# Assignment 2 Covering Weeks 3-6

## Overview

This assignment will build on the ideas in the original video service to add OAuth 2.0

authentication of clients and the ability to "like" videos. To complete

this assignment, you must allow users to authenticate using the OAuth 2.0 Password Grant flow.

Once authenticated, users must be able to like/unlike videos, as well as search for video

by name and duration. In addition, video data must be stored in a Spring Data JPA repository.

Finally, all communication must take place over https, port 8443. The code to setup HTTPS

on the embedded Tomcat instance is included in the Application class.

You should note that this assignment also requires concepts used in the first assignment, such

as request mapping and JSON marshalling in request/response bodies.

## Warning

UNDER NO CIRCUMSTANCES SHOULD YOU USE THE INCLUDED KEYSTORE IN A PRODUCTION APP!!!

UNDER NO CIRCUMSTANCES SHOULD YOU USE THIS APP "AS IS" IN PRODUCTION!!!

## Running the Application

Please read the instructions carefully.

To run the application:

1. Right-click on the Application class in the assignment project->Run As->Java Application (the

application may try to start and fail with an error message - this is OK). If the application

successfully starts, stop the application before proceeding to the next step.

2. (Menu Bar) Run->Run Configurations

3. Under Java Applications, select your run configuration for this app's Application class that

was just created in step 1 (if you select the run configuration, it should list the assignment

as the project name)

4. Open the Arguments tab

5. In VM Arguments, provide the following information to use the

default keystore provided with the sample code:

-Dkeystore.file=src/main/resources/private/keystore -Dkeystore.pass=changeit

6. Note, this keystore is highly insecure! If you want more security, you

should obtain a real SSL certificate:

http://tomcat.apache.org/tomcat-7.0-doc/ssl-howto.html

7. This keystore is not secured and should be in a more secure directory -- preferably

completely outside of the app for non-test applications -- and with strict permissions

on which user accounts can access it

## Instructions

First, clone this Git repository and import it into Eclipse as described

in the development environment setup guide

[https://class.coursera.org/mobilecloud-001/wiki/Installing\_Eclipse%2C\_Git%2C\_and\_Gradle].

This assignment tests your ability to create a web application that

allows clients to authenticate using the OAuth 2.0 Password Grant Flow.

Clients can upload video metadata (name, duration, etc.) once logged in,

as well as like/unlike videos.

The test that is used to grade your implementation is AutoGradingTest

in the org.magnum.dataup package in src/test/java. \*\*\_You should use the

source code in the AutoGradingTest as the ground truth for what the expected

behavior of your solution is\_.\*\* Your app should pass this test without

any errors. The test methods are annotated with @Rubric and specify

the number of points associated with each test, the purpose of the test,

and the videos relevant to the test.

The HTTP API that you must implement so that this test will pass is as

follows:

POST /oauth/token

- The access point for the OAuth 2.0 Password Grant flow.

- Clients should be able to submit a request with their username, password,

client ID, and client secret, encoded as described in the OAuth lecture

videos.

- The client ID for the Retrofit adapter is "mobile" with an empty password.

- There must be 2 users, whose usernames are "user0" and "admin". All passwords

should simply be "pass".

- Rather than implementing this from scratch, we suggest reusing the example

configuration from the OAuth 2.0 example in GitHub by copying these classes over:

https://github.com/juleswhite/mobilecloud-14/tree/master/examples/9-VideoServiceWithOauth2/src/main/java/org/magnum/mobilecloud/video/auth

You will need to @Import the OAuth2SecurityConfiguration into your Application or

other configuration class to enable OAuth 2.0. You will also need to remove one

of the containerCustomizer() methods in either OAuth2SecurityConfiguration or

Application (they are the exact same code). You may need to customize the users

in the OAuth2Config constructor or the security applied by the ResourceServer.configure(...)

method. You should determine what (if any) adaptations are needed by comparing this

and the test specification against the code in that class.

GET /video

- Returns the list of videos that have been added to the

server as JSON. The list of videos should be persisted

using Spring Data. The list of Video objects should be able

to be unmarshalled by the client into a Collection<Video>.

- The return content-type should be application/json, which

will be the default if you use @ResponseBody

POST /video

- The video metadata is provided as an application/json request

body. The JSON should generate a valid instance of the

Video class when deserialized by Spring's default

Jackson library.

- Returns the JSON representation of the Video object that

was stored along with any updates to that object made by the server.

- \*\*\_The server should store the Video in a Spring Data JPA repository.

If done properly, the repository should handle generating ID's.\_\*\*

- A video should not have any likes when it is initially created.

- You will need to add one or more annotations to the Video object

in order for it to be persisted with JPA.

GET /video/{id}

- Returns the video with the given id or 404 if the video is not found.

POST /video/{id}/like

- Allows a user to like a video. Returns 200 Ok on success, 404 if the

video is not found, or 400 if the user has already liked the video.

- The service should should keep track of which users have liked a video and

prevent a user from liking a video twice. A POJO Video object is provided for

you and you will need to annotate and/or add to it in order to make it persistable.

- A user is only allowed to like a video once. If a user tries to like a video

a second time, the operation should fail and return 400 Bad Request.

POST /video/{id}/unlike

- Allows a user to unlike a video that he/she previously liked. Returns 200 OK

on success, 404 if the video is not found, and a 400 if the user has not

previously liked the specified video.

GET /video/{id}/likedby

- Returns a list of the string usernames of the users that have liked the specified

video. If the video is not found, a 404 error should be generated.

GET /video/search/findByName?title={title}

- Returns a list of videos whose titles match the given parameter or an empty

list if none are found.

GET /video/search/findByDurationLessThan?duration={duration}

- Returns a list of videos whose durations are less than the given parameter or

an empty list if none are found.

