Streaming Manager API Workshop I

Author: George Owen

This page left intentionally blank

Document History

Revision History

|  |  |
| --- | --- |
| Date of this revision: 16th March 2017 | Date of next revision On Request |

|  |  |  |  |
| --- | --- | --- | --- |
| Revision Number | Revision Date | Summary of Changes | Changes marked |
| V0.01 | Mar 16, 2017 | Initial Version | (N) |
| V0.2.0 | May 17, 2017 | General updates |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[1. Introduction 6](#_Toc486432708)

[2. Streaming Manager API Lab 1 Setup 7](#_Toc486432709)

[3. Solution Code Overview 15](#_Toc486432710)

[3.1 HTML 15](#_Toc486432711)

[3.2 PHP 16](#_Toc486432712)

[3.3 JavaScript and JQuery 19](#_Toc486432713)

[3.4 Eclipse 19](#_Toc486432714)

[4. Lab 1 – Build Web Page with Default Channel Data 21](#_Toc486432715)

[4.1 Update Default Page 21](#_Toc486432716)

[4.2 Reset API Keys 21](#_Toc486432717)

[4.3 First Build Checkpoint 22](#_Toc486432718)

[4.4 Build First Tab 23](#_Toc486432719)

[4.4.1 API Login 23](#_Toc486432720)

[4.4.2 Extract account id 24](#_Toc486432721)

[4.5 Lab 1 Checkpoint 25](#_Toc486432722)

[4.6 Acquiring Content 25](#_Toc486432723)

[4.6.1 Retrieve Channels 26](#_Toc486432724)

[4.6.2 Retrieve Videos 26](#_Toc486432725)

[4.6.3 Build HTML page 27](#_Toc486432726)

[4.7 Lab 1 Checkpoint 29](#_Toc486432727)

[5. Lab 2 – Manage the Embedded Player 30](#_Toc486432728)

[5.1 Lab 2 Initial Code Review 30](#_Toc486432729)

[5.2 Player API Overview 32](#_Toc486432730)

[5.3 Defining HTML Structure 34](#_Toc486432731)

[5.3.1 Load Container Structure 34](#_Toc486432732)

[5.3.2 Player Tab JavaScript Review 35](#_Toc486432733)

[5.3.3 Update tab Management 36](#_Toc486432734)

[5.3.4 Defining the Player Tab Embed Container 37](#_Toc486432735)

[5.3.5 Defining the Title and Player Elements 39](#_Toc486432736)

[5.3.6 Defining the Status and Stats Elements 39](#_Toc486432737)

[5.3.7 Define Quality Element 40](#_Toc486432738)

[5.3.8 Define Player Controls 40](#_Toc486432739)

[5.4 Supporting JavaScript 41](#_Toc486432740)

[5.4.1 Loading the Player Object 41](#_Toc486432741)

[5.4.2 Trapping the Button Control Events 41](#_Toc486432742)

[5.4.3 Trapping Player Events 44](#_Toc486432743)

[5.4.4 Displaying and Changing the Player Quality 46](#_Toc486432744)

[5.5 Lab 2 Checkpoint 47](#_Toc486432745)

[5.6 Lab 2 Debugging 49](#_Toc486432746)

[6. Lab 3 Creating a Channel 51](#_Toc486432747)

[6.1 Updating the Home Page 51](#_Toc486432748)

[6.2 Add a Channel Create script 52](#_Toc486432749)

[6.3 Lab 3 Checkpoint 54](#_Toc486432750)

[7. Lab 4 – Upload Video Files to Channel 55](#_Toc486432751)

[7.1 Lab Setup 55](#_Toc486432752)

[7.2 File Upload Overview 57](#_Toc486432753)

[7.3 Updating the home page 59](#_Toc486432754)

[7.3.1 Display Status Message 60](#_Toc486432755)

[7.3.2 File Selection Fields 60](#_Toc486432756)

[7.3.3 Building the channel selection field 61](#_Toc486432757)

