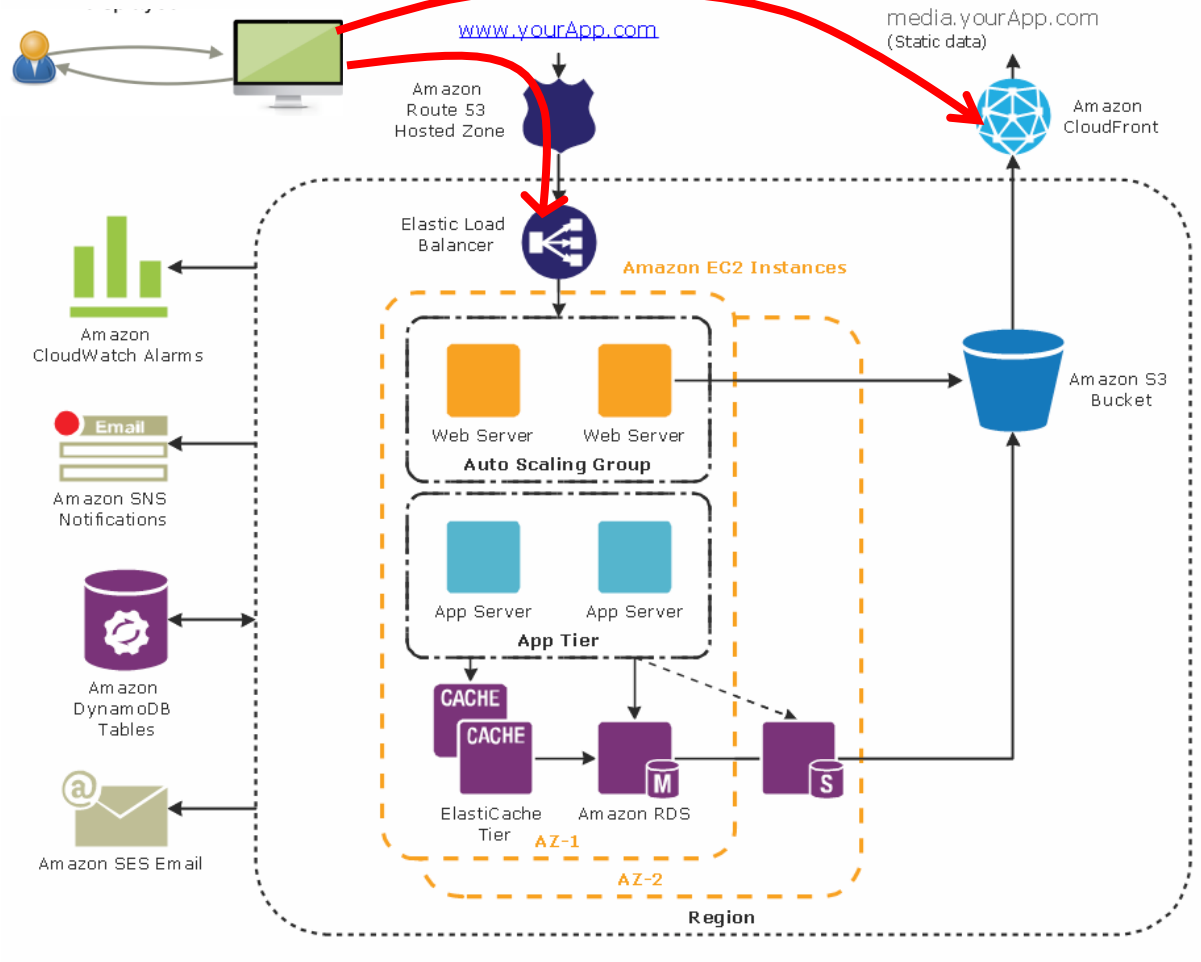
- A view HTML
- A controller.hs
- That include lots of stuff,
  - CSS
  - Images
  - … ...

Amazon S3 Website Steps

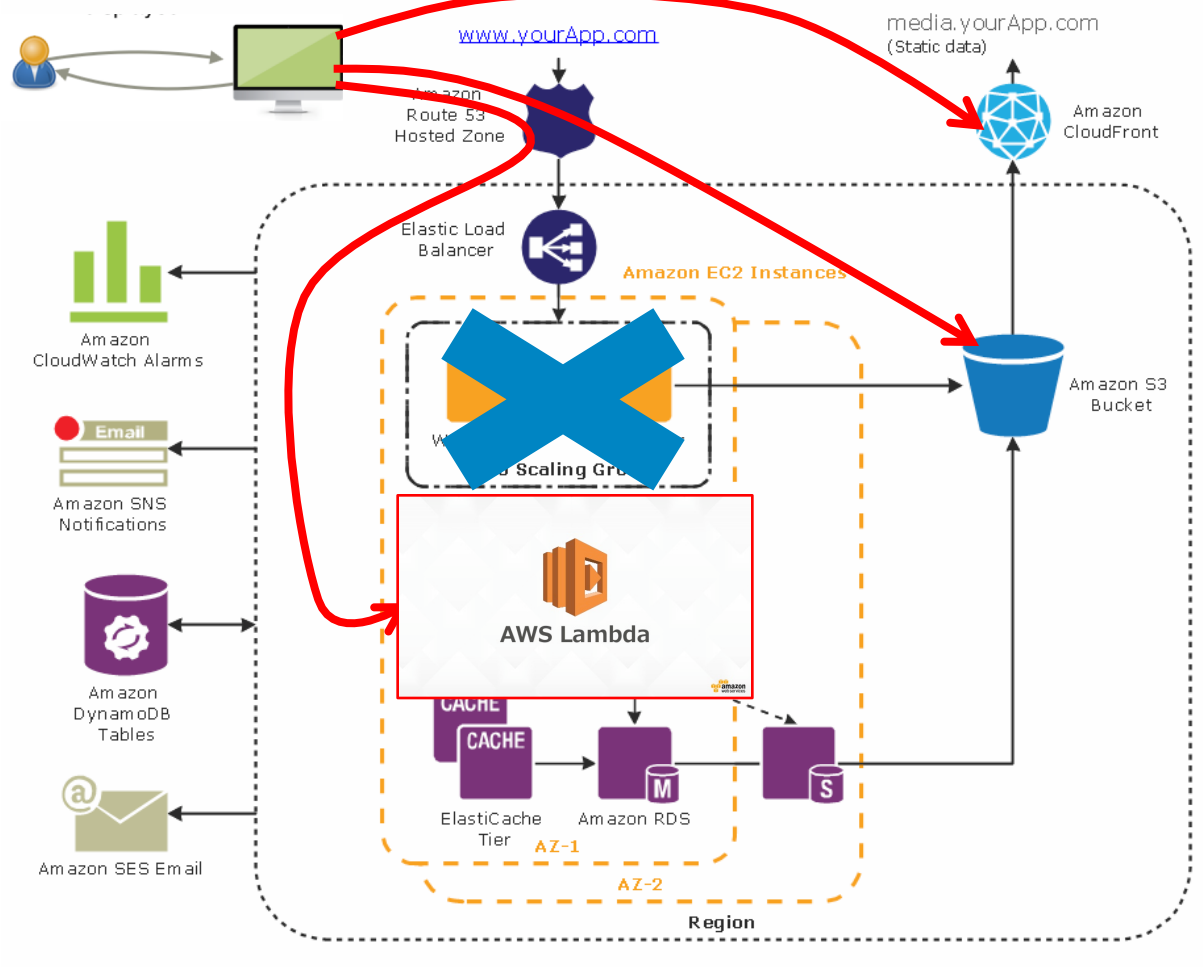
http://docs.aws.amazon.com/gettingstarted/latest/swh/setting-up.html

1. Setting Up: (Account, Permissions, etc).
2. Create the Buckets for Your Website
3. Configure Your Buckets
4. Deploy Your Website
5. ~~(Optional) Register Your Domain Name~~
6. ~~(Optional) Associate a Domain Name with Your Website~~
7. ~~(Optional) Speed Up Your Website~~