This assignment also requires that you store your data using a Spring Data Jpa Repository.

This will make implementing the findByName and findByDurationLessThan functionality much

easier.

The AutoGradingTest should be used as the ultimate ground truth for what should be

implemented in the assignment. If there are any details in the description above

that conflict with the AutoGradingTest, use the details in the AutoGradingTest

as the correct behavior and report the discrepancy on the course forums. Further,

you should look at the AutoGradingTest to ensure that

you understand all of the requirements. It is perfectly OK to post on the forums and

ask what a specific section of the AutoGradingTest does. Do not, however, post any

code from your solution or potential solution.

There is a VideoSvcApi interface that is annotated with Retrofit annotations in order

to communicate with the video service that you will be creating. Your solution controller(s)

should not directly implement this interface in a "Java sense" (e.g., you should not have

YourSolution implements VideoSvcApi). Your solution should support the HTTP API that

is described by this interface, in the text above, and in the AutoGradingTest. In some

cases it may be possible to have the Controller and the client implement the interface.

Again -- the ultimate ground truth of how the assignment will be graded, is contained

in AutoGradingTest, which shows the specific tests that will be run to grade your

solution. You must implement everything that is required to make all of the tests in

this class pass. If a test case is not mentioned in this README file, you are still

responsible for it and will be graded on whether or not it passes. \_\_Make sure and read

the AutoGradingTest code and look at each test\_\_!

You should not modify any of the code in VideoSvcApi, AutoGrading, or AutoGradingTest.

## Testing Your Implementation

To test your solution, first run the application as described above. Once your application

is running, you can right-click on the AutoGradingTest->Run As->JUnit Test to launch the

test. Eclipse will report which tests pass or fail.

To get an estimated score for your solution, right-click on AutoGrading (not AutoGradingTest) and

Run As->Java Application. The AutoGrading application will run AutoGradingTest and then print a

summary of the test results and your score to the Eclipse Console (Window->Show View->Console).

The AutoGrading application will also zip all of your source code into a submission package that

you can submit to Coursera to receive your official grade. Note: each time that you run AutoGrading

it will create a separate zip file. Make sure that you choose the right zip file when submitting

your assignment! All of the submission zip files are placed in the coursera-submission folder.

## Submitting Your Assignment

To submit your assignment, you must first run the AutoGrading application as described in the previous

step to create your submission zip file. Make sure that you take note of the name of the submission

package that is printed in the console to ensure that you submit the correct zip file. You should

submit the submission package that is generated in the coursera-submission folder as the

"Output Submission". Leave the "Additional Submission" empty.

After submitting your solution to Coursera, your submission package will be sent to the auto-grading

servers. It may take a few minutes for a score to be assigned to your submission. Once the submission

is graded, a detailed score will be registered with Coursera.

Note: locally running the AutoGrading application DOES NOT submit your solution to Coursera and will

not be counted as a valid submission. The grade that you see when running the AutoGrading application

is an estimate of your grade only. You must correctly submit the solution to Coursera to receive an

official grade.

## Provided Code

- \_\_org.magnum.mobilecloud.video.repository.Video\_\_: This is a simple class to represent the metadata for a video.

You must annotate this object properly in order for it to be stored in the JPA repository. The annotations

that you may want to include are @Entity, @Id, @GeneratedValue, and @ElementCollection.

- \_\_OAuth 2.0 Configuration Code from the Examples\_\_: This code is available here:

https://github.com/juleswhite/mobilecloud-14/tree/master/examples/9-VideoServiceWithOauth2/src/main/java/org/magnum/mobilecloud/video/auth

You should ensure that you create the proper users and set the proper security on the various endpoints

to match the specification. We could have just included this code, but we wanted to give you the choice

of whether or not you used it. If you want to use it, please do so. If not, feel free to implement your

own approach that meets the assignment specification.

- \_\_SecuredRestBuilder\_\_: This wrapper around the Retrofit library is used by the tests to construct a client

that will automatically perform OAuth 2.0 authentication with a password grant before API methods are

invoked.

## Hints

- If you want to test your application without security (e.g., to add a simple request mapping

and try it without OAuth), you will need to uncomment the following lines in the build.gradle

file and then right-click on build.gradle->Refresh All:

```

compile("org.springframework.boot:spring-boot-starter-security:${springBootVersion}")

compile("org.springframework.security.oauth:spring-security-oauth2:2.0.0.RC2")

compile("org.springframework.security.oauth:spring-security-oauth2-javaconfig:1.0.0.M1")

```

- The examples in GitHub will be helpful on this assignment

- A valid solution is going to have at least one class annotated with @Controller

- There will probably need to be several different methods annotated with @RequestMapping to

implement the HTTP API described

- It is unlikely that you will be able to use Spring Data Rest to complete the assignment due to

differences in the responses provided by Spring Data Rest when adding new videos, etc.

- Any Controller method can take a Principal as a parameter to gain access/control over the

user who is currently authenticated. Spring will automatically fill in this parameter when your

Controller's method is invoked:

```java

...

@RequestMapping("/some/path/{id}")

public MyObject doSomething(

@PathVariable("id") String id,

Principal p) {

String username = p.getName();

// Maybe you want to add this users name to

// the list of people who like a video

....

}

```

- The IDs must be of type long. The tests send long values to the server and will generate

400 response codes if you use an int.

- If you get an error 400, you have incorrectly specified the parameter values that the method

should accept and their mapping to HTTP parameters.

- There are multiple ways to implement most pieces of the application. Any solution that passes

the tests will be given full credit.

- None of your Controllers or other classes should "implement VideoSvcApi" -- which is an interface

that is only used to create a Retrofit client. None of your classes should look like this:

```java

public class SomeClass implements VideoSvcApi // Don't implement this interface!

{

...

}

```

`