[7.3.4 Update the CSS Styling 61](#_Toc486432758)

[7.4 Building out the upload script 61](#_Toc486432759)

[7.4.1 Prepare for FTP connection Request 62](#_Toc486432760)

[7.4.2 Pick Up the Title 63](#_Toc486432761)

[7.4.3 Request an FTP Connection 63](#_Toc486432762)

[7.4.4 Make an FTP Request 63](#_Toc486432763)

[7.4.5 Set the video to a ready state 65](#_Toc486432764)

[7.4.6 Verify the uploaded video state 65](#_Toc486432765)

[7.5 Lab 4 Checkpoint 66](#_Toc486432766)

[8. Lab 5 – Change Account Access 67](#_Toc486432767)

[8.1 Authentication Tab Formatting 67](#_Toc486432768)

[8.2 Setup for OAuth Call 68](#_Toc486432769)

[8.3 OAuth Overview 70](#_Toc486432770)

[8.4 Build OAuth Redirect Structure 70](#_Toc486432771)

[8.4.1 Review Redirect Home Page 71](#_Toc486432772)

[8.4.2 Build redirect script 71](#_Toc486432773)

[8.4.3 Review Before Leaving 73](#_Toc486432774)

[8.5 How Does the Home Page Know? 73](#_Toc486432775)

[8.6 Lab 5 Checkpoint 74](#_Toc486432776)

[8.7 Lab 5 Debugging 76](#_Toc486432777)

[9. Tool-based API Calls 78](#_Toc486432778)

[9.1 Retrieve Auth Token 78](#_Toc486432779)

[9.1.1 Curl 78](#_Toc486432780)

[9.1.2 Chrome Poster 79](#_Toc486432781)

[9.1.3 Postman 81](#_Toc486432782)

[9.2 Retrieve User Data 82](#_Toc486432783)

[9.2.1 Curl 82](#_Toc486432784)

[9.2.2 Chrome Poster 83](#_Toc486432785)

[9.2.3 Postman 84](#_Toc486432786)

[9.3 Retrieve Channel Data 84](#_Toc486432787)

[9.3.1 Curl 84](#_Toc486432788)

[9.3.2 Chrome Poster 85](#_Toc486432789)

[9.3.3 Postman 86](#_Toc486432790)

[9.4 Retrieve Videos 87](#_Toc486432791)

[9.4.1 Curl 87](#_Toc486432792)

[9.4.2 Chrome Poster 87](#_Toc486432793)

[9.4.3 Postman 88](#_Toc486432794)

# Introduction

This Streaming Manager API lab is designed to provide basic techniques for using the Streaming Manager Player API and accessing the channel and video content of an account.

The 1st lab will demonstrate how to retrieve channel and video account information. The APIs involved are limited, although there is great latitude in how that information can be used.

The 2nd lab will add controls to manage the embedded Player and fields to display dynamic changes in the video status.

Lab 3 will provide the ability to upload videos to a channel.

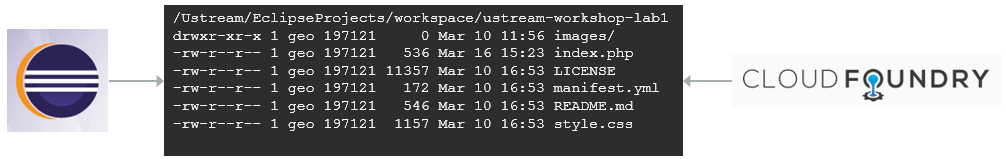
Lab 4 will demonstrate how to create a new channel in the account.

Finally lab 5 will show how to authenticate to a different account and show new channels and videos on the page.

# Streaming Manager API Lab 1 Setup

This first part of lab 1 will involve building the local environment to work with the Bluemix platform.

The Eclipse environment will be used to update the content of the lab application. There are no builds or exports necessary. The Cloud Foundry environment is used to update Bluemix with the current content of the lab application. To simplify development tasks, a common folder will be used by Eclipse and Cloud Foundry. This way changes from Eclipse are immediately ready to be pushed to Bluemix by Cloud Foundry.