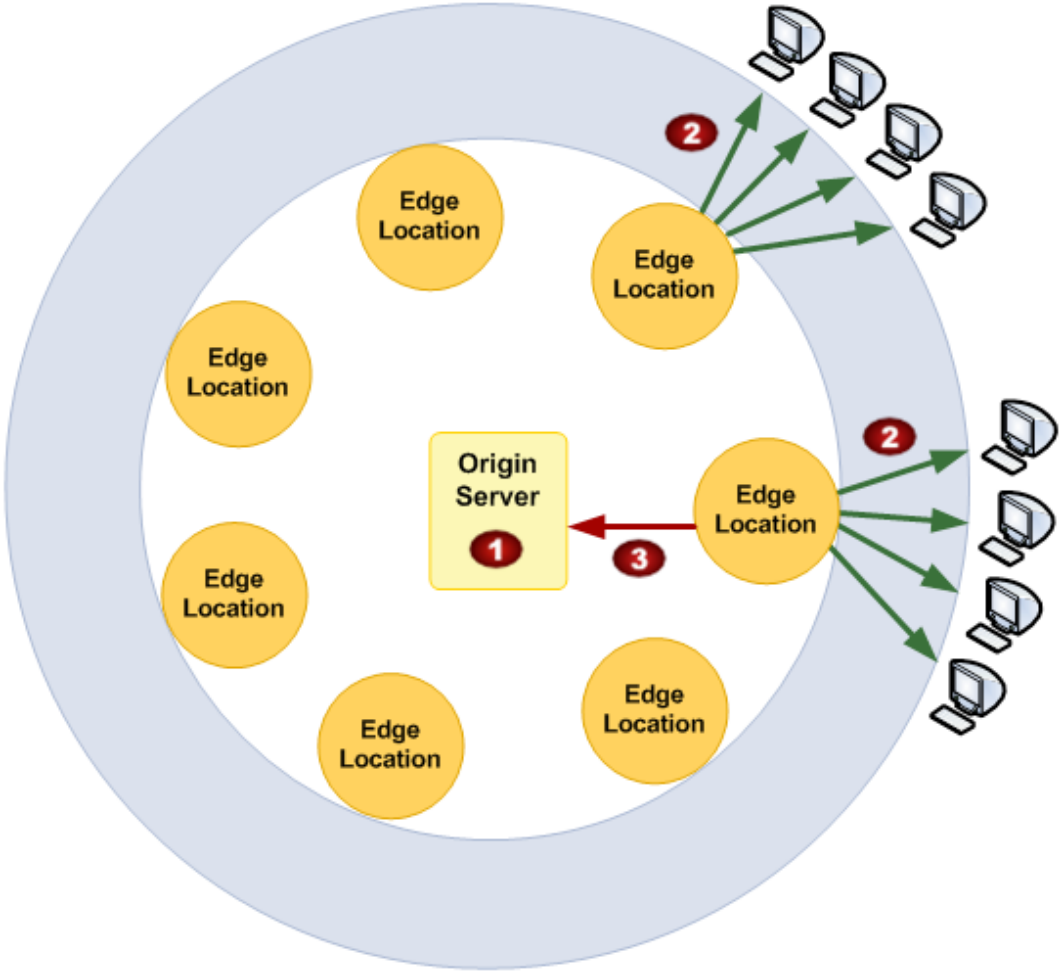
"Traditional" AWS Topology



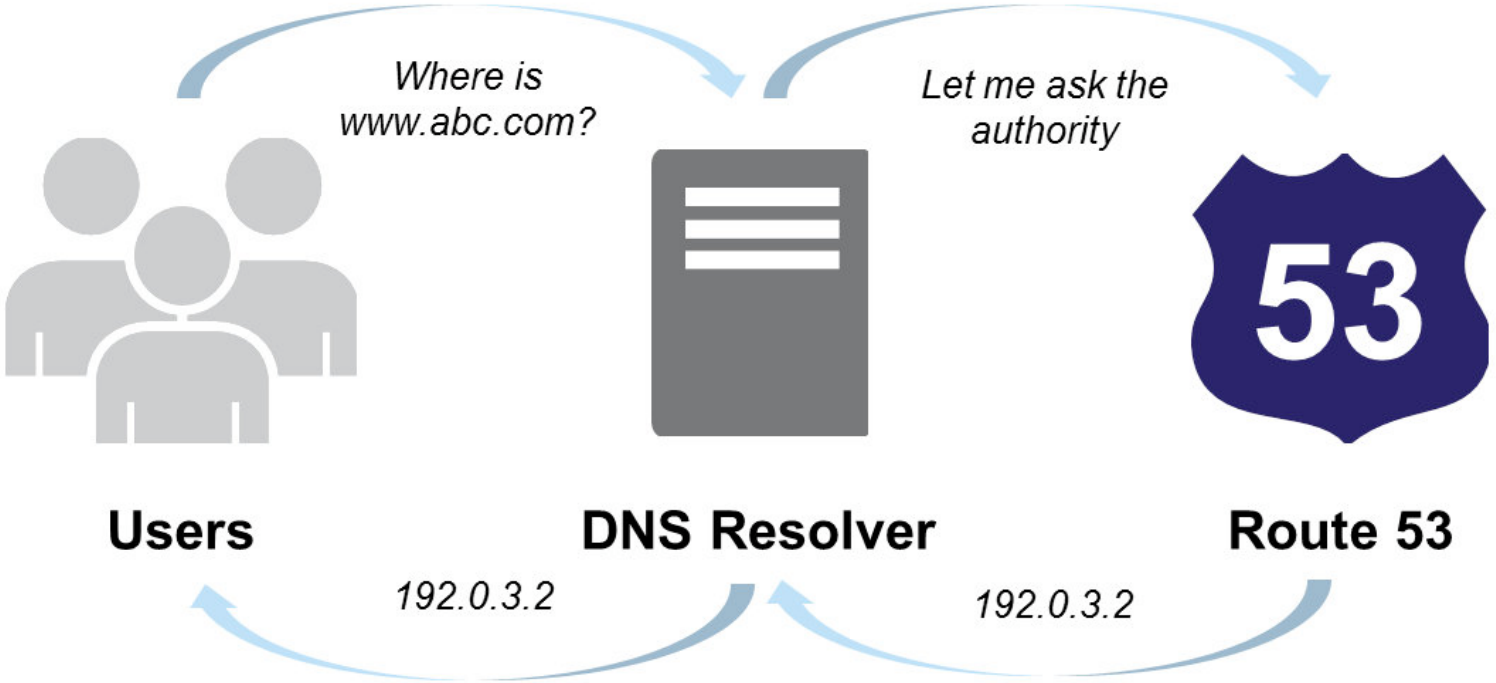
Our Model



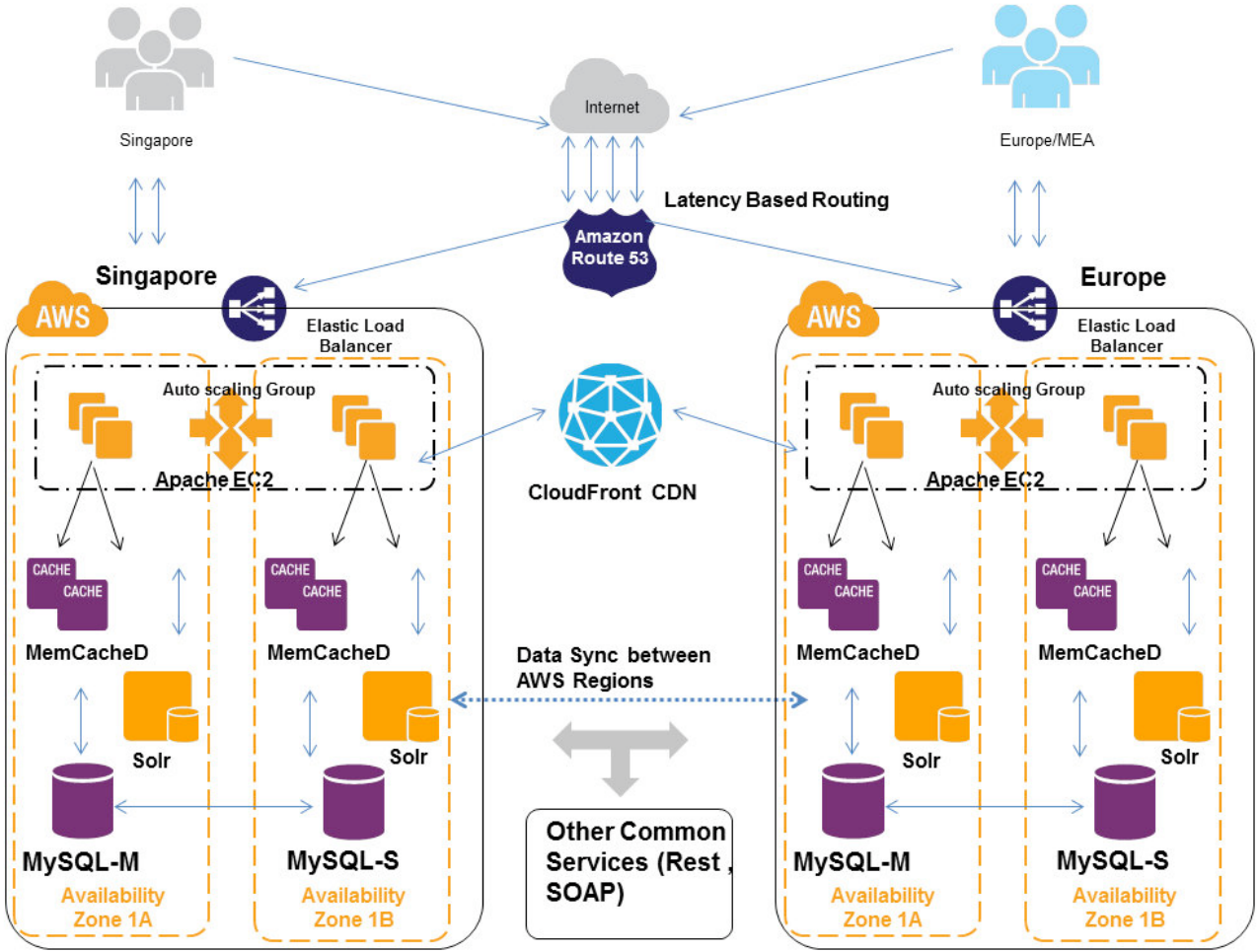
AWS CloudFront



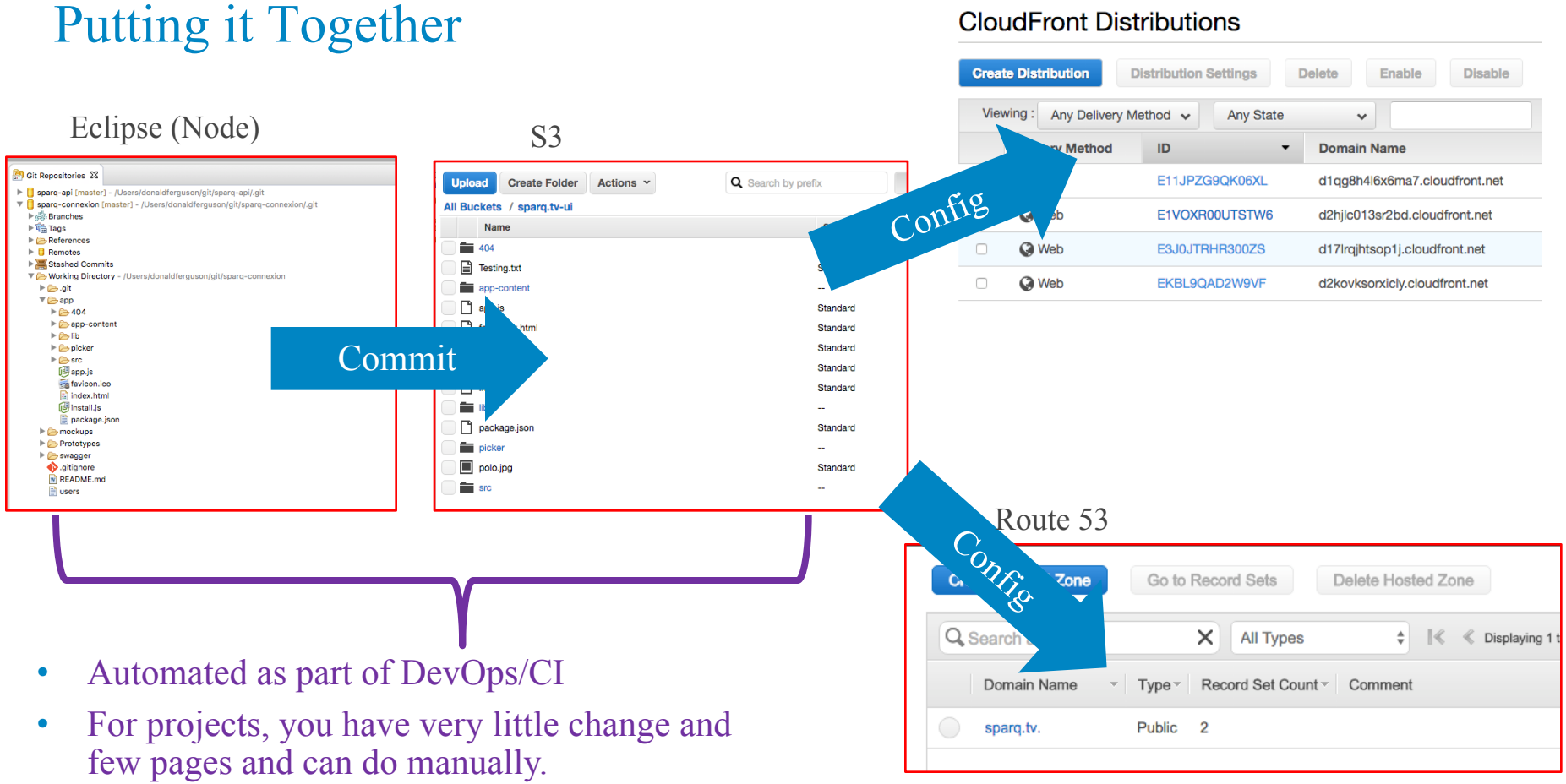
AWS Route 53



Example

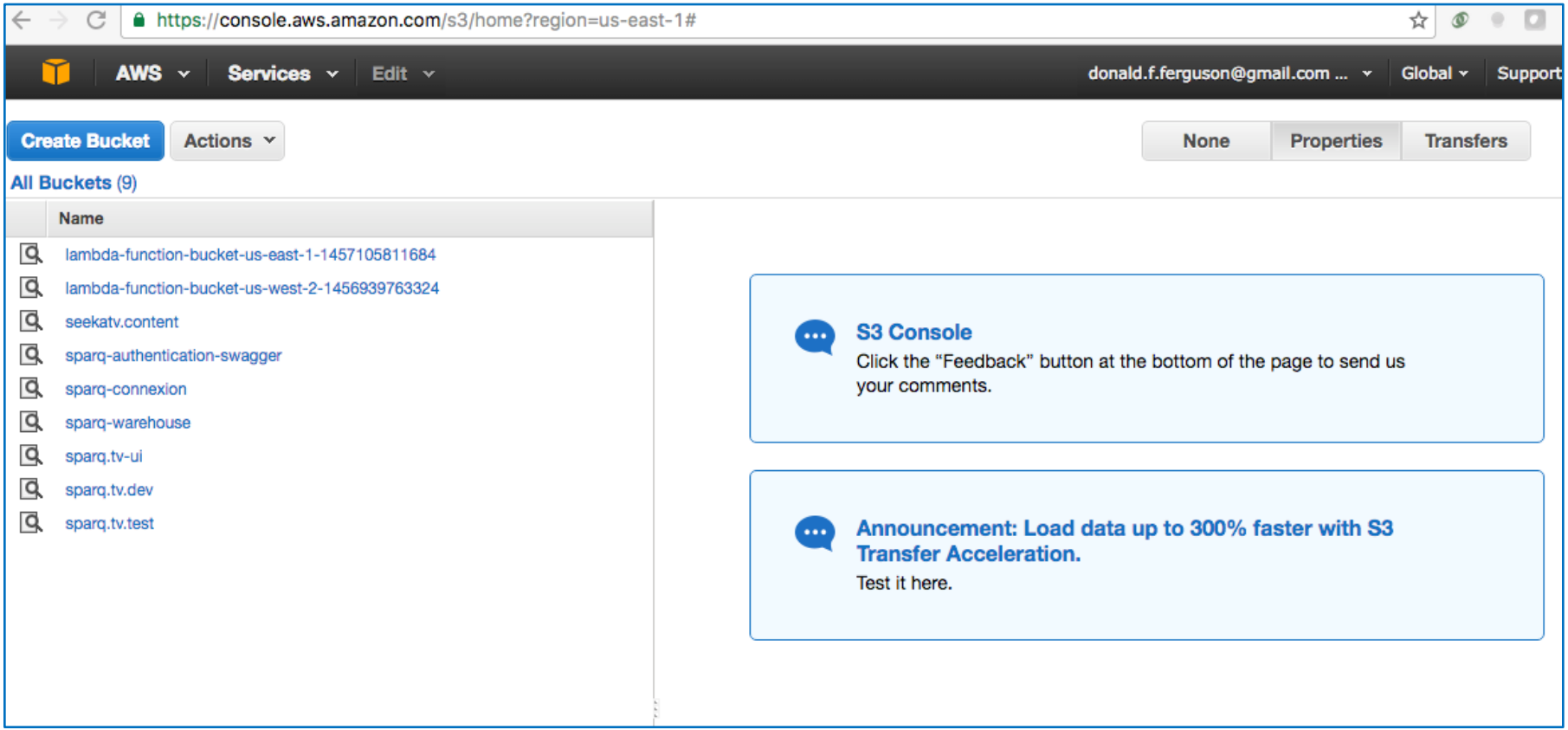


Putting it Together

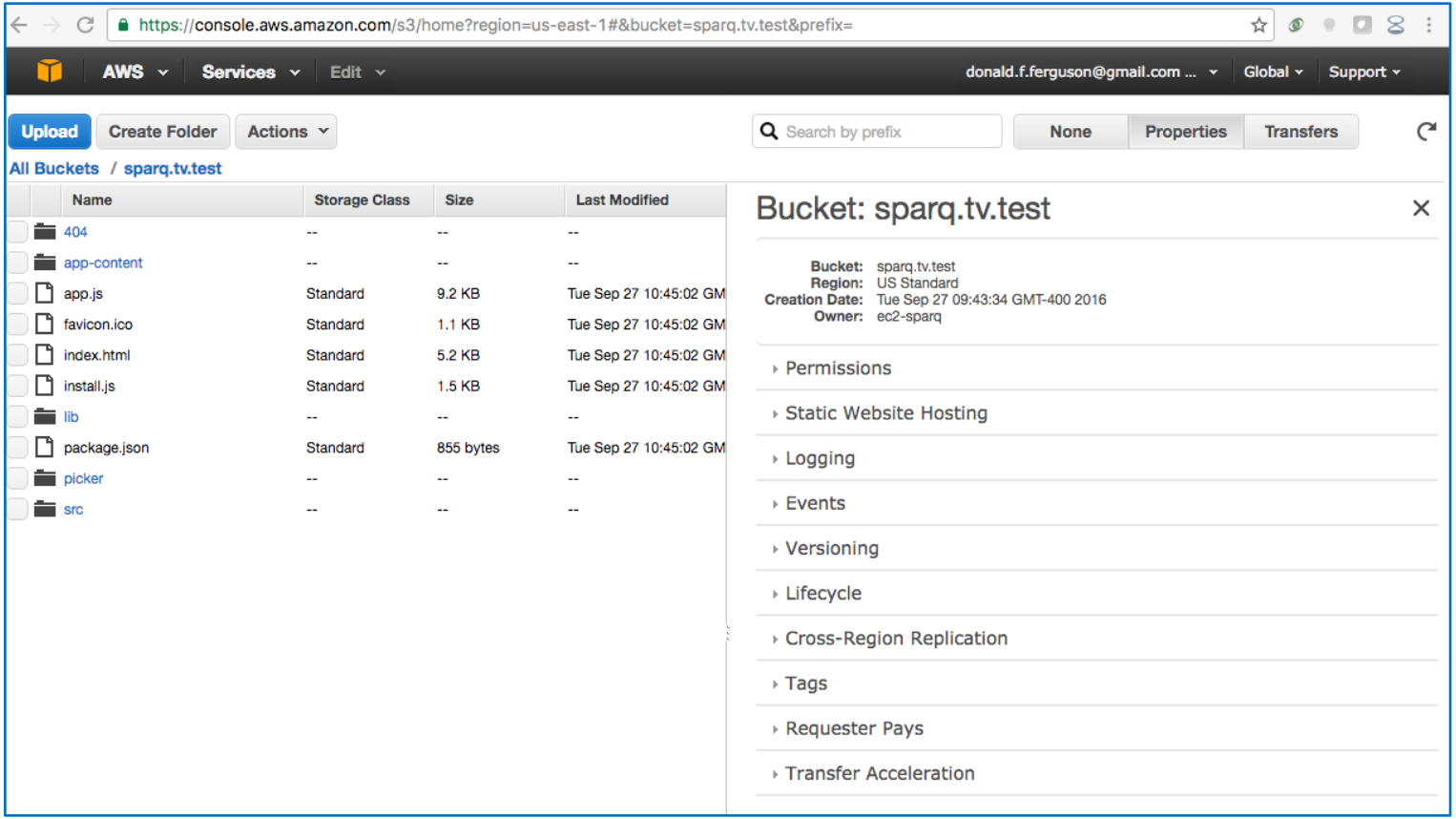


- Automated as part of DevOps/CI
- For projects, you have very little change and few pages and can do manually.

