‘Mewtube’ Example Application

Table of Contents

Introduction

In response to minimum requirements from a potential employer, I've created http://mewtube.thirdstart.com as a basic video storage and playback application that:

1. Demonstrates writing both a "web service" and an application: in this case, a simple JSON-based service tier and a bare-bones UI demonstrating its functions
2. Shows use of modern Java/JVM technologies with the Spring Security framework
3. Uses a cloud-hosted (Amazon RDS) PostgreSQL database to store application data
4. Demonstrates Spring-based authentication enforcement
5. Implements cloud-focused use of storage for uploads
6. Document assumptions and tradeoffs (later in this document)

To fulfill "extra credit" requirements, it also:

1. Is deployed and running on an EC2 instance with Amazon Web Services
2. Uses an S3 bucket to store uploaded videos
3. Has API documentation (later in this document)
4. Shows basic information about the video (e.g. duration, original resolution)
5. Provides a straightforward UI: it redirects users to an authentication page and provides uploading and transcoding feedback for uploaded videos
6. Transcodes uploaded videos to an HLS stream and provides a streaming player that works across most desktop and mobile devices

TI;dr - Where's the code?

The Git repo storing the application has what may look like a lot of code. To jumpstart development, I based the application on a template I maintain for Grails 3.x + Spring Security + Twitter Bootstrap prototyping work. You'll probably be most interested in:

1. AmazonGateway - Abstracts the AWS API away from the API tier
2. VideoController - Provides a simple JSON api used by the client
3. VideoService - A transactional service that coordinates the domain model and Amazon operations

Assumptions

Video is Tricky

Video is a very hard problem to solve across all devices and platforms. Requirements were to accept an mp4 file of up to 10 megabytes and playback was considered “extra credit.” Therefore:

1. It might not work for uploads from mobile devices
2. Video might not play back on all devices, but has been tested on Mac OS X/Chrome, Mac OS X/Safari, and iOS phones

“Java” is a Vague Term

The requirements were to use “Java.” I chose to use the “Java” platform, but not the language:

1. The application is written in Groovy: in addition to being cleaner and nearly as performant, its brevity forces a developer to show whether or not they understand the tools they’re using or if they’re just pasting together answers from StackOverflow
2. The application runs within Grails: There’s no need to make anyone read another VideoDao.java and VideoDaoImpl.java pair in a code exercise like this. **In production, this should be a raw Spring Boot application providing a headless API to a single-page application, not Grails combination of service and client as one codebase.**

What’s Good About It

Cloud-First Design

Even though it’s using Grails to bootstrap itself, it’s designed to be refactored to a smaller-scoped application (e.g. microservice suite):

1. Server resources and storage are minimal: uploaded videos go straight to S3 via a signed upload request
2. AWS abstracted: the “heavy lifting” of coordinating uploads, transcoding jobs, and their resolution is kept with a class (AmazonGateway) that doesn’t know anything about Grails or the runtime of the application

Standalone Deployment

While the resultant .jar does carry the weight of the Grails dependencies, the application itself is a Spring Boot application. There's no external continer: it's just running as `java -jar` and ready to be put into a container.

What's Not So Great

Video Security by Obscurity

HLS playlists defer clients to secondary files: I considered it outside of the scope of this demonstration to work with the Flowplayer API to dynamically control the URLs contained in HLS playlists to be signed URLs. In other words, the only thing protecting video privacy is the obscurity of guessing UUID strings. (If it's good enough for Facebook, it'll do for this.)

The Interface is Junk Code

The HTML interface isn't driven by Angular, React, or any modern framework. It's dirt-simple Model-View-Controller with jQuery and a little bit of Vue.js providing data binding within a monolithic template.

Very Incomplete Javadoc

I have to admit that I've left a lot of generated @param and @return annotations in the Javadoc: most notations state the intent of a method and move on.

Appendix 1 - API Documentation

The API provided by this application is rudimentary. It's not REST. It's just simple, "GET-POST-WHATEVER" URLs with a parameter or two in the query string.

Operations

/list

Returns a JSON representation of an array of uploaded videos associated with the current user including their metadata and transcoding status.

Expects parameters:

(none)

/ (index)

Returns JSON representing an uploaded video. Expects a video with an S3 key matching the ID parameter to already by uploaded. Validates format of the video, returning exceptions if the format is invalid. Creates and launches a transcoding job if the video has not been transcoded.

Expects parameters:

- id - A string representation of a UUID

/video/policy

Returns a JSON object representing a signed upload policy allowing a client application to upload a video directly to S3.

Expects parameters:

(none)

/remove

Removes a video and its S3 assets.

Expects parameters:

- id - A string representation of a UUID

/transcodingStatus

For a given video, returns a JSON object stating whether or not transcoding has started and whether or not it has completed.

Expects parameters:

- id - A string representation of a UUID