Step 1: Create a local project folder on your host machine (the specific location and actual path name are not important).

Stan-Getz-iMac:~ workspace$ pwd (Mac)

/workshops/video/projects/workspace

C:\Development\StreamingManager\workspace> (Windows)

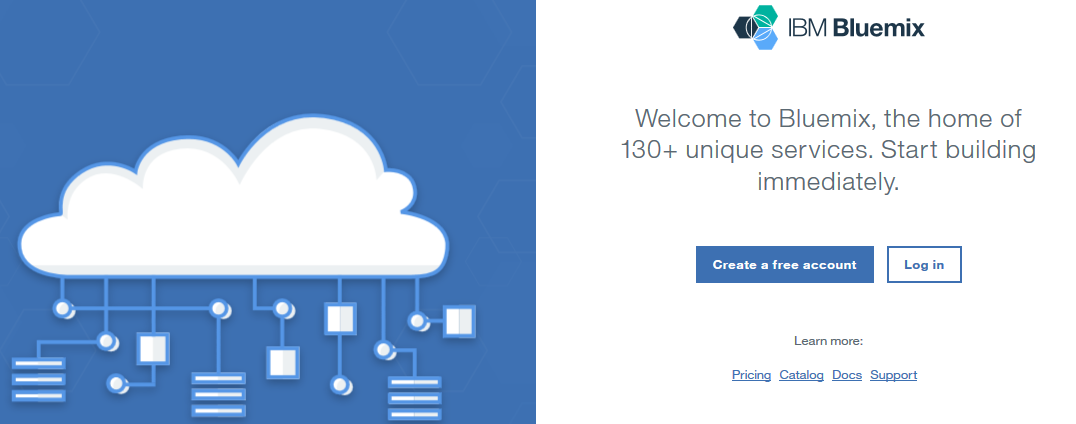
Step 2: Switch to the new folder and check for Cloud Foundry using your local command prompt.

C:\Development\StreamingManager\workspace> cf –v

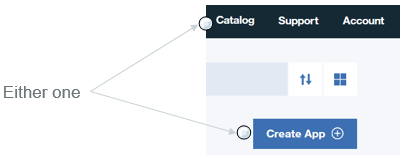
cf version 6.23.1+a70deb3.2017-01-13

Step 3: Create sample application in Bluemix

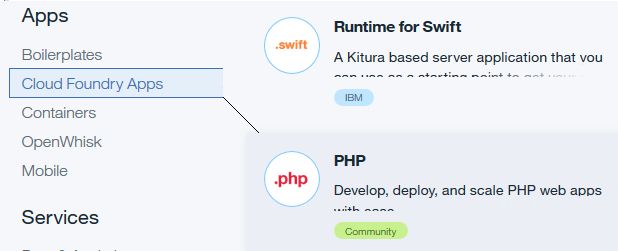
1. Login to your Bluemix account in your web browser



1. From your dashboard there are two ways to get to the Applications page



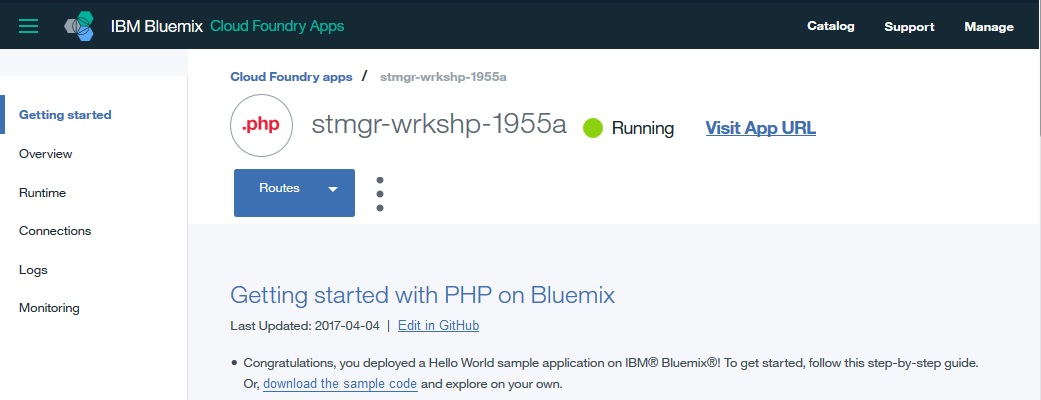
1. On the next page select “Cloud Foundry App” under Apps and then “PHP”