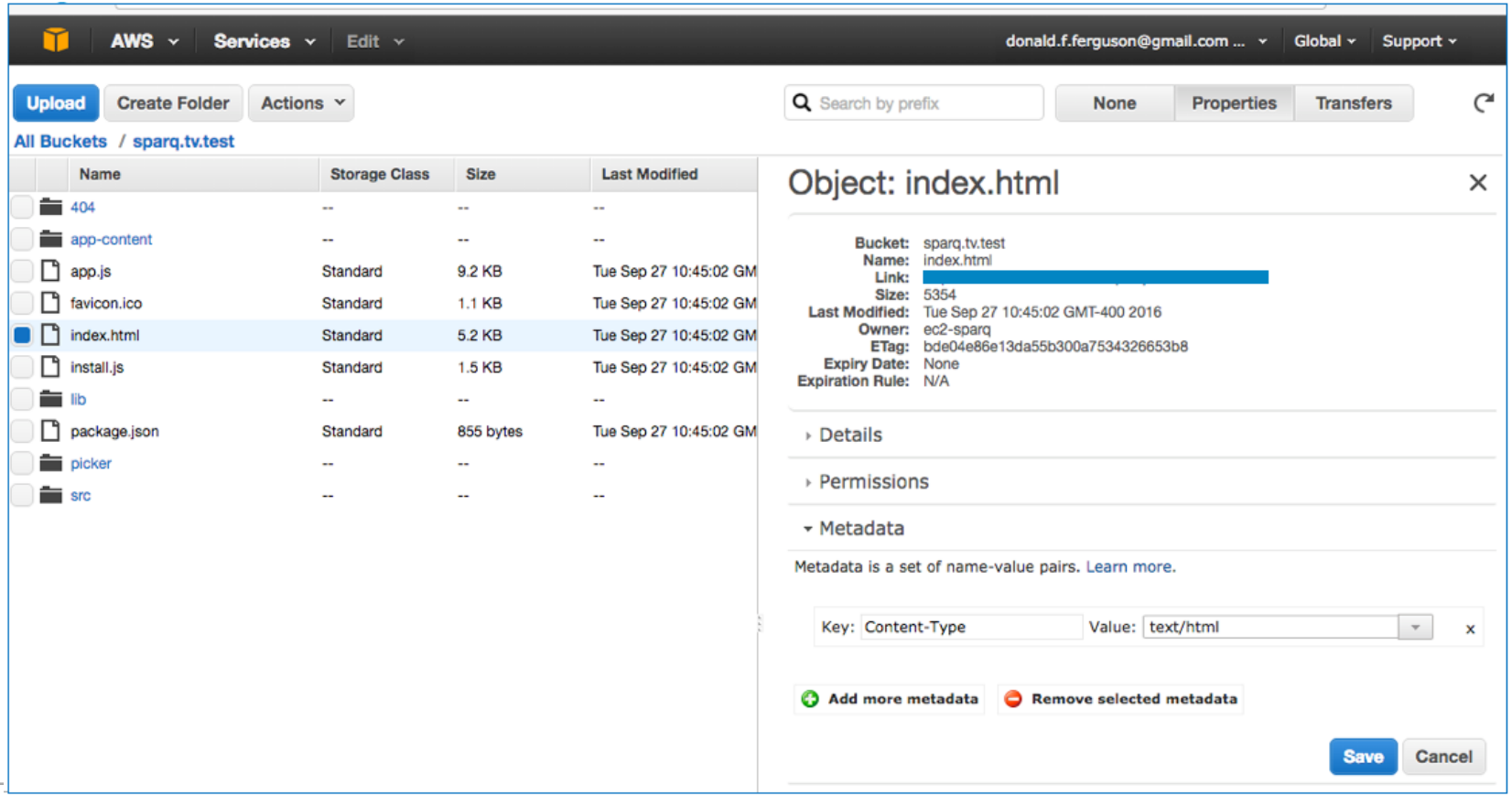
Demo



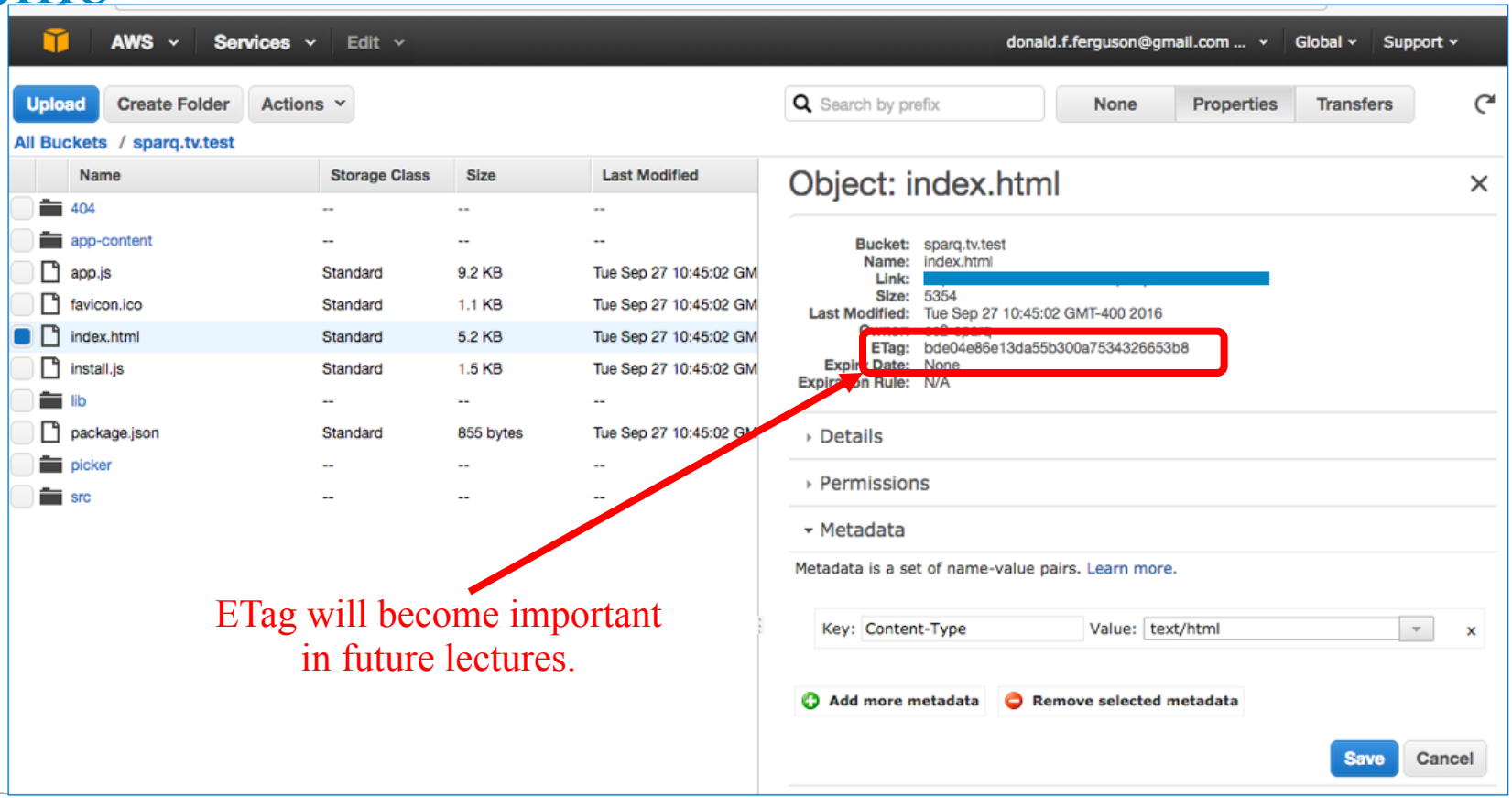
Demo



Demo



Demo



Project Implications

- You web site is relatively static and relatively simple. Something like …
  - /content
    - CSS files
    - Images
    - etc.
  - /app
    - /controllers
      - Controller1.js
      - Controller2.js
      - …
    - /views
      - View1.html
      - View2.html
      - ...

You have the following options:

- Just do it manually through the AWS console.
- Use one of the various git commit/push tools.
- Use s3 synch command line. (http://docs.aws.amazon.com/cli/latest/userguide/using-s3-commands.html)

The `sync` command has the following form. Possible source-target combinations are:

- Local file system to Amazon S3
- Amazon S3 to local file system
- Amazon S3 to Amazon S3

```
$ aws s3 sync <source> <target> [--options]
```

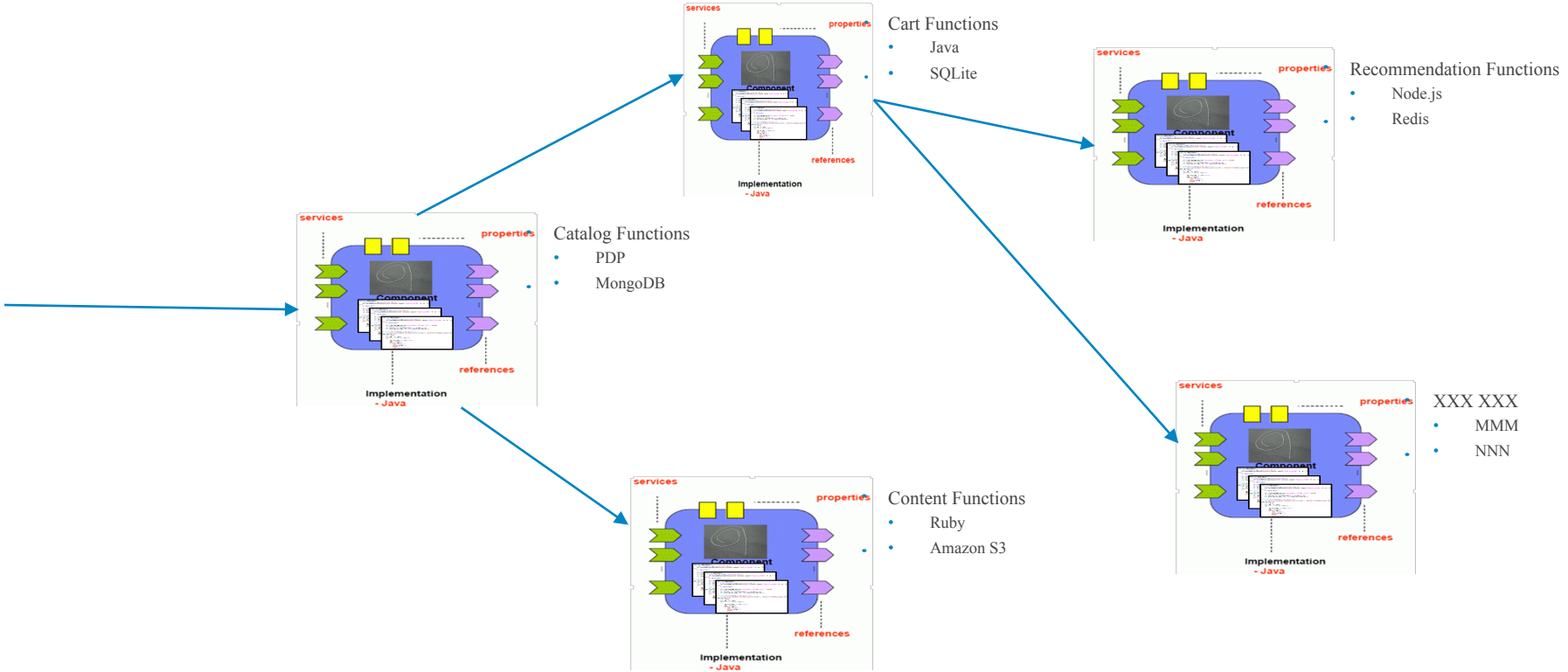
The following example synchronizes the contents of an Amazon S3 folder named *path* in *my-bucket* with the current working directory. `s3 sync` updates any files that have a different size or modified time than files with the same name at the destination. The output displays specific operations performed during the sync. Notice that the operation recursively synchronizes the subdirectory *MySubdirectory* and its contents with *s3://my-bucket/path/MySubdirectory*.

```
$ aws s3 sync . s3://my-bucket/path
upload: MySubdirectory\MyFile3.txt to s3://my-bucket/path/MySubdirectory/MyFile3.txt
upload: MyFile2.txt to s3://my-bucket/path/MyFile2.txt
upload: MyFile1.txt to s3://my-bucket/path/MyFile1.txt
```

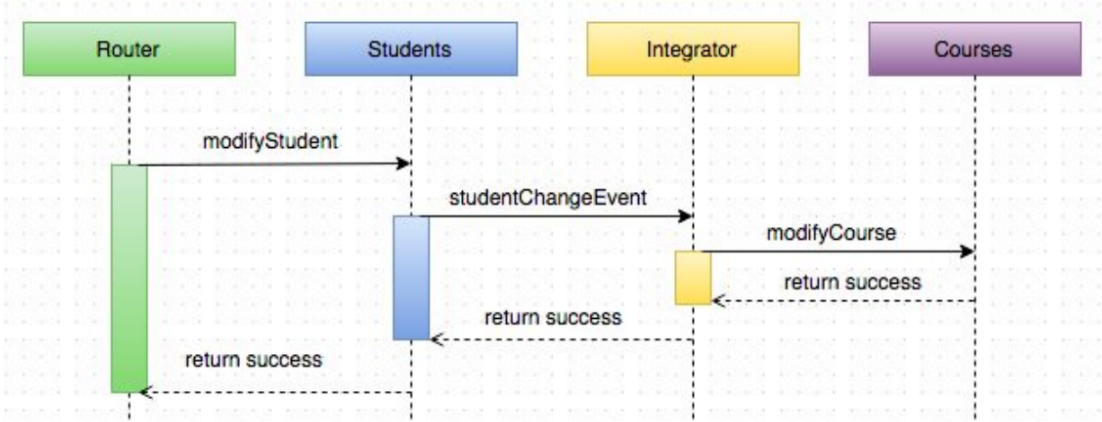
Asynchronous Operations and Message Queues

Asynchronous Operations

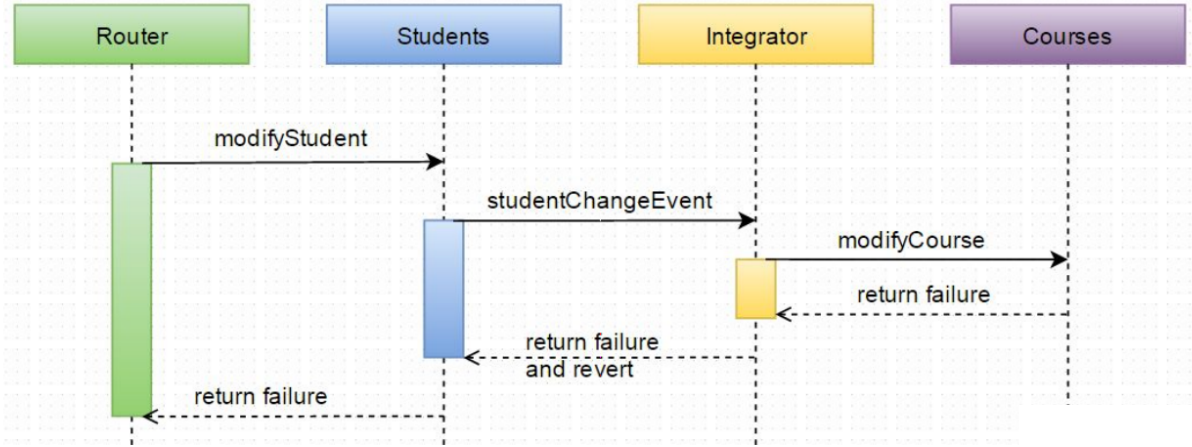
REST Implementations Call REST Services



Synchronous Integration



However, if that request fails when updating a course in Courses, the sequence looks as follows:



Synchronous and Asynchronous

- HTTP operations are inherently *synchronous*
  - Connect
  - Verb
  - Response
  - Disconnect
- The implementation of the operation may "take a while,"
  - A calls B before returning. B calls C and D before returning to A. …
  - The implementation may be computationally or data intensive, e.g. DB query, image transcoding, ...
  - Something in the implementation may operate at "people speed," e.g.
    - Creating an Account may require.
    - Manual approval by "Dave."
- Holding open connections can cause problems
  - A server typically has an upper limit on open connections. Holding a connection for long requests will cause some client calls to "get a busy signal."
  - A connection is more likely to break the longer it is open. A call graph of "long" connections is inherently fragile. One breaks and the whole thing breaks.

Asynchronous Operation

Request:

This is going to "take a while."

```
POST /blogs HTTP/1.1
<xml>
    blogdata
</xml>
```

Response:

```
HTTP/1.1 202 Accepted
Location: /queue/12345
```

So, I am going to

- Tell you that your request looks OK.
- Acknowledge that I am working on it.
- Give you a URL on which you can poll with GET for the answer

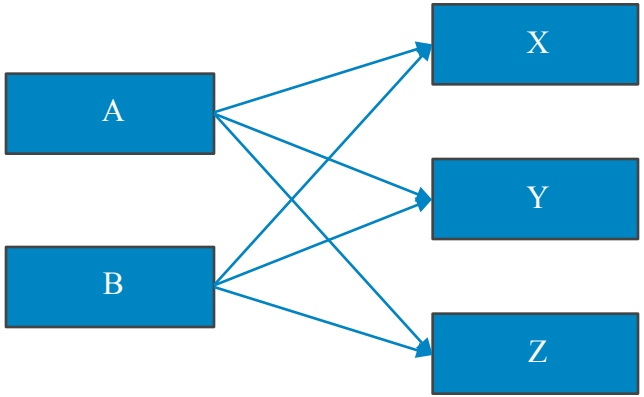
Basic Implementation Approach

- Define a *collection* /QueuedResponses
  - A client can call …/QueuedResponses/21 to get a specific response.
  - You already know how to do this for …/customer, …/address
  - The data format in the table is {id, status, JSONString}, where
    - JSONString is the response you would have received for a synchronous request.
    - Status is one of "in-progress" or "complete."
- A simple implementation would be writing a *façade*
  - Accept request
  - Create new table entry with status = "in progress"
  - Return 202 and URL
  - Call the *actual* implementation
  - Update the database table entry with the JSON result
- Most application platforms have middleware approaches to support registering callbacks, threads, etc. The implementation would typically
  - Invoke some long running action, e.g. DB query, workflow process and register a callback
  - The callback implementation updates the entry in the response table.

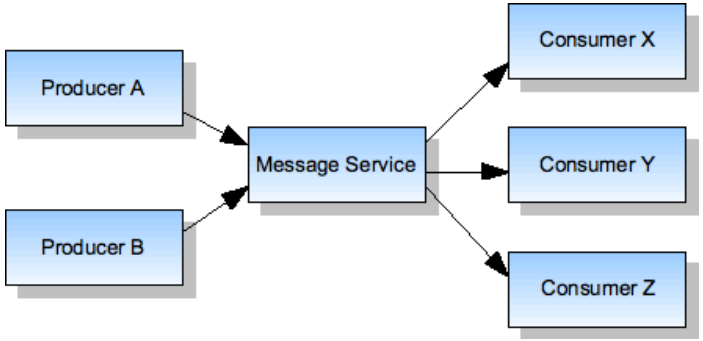
Message Driven Processing

Message Queue Service Pattern

Anti-Pattern



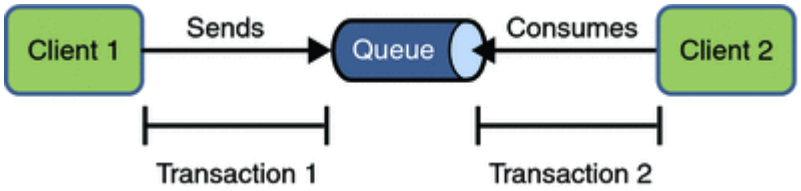
Best Practice Pattern

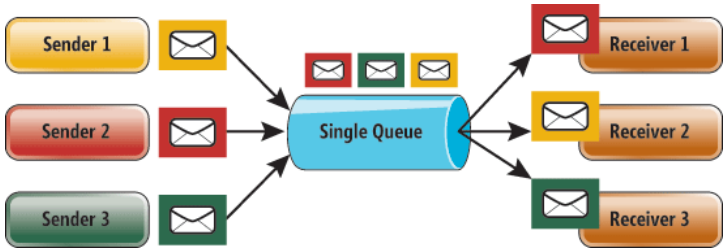


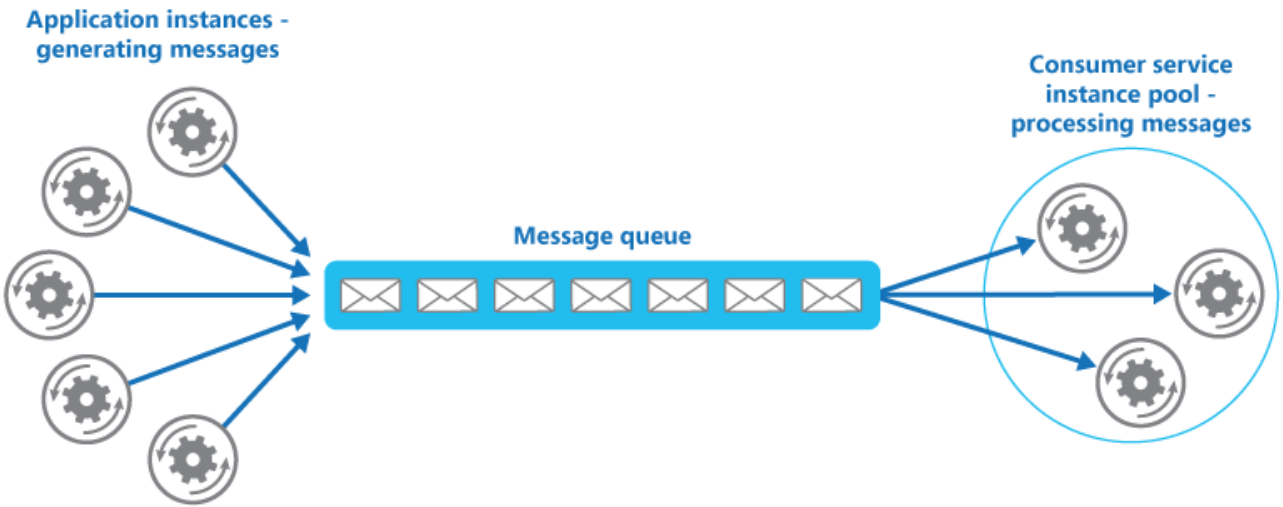
Point-to-Point communication between modules becomes fragile at scale

- Adding modules P and Q requires finding and configuring all senders.
- I want to send M to any one of X1, X2 and X3, but only one.
- I want to make sure that someone processes M but do not want to hold the transaction until I get a response (I may not even need the response).
- My destinations are not "up" at the same times I am.