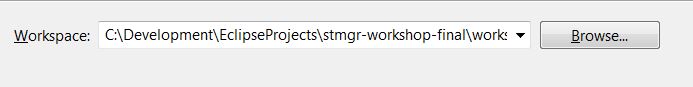
1. Enter an App name and Host name then click **Create**
   * These are usually the same but don’t have to be, the main restriction being that the hostname must be unique in the global Bluemix environment



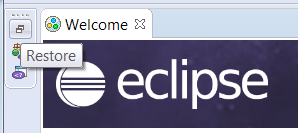
1. After the app is started the page will include a link to the sample code we will use as the base for the workshop modifications. Click the link to download the starter code package.



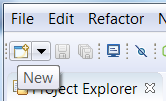
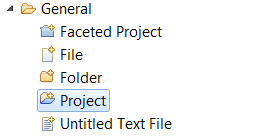
1. Open the Eclipse application
2. Browse for the workspace folder previously created



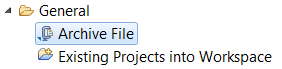
1. The first time in Eclipse you need to restore the project workspace



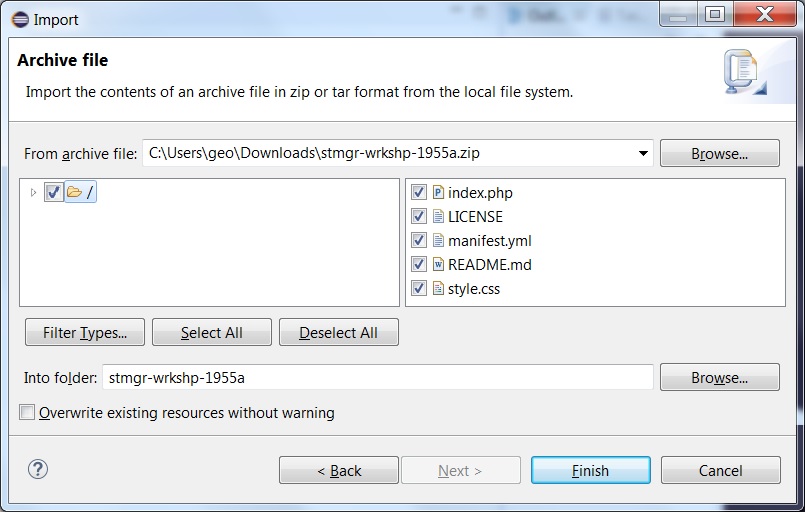
1. Now click the **New** icon and select **Project** under the **General** section, then **Next**

 🡺 

1. Enter a project name, preferably without spaces, and click **Finish**. Having it match the Bluemix application name would make sense but that is not required.
2. A new folder is now created under your workspace folder
3. Right-click on the project and click **Import**, then General, then **Archive File**, then **Next**