Message Exchange Patterns







Why Would I Use a Queue

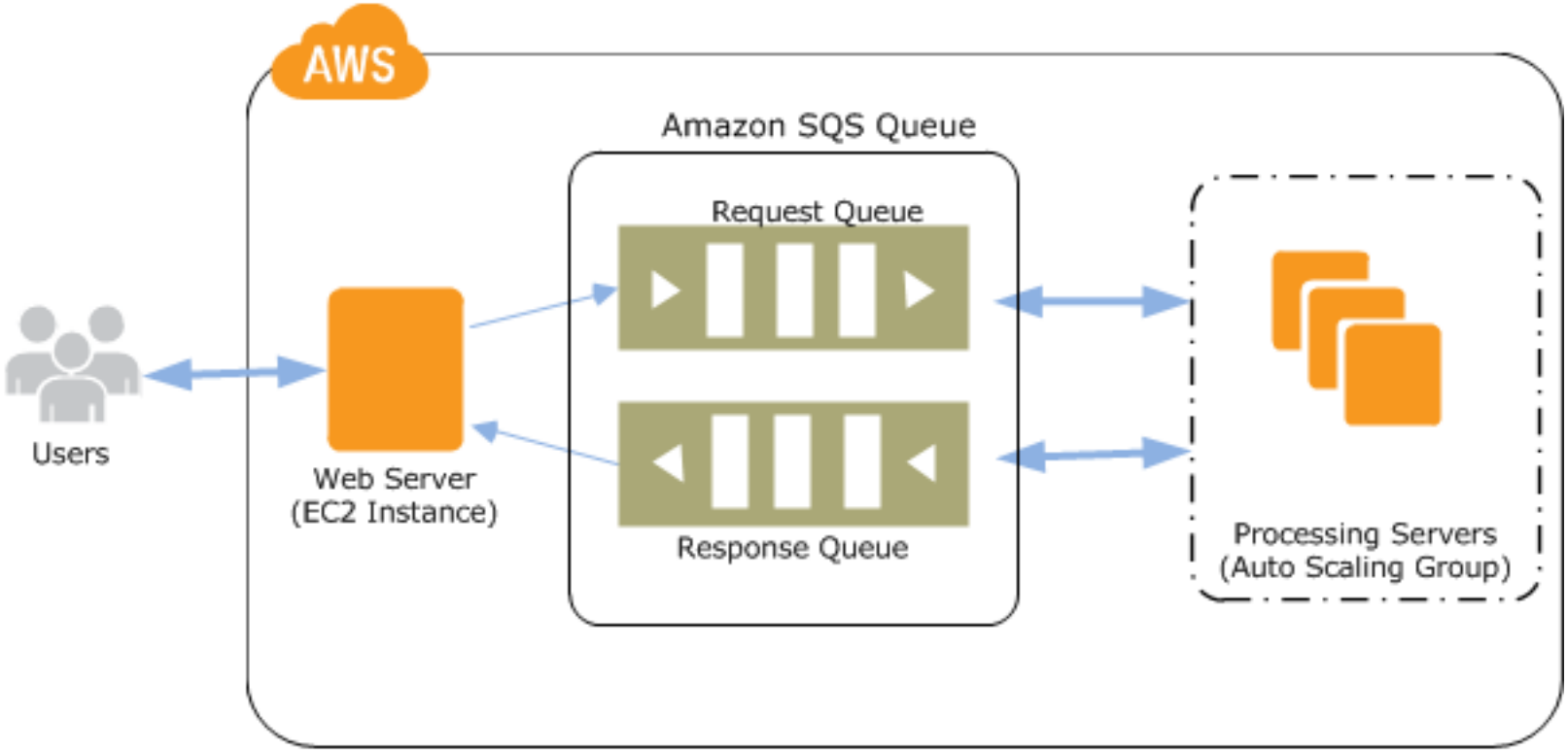
http://www.javacodegeeks.com/2013/06/working-with-amazon-simple-queue-service-using-java.html is one example)

- Flow control
  - My server may be able to process 100 reqs/second
  - There are thousands of clients
  - The aggregate rate could be 10 req/s or 300 req/s
  - Without a queue, clients would get connect failures under load
- The request is going to take a long time
  - Image reformatting and tagging
  - I cannot have my code "wait" for a response
  - And I do not want to write a lot of "Is it done yet?" calls
- I do not know (or care) which one implements the logic
  - I have 5 – 10 image processors for GIFs
  - I have 7 – 10 processors for .wav files
  - I do not want to know which ones are active, when and processing what.
  - I just want to put the "thing in a queue" knowing that someone will eventually pick it up.
- Think "email" for API calls

Many use cases
and patterns for
Message driven processing.

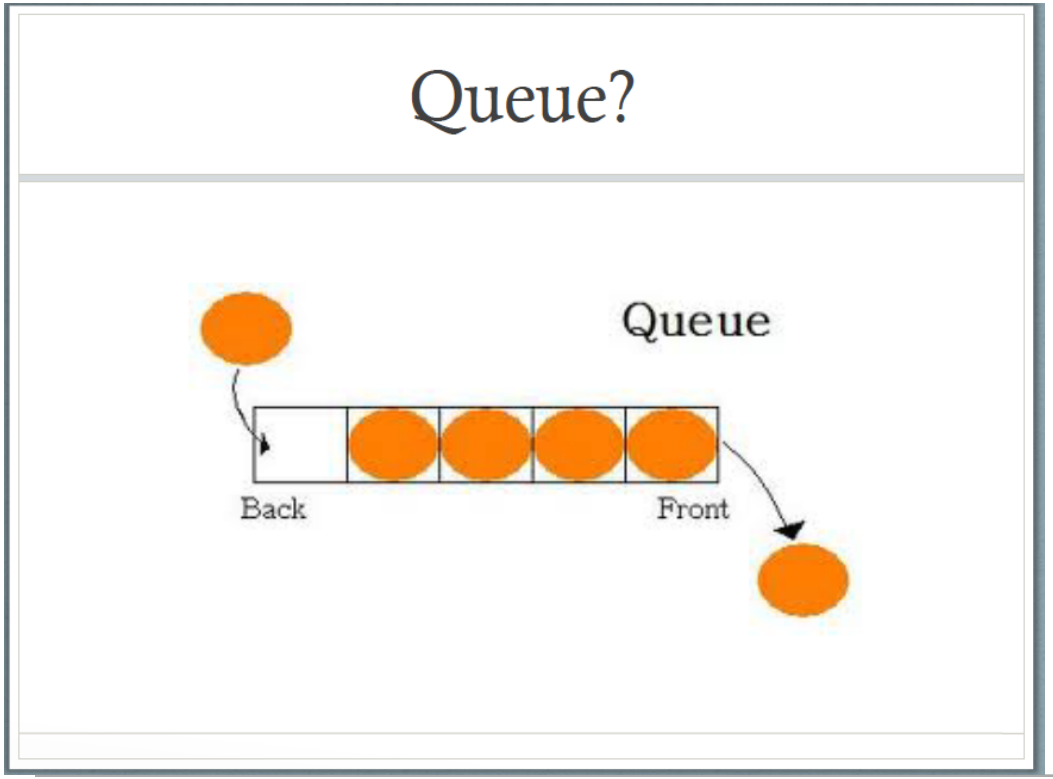
Amazon SQS

Amazon SQS – Example



SQS

http://www.slideshare.net/sandipkchaudhari/sqs-and-loose-coupling-presentation



Queue Operations

- Create Q
- Delete Q
- Send Message
- Receive Message
- Delete Message

FIFO Semantics

- ~~Strict, limited scalability.~~
- Best effort, highly scalable.
- ~~Transactional~~

What is SQS?

- Amazon ***S***imple ***Q***ueue ***S***ervice is a **fast, reliable, scalable and fully managed** queue service

- Amazon SQS enables **asynchronous message-based communication** between **distributed components** of an application

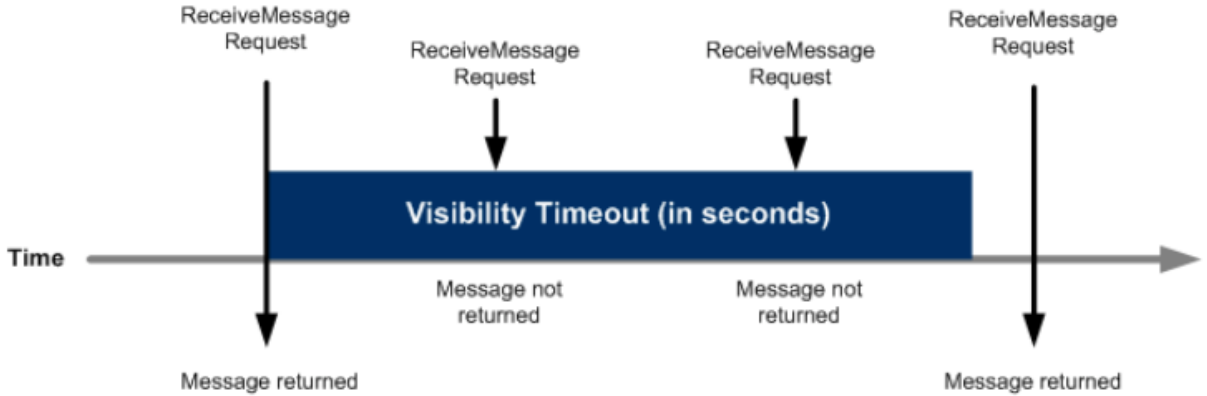
SQS – What it can do?

- Amazon AWS SQS page:
  - SQS makes it simple and cost-effective to ***decouple the components*** of a cloud application
  - You can use SQS to transmit any volume of data, at any level of throughput, ***without losing messages*** or requiring other services to be always available
  - With SQS, you can offload the administrative burden of ***operating and scaling*** a highly available messaging cluster, while paying a low price for only what you use.

SQS – Major Features

- Redundant Infrastructure
- Multiple writers and readers
- Configurable settings per queue
- Variable message size (up to 256 KB)
- Access Control
- Delay Queues

SQS – Message Visibility Timeout



SQS – Message Visibility Timeout

- The visibility timeout clock starts ticking once Amazon SQS returns the message
- During that time, the component processes and deletes the message
- But what happens if the component fails before deleting the message?
- If your system doesn't call *DeleteMessage* for that message before the visibility timeout expires, the message again becomes visible to the ReceiveMessage calls placed by the components in your system and it will be received again
- If a message should only be received once, your system should delete it within the duration of the visibility timeout
- Each queue starts with a default setting of 30 seconds for the visibility timeout

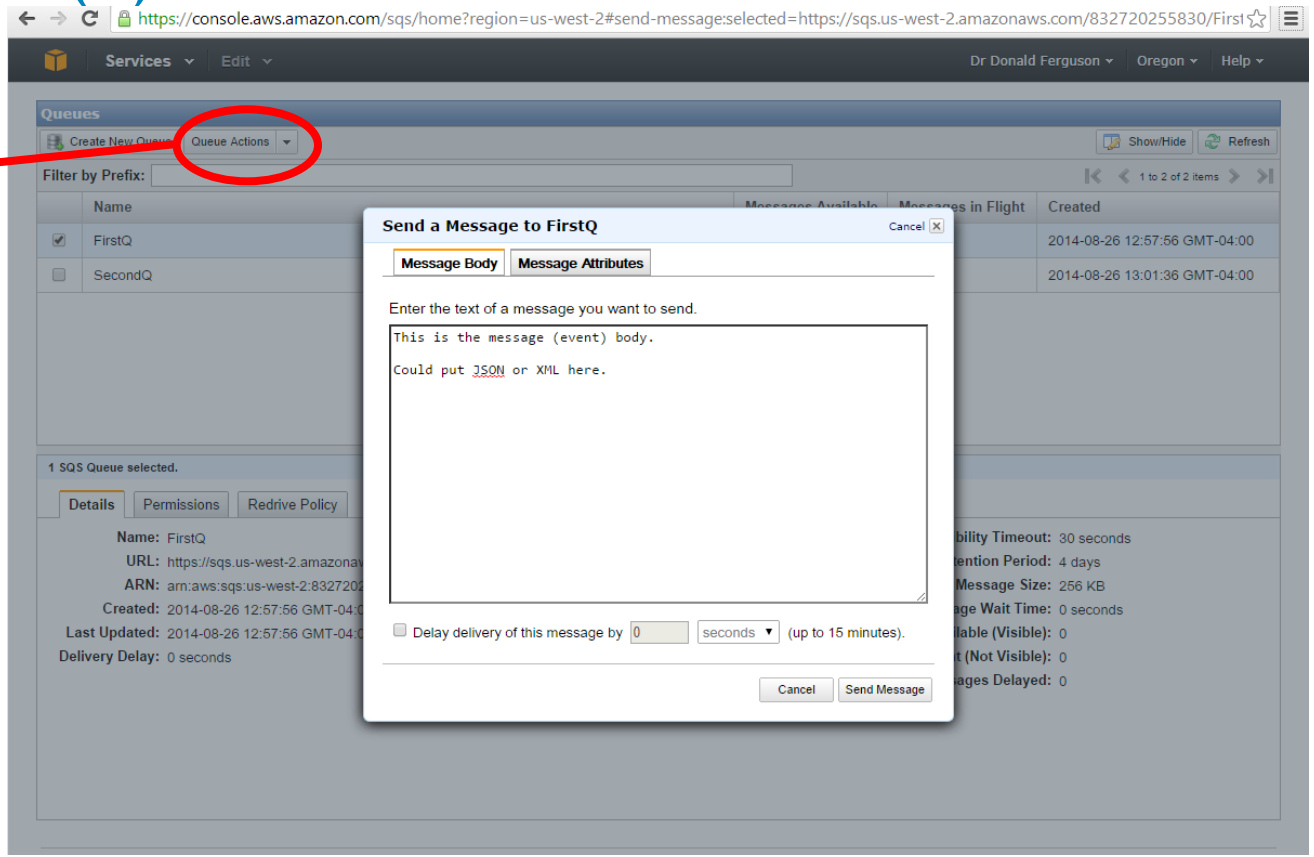
SQS Message Lifecycle

- Messages that are stored in Amazon SQS have a lifecycle that is easy to manage but ensures that all messages are processed.

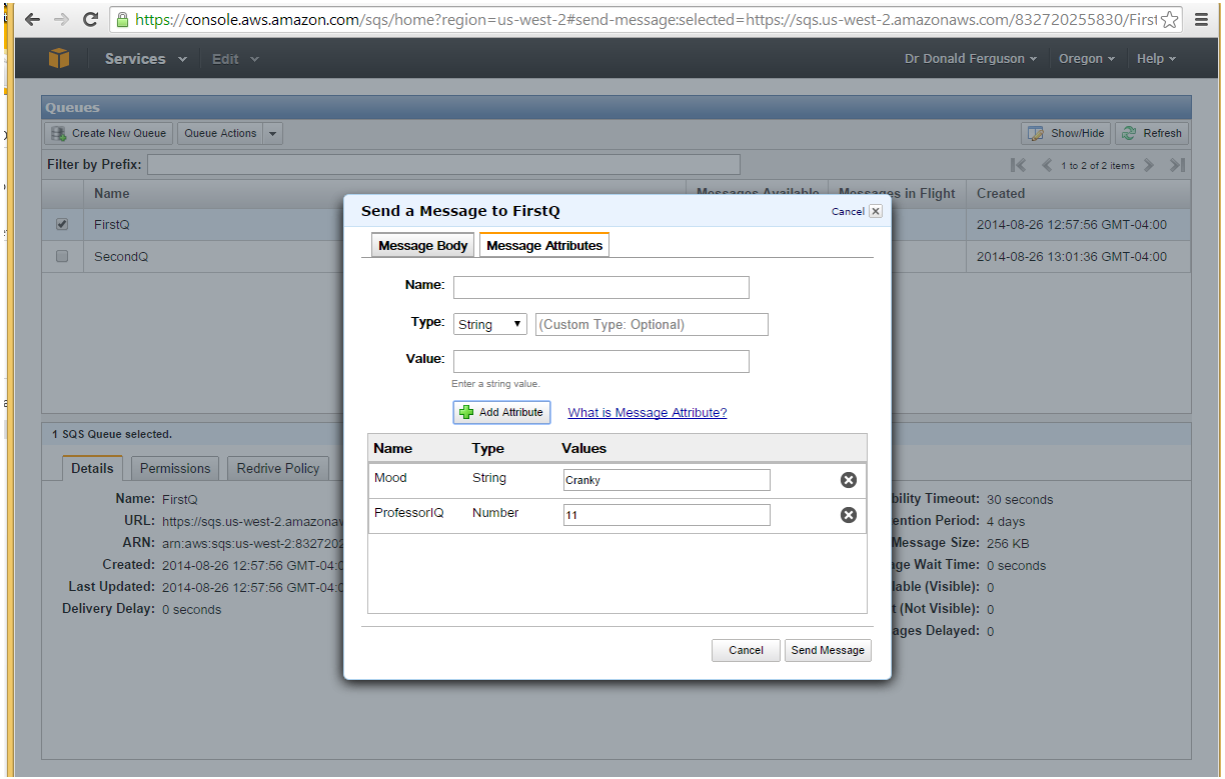
1. A system that needs to send a message will find an Amazon SQS queue, and use *SendMessage* to add a new message to it
2. A different system that processes messages needs more messages to process, so it calls *ReceiveMessage*, and this message is returned
3. Once a message has been returned by *ReceiveMessage*, it will **not** be returned by any other *ReceiveMessage* until the **visibility timeout** has passed. This keeps multiple computers from processing the same message at once.
4. If the system that processes messages successfully finishes working with this message, it calls *DeleteMessage*, which removes the message from the queue so no one else will ever process it. If this system fails to process the message, then it will be read by another *ReceiveMessage* call as soon as the **visibility timeout passes**

Amazon SQS (I)

Selected
"Send Msg"



Amazon SQS (I)



SQS API

(AWS.Request) **addPermission**(params = {}, callback)

Adds a permission to a queue for a specific principal.

(AWS.Request) **changeMessageVisibility**(params = {}, callback)

Changes the visibility timeout of a specified message in a queue to a new value.

(AWS.Request) **changeMessageVisibilityBatch**(params = {}, callback)

Changes the visibility timeout of multiple messages.

(AWS.Request) **createQueue**(params = {}, callback)

Creates a new queue, or returns the URL of an existing one.

(AWS.Request) **deleteMessage**(params = {}, callback)

Deletes the specified message from the specified queue.

(AWS.Request) **deleteMessageBatch**(params = {}, callback)

Deletes up to ten messages from the specified queue.

(AWS.Request) **deleteQueue**(params = {}, callback)

Deletes the queue specified by the **queue URL**, regardless of whether the queue is empty.

(AWS.Request) **getQueueAttributes**(params = {}, callback)

(AWS.Request) **getQueueUrl**(params = {}, callback)

Returns the URL of an existing queue.

(AWS.Request) **listDeadLetterSourceQueues**(params = {}, callback)

Returns a list of your queues that have the RedrivePolicy queue attribute configured with a dead letter queue.

For more information about using dead letter queues, see Using Amazon SQS Dead Letter Queues.

.

(AWS.Request) **listQueues**(params = {}, callback)

Returns a list of your queues.

(AWS.Request) **purgeQueue**(params = {}, callback)

Deletes the messages in a queue specified by the **queue URL**.

When you use the `PurgeQueue` API, the deleted messages in the queue cannot be retrieved.

When you purge a queue, the message deletion process takes up to 60 seconds.

(AWS.Request) **receiveMessage**(params = {}, callback)

Retrieves one or more messages, with a maximum limit of 10 messages, from the specified queue.

(AWS.Request) **removePermission**(params = {}, callback)

Revokes any permissions in the queue policy that matches the specified `Label` parameter.

(AWS.Request) **sendMessage**(params = {}, callback)

Delivers a message to the specified queue.

(AWS.Request) **sendMessageBatch**(params = {}, callback)

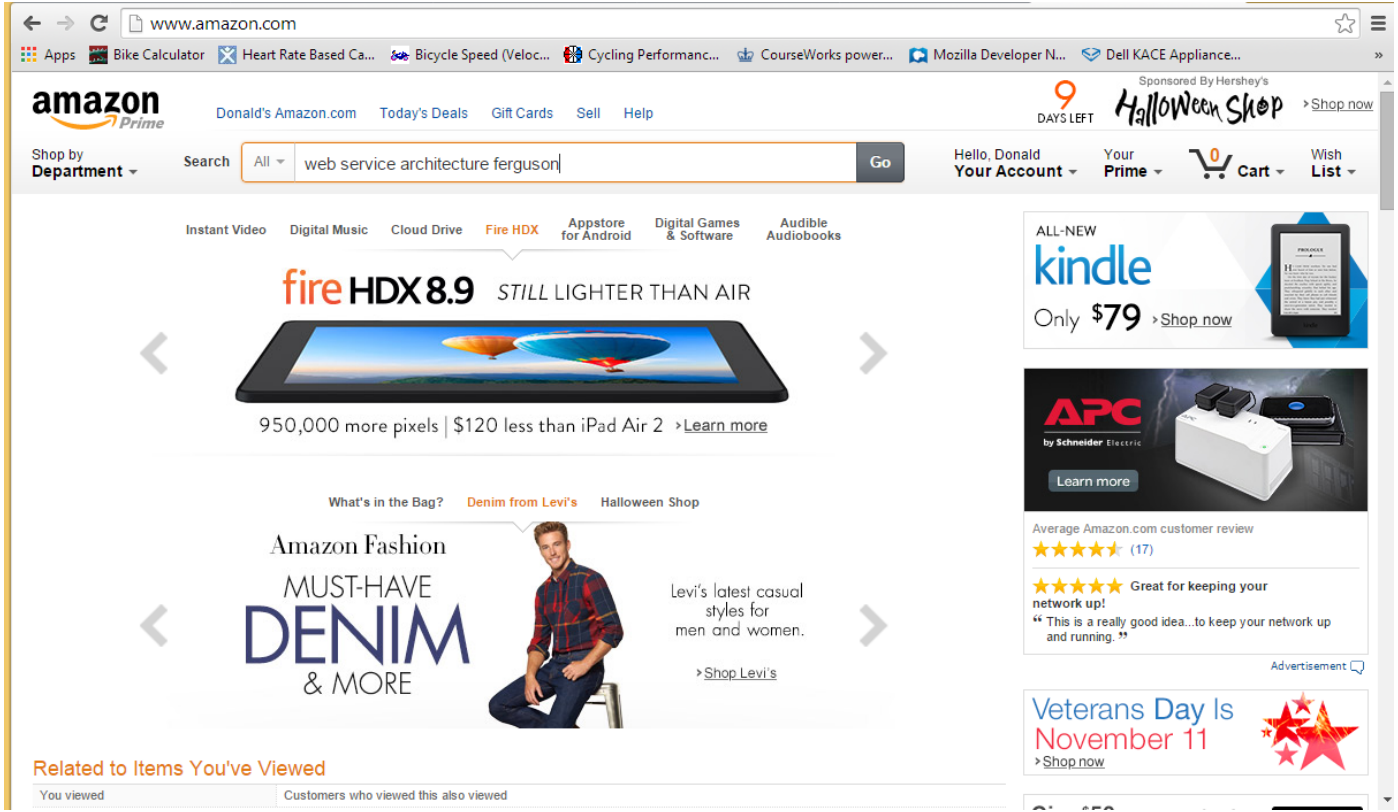
Delivers up to ten messages to the specified queue.