1. Browse for the sample code archive downloaded .zip file from Bluemix, then **Finish**

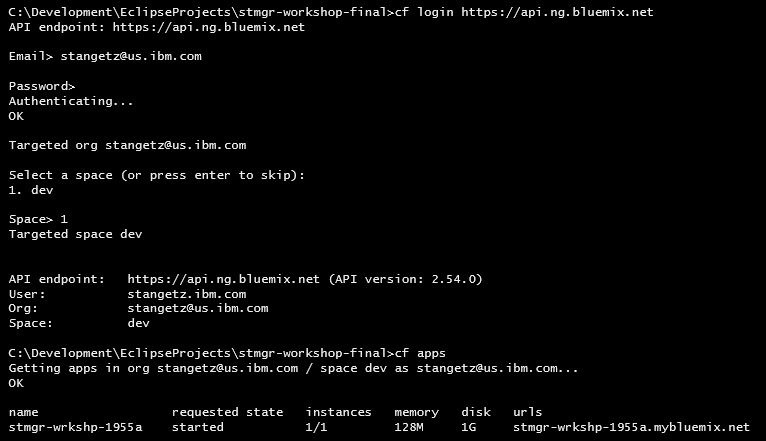


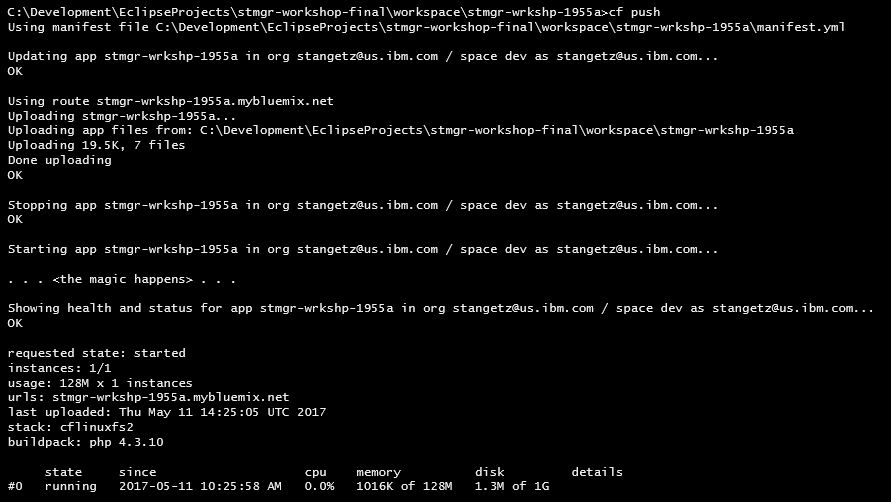
1. The Eclipse project code is now loaded into the Eclipse workspace and can be managed directly with Bluemix commands
2. In your host command window, switch to the project folder created in step 9, verify your Bluemix connection, then test the ability to push the application up to Bluemix with this command sequence

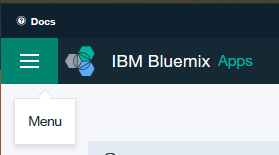
C:\Development\StreamingManager\workspace>cf login https://api.ng.bluemix.net

C:\Development\StreamingManager\workspace>cf apps

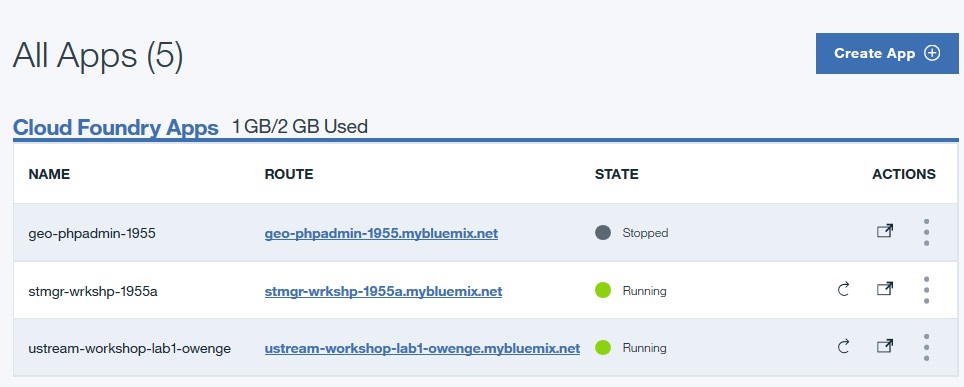
C:\Development\StreamingManager\workspace>cf push



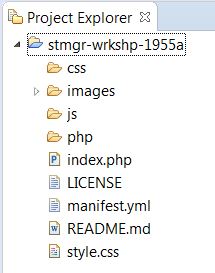


And on your Bluemix account Dashboard (from Menu at top left): 

Confirm your app is running.