(AWS.Request) **setQueueAttributes**(params = {}, callback)

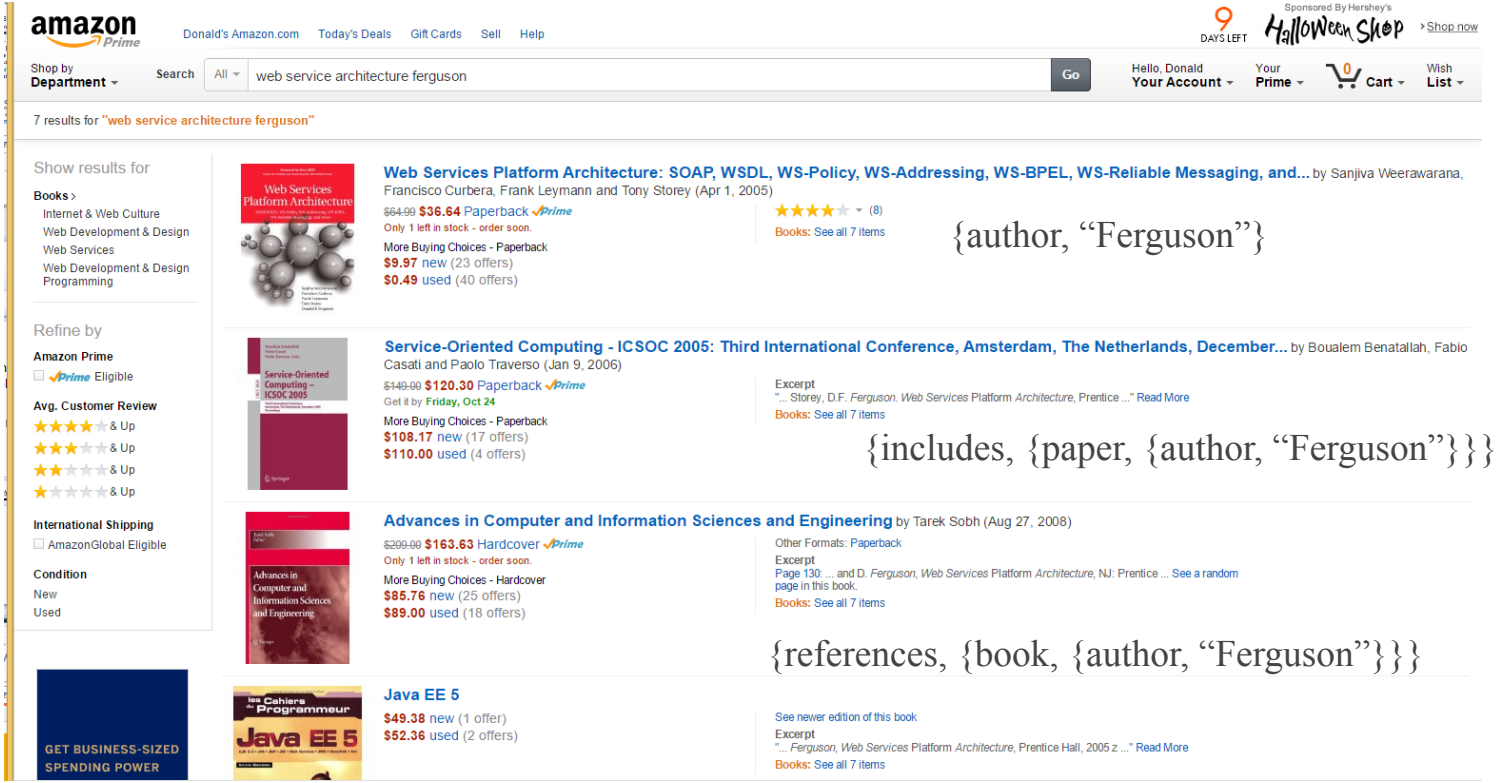
Sets the value of one or more queue attributes.

Content Management Introduction

Complex Website Catalog

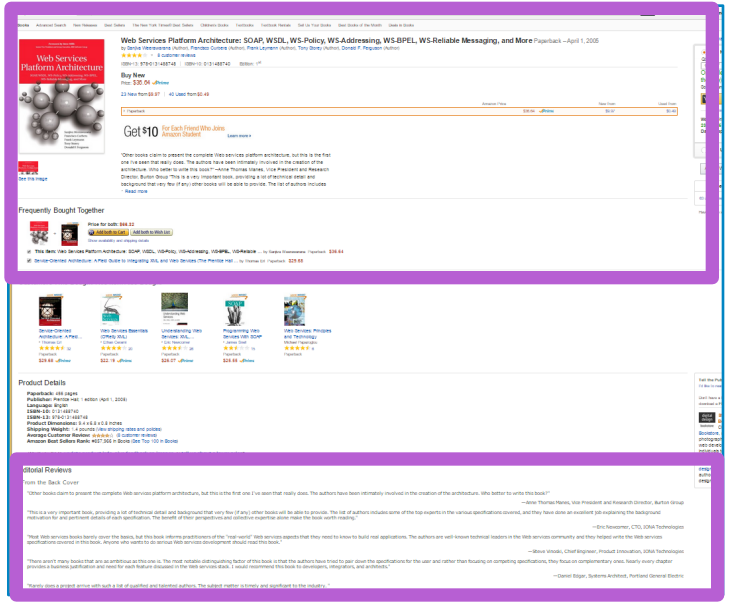


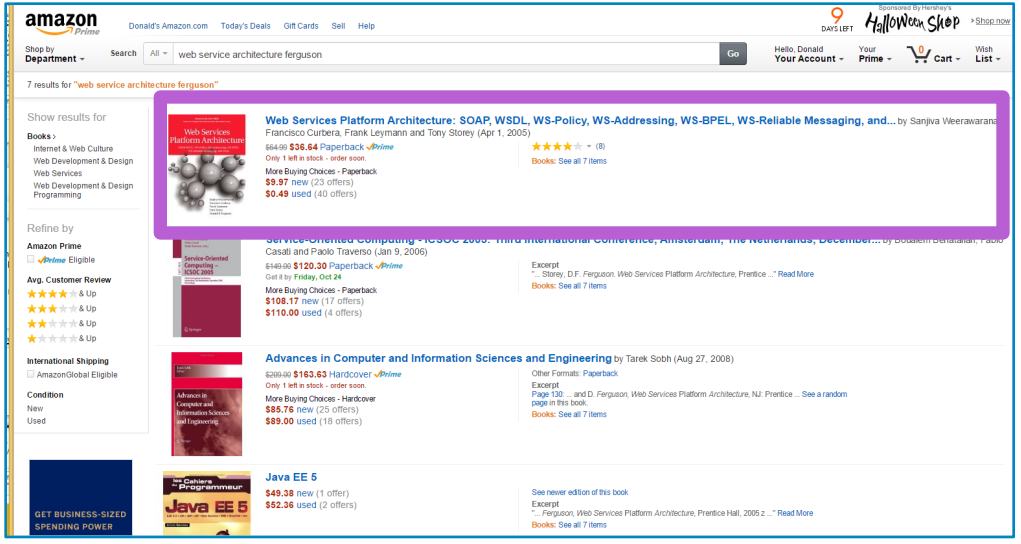
Query Results



Data and Rendering

- Two different *renderings*
- Of the same underlying document(s)

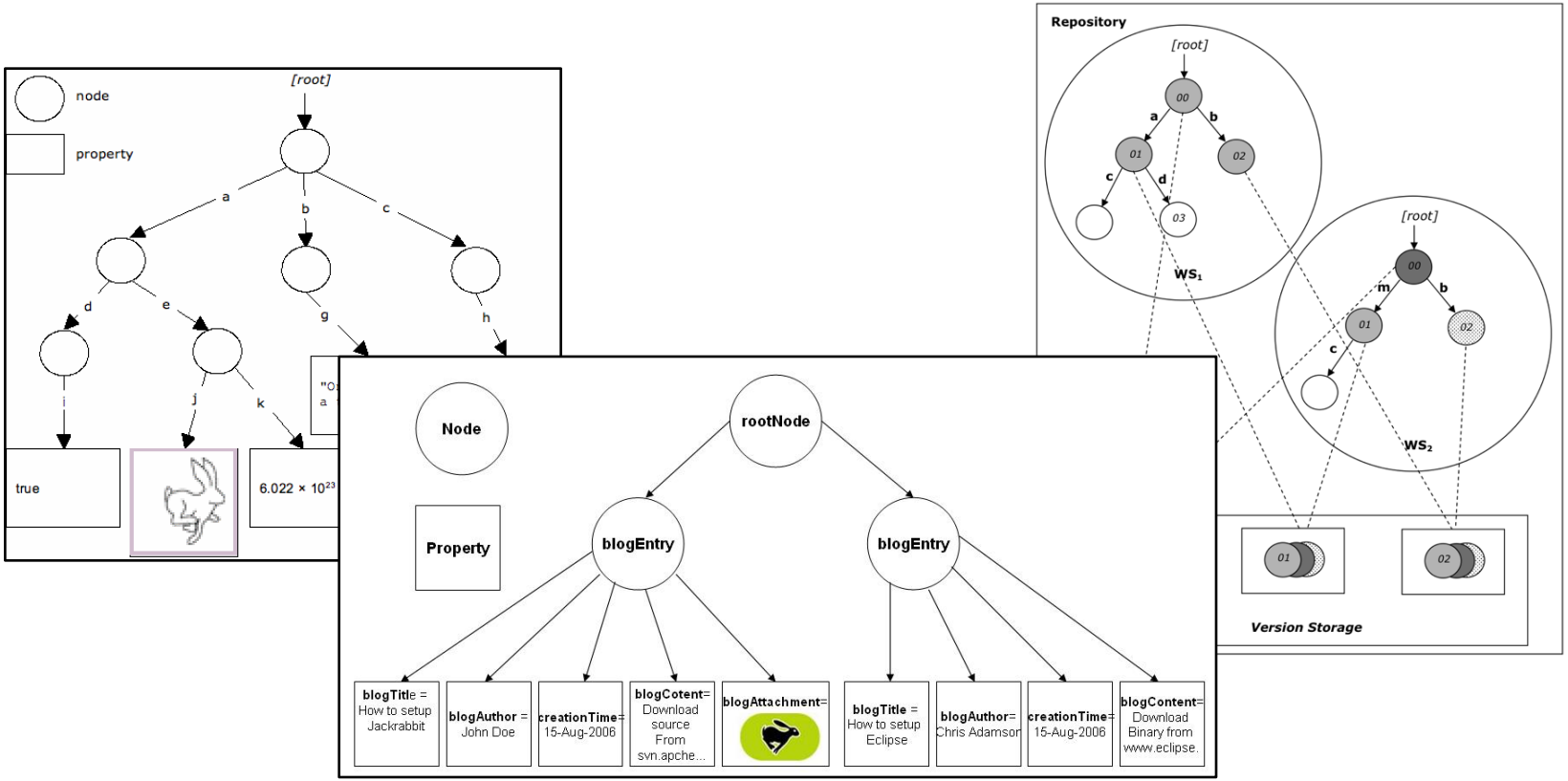




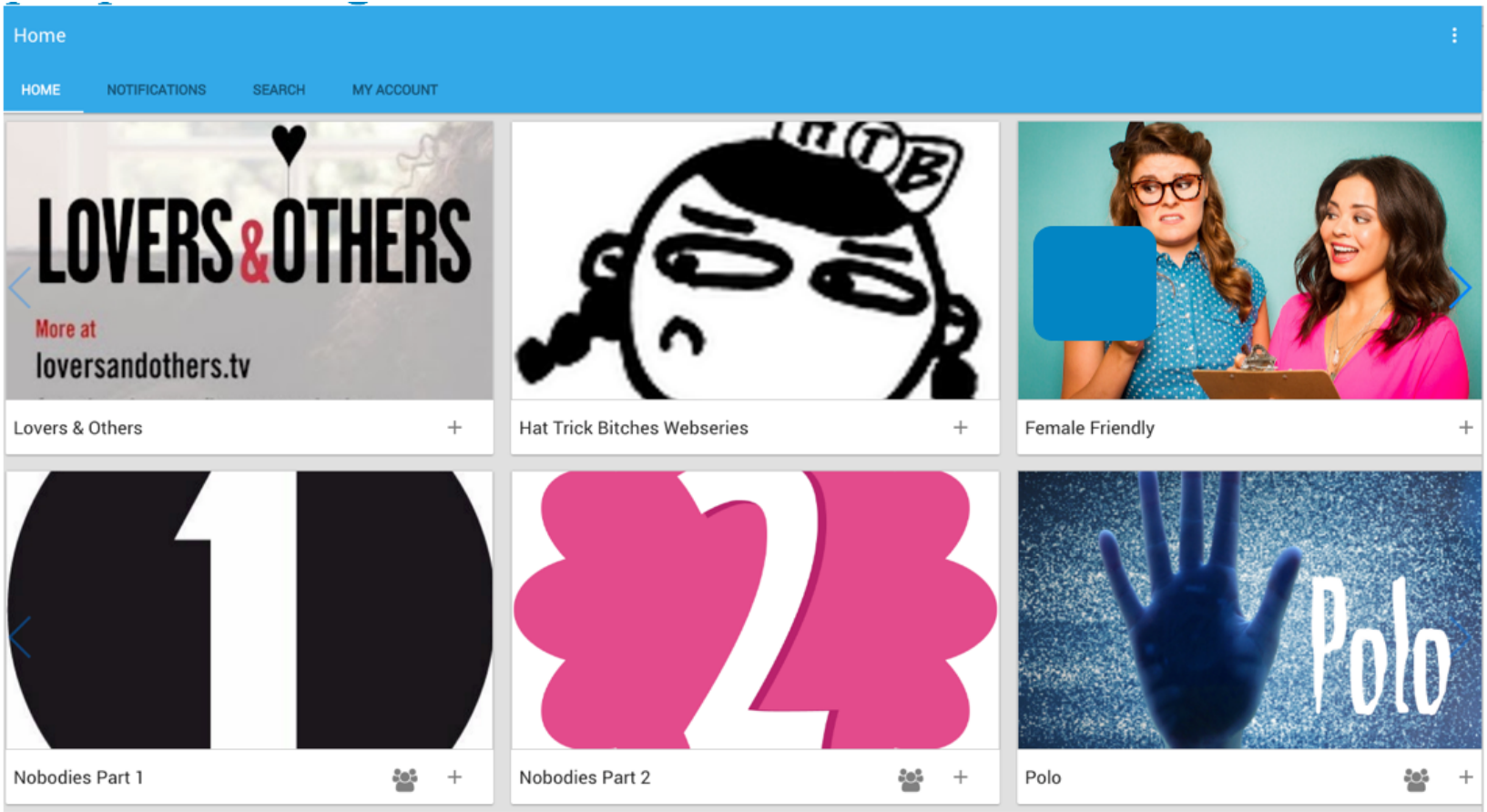
Content Management System
Some Terminology

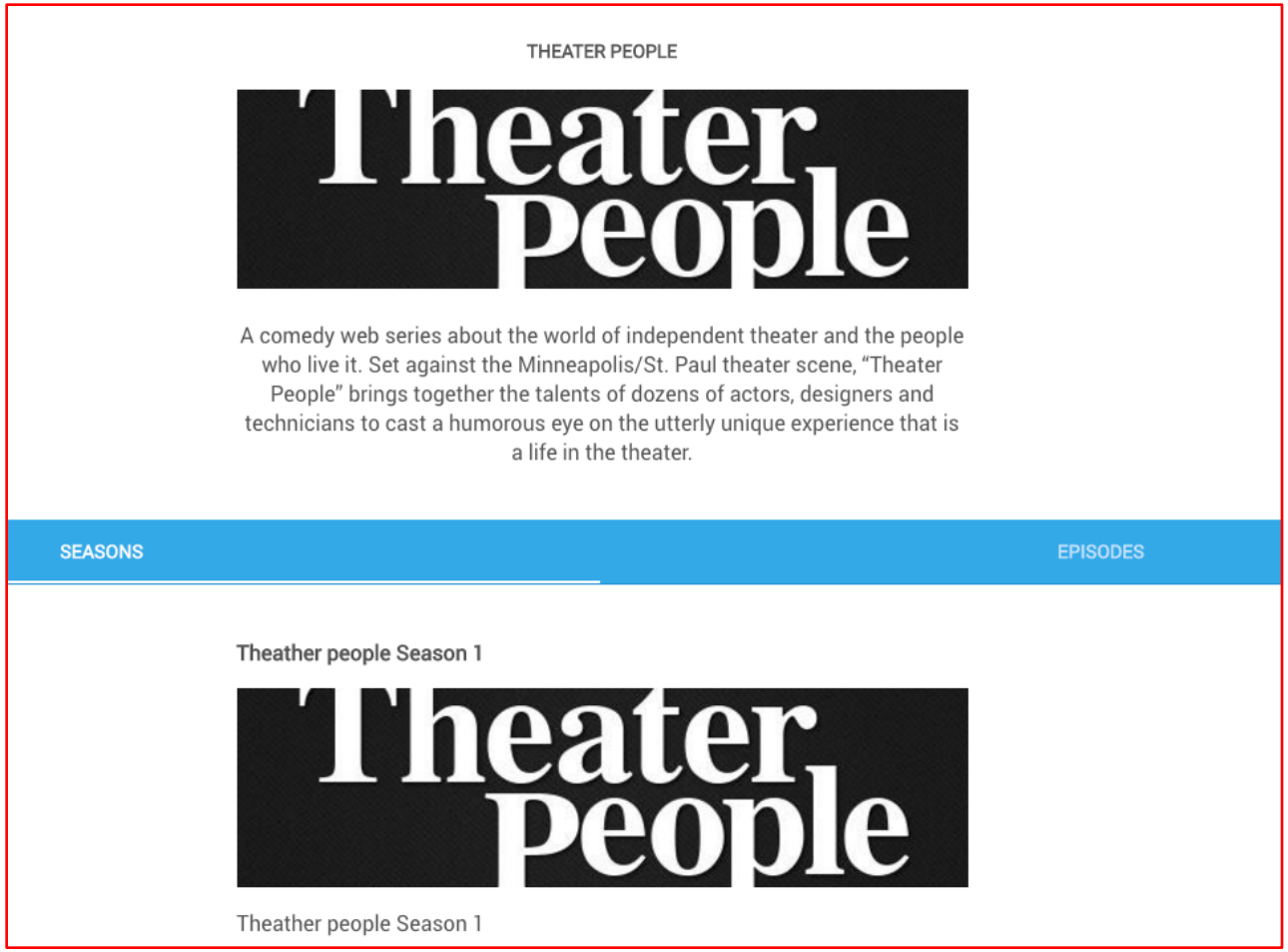
- A content management system (CMS) is a computer application that
  - Allows publishing, editing and modifying content, organizing, deleting as well as maintenance from a central interface.
  - Such systems of content management provide procedures to manage workflow in a collaborative environment.
- A *web* content management system is a bundled or stand-alone application
  - To create, deploy, manage and store content on Web pages.
  - Content includes text and embedded graphics, photos, video, audio, …
  - Content includes *code* that displays content in a specific way
  - A Web CMS may catalog and index content, select or assemble content at runtime, or deliver content to specific visitors in a requested way, such as other languages.

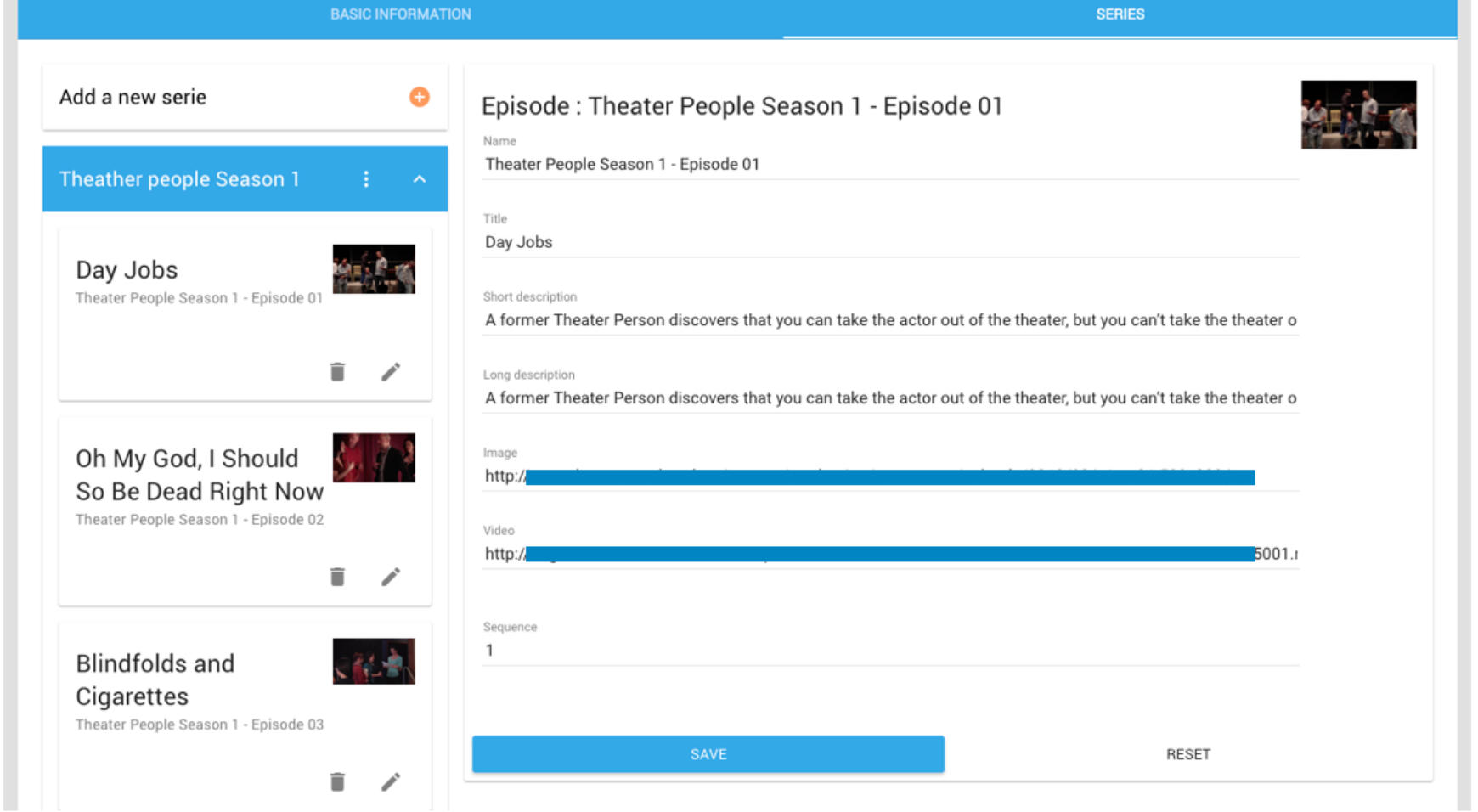
CMS Data Model

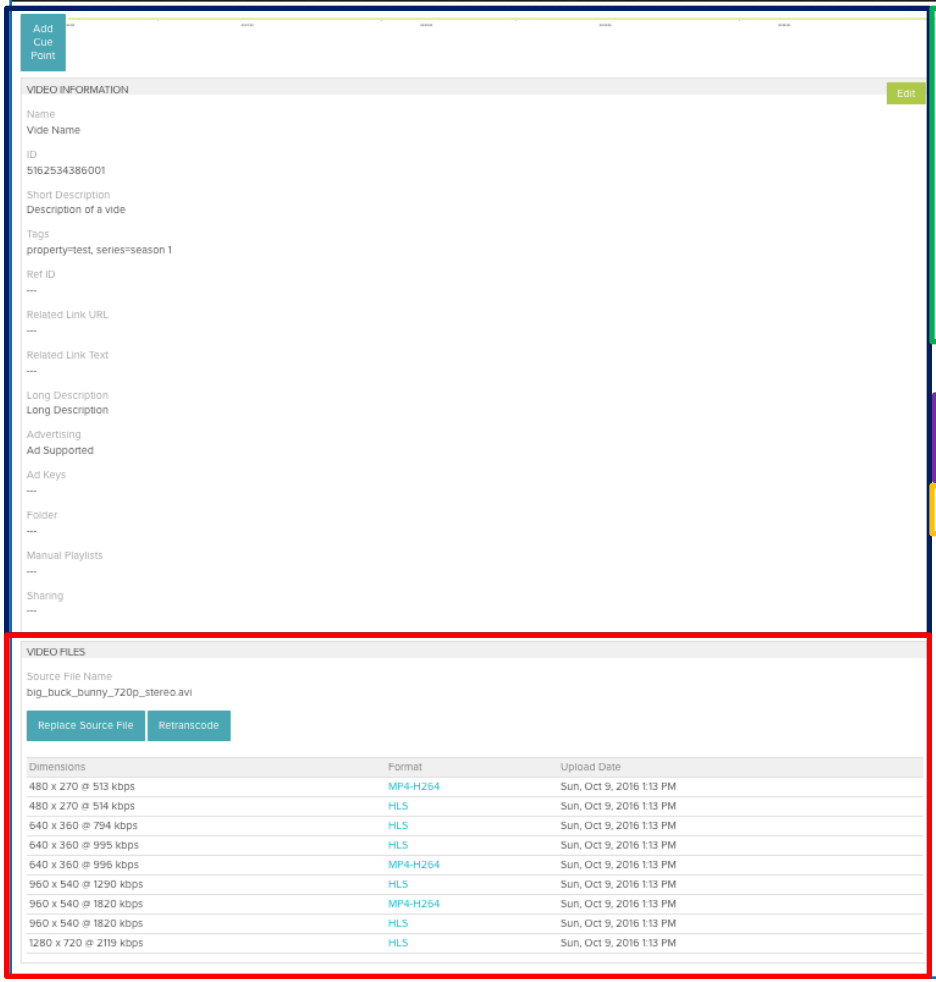


Sparq Home Page









Content Managed

- Metadata
- Renditions
- Poster image
- Thumbnail image
- Text/subtitle tracks
- Availability (time, geos, …)

Putting the Pieces Together

4th Project (Part…) Building on 3rd Project

- Define and provide a Lambda/REST interface to a "content catalog" in DynamoDB.
  - Model: Content Instance
    - Property (e.g. HBO, CBS) is a set of one or more Franchises.
    - A Franchise (e.g. Game of Thrones, Bing Bang Theory) is a set of one or more Series.
    - A Series (e.g. Season 1, Christmas Specials) is a set of one or more Episodes.
    - An Episode is an instance of a "TV show."
    - Data for each type is just (ID, name).
  - Co
- Gra
  -
  -
  -
  -
  -
- Exp
  -
  -
- Log
  -
  - Us

Content Loading Engine

- Manifest containing metadata, links to thumbnail and video.
- Send the manifest to a queue
- Pipeline of queues/processors
  - Call REST Service to create content instance and metadata
  - Load thumbnail into S3
  - Load video into S3