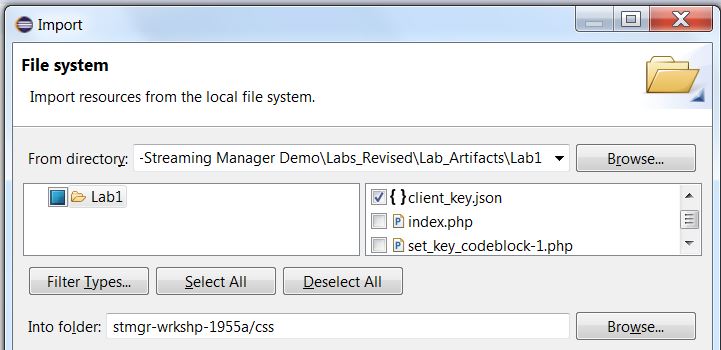
1. Create 3 new folders in the Eclipse project
   * Right-click the project folder name
   * Select **New**, then **Folder**
   * Name the 1st folder ‘css
   * Repeat and name the 2nd folder ‘js’ and the 3rd ‘php’



1. From the **Lab Artifacts/Lab 1** folder copy the following files into the corresponding folders:
   * Top-level
     1. client\_key.json
   * css:
     1. style.css
   * php:
     1. channel-video\_api.php
     2. set\_key.php
   * js:
     1. tab-manage.js

There are 2 ways to import the files.

* 1. Use the local command line, copy from **Lab Artifacts** to project folder, or
  2. Right-click the respective folder and select *Import*, then browse for the file



1. Delete the existing **style.css** file from the root project folder

# Solution Code Overview

This workshop will provide an understanding of how to use various APIs to access and manage Streaming Manager resources. Since the platform for using the APIs is a web browser, the solution will also require the use of HTML, CSS, PHP and JavaScript programming.

## HTML

The default home page from Bluemix contains a single HTML table with 1 row that includes a default graphic and a simple “Hello World” title. The workshop solution will replace that with 5 tabs, each demonstrating a single API function. The home tab displays the channel and video content of the default account.

After the headings a block is defined to contain all of the top-level tabs. A Javascript function is called to manage switching between tabs.

**HTML**

<div class=*"tab"*>

<button class=*"tablinks active"* onclick="openVideoApiFunction(event, 'access')">1. Accessing Content</button>

<button class=*"tablinks"* onclick="openVideoApiFunction(event, 'player')">2. Player</button>

<button class=*"tablinks"* onclick="openVideoApiFunction(event, 'channel')">3. Channel Create</button>

<button class=*"tablinks"* onclick="openVideoApiFunction(event, 'upload')">4. Video Upload</button>

<button class=*"tablinks"* onclick="openVideoApiFunction(event, 'auth')">5. Authentication</button>

</div>

Below the tabs a set of 5 blocks are defined to contain the content for each tab.

<!--content for Content tab-->

<div id=*"access"* class=*"tabcontent"* style="display: *block*;">

<h3>Accessing content</h3>

<p>You can access a user's channels and videos from the API. In the

API response, you will get all the related metadata like titles,

thumbnails, live status, length, etc.</p>

</div>

The Javascript function selectively exposes the content blocks for each tab and sets the active tab.

**JS**

// Hide all of the content blocks

**for** (i = 0; i < tabcontent.length; i++) {

tabcontent[i].style.display = "none";

}

// Turn off the active class for all tabs

**for** (i = 0; i < tablinks.length; i++) {

tablinks[i].className = tablinks[i].className.replace(" active", "");

}

// Make the current tab active and expose its content block

document.getElementById(apiFunction).style.display = "block";

evt.currentTarget.className += " active";

Style code is then used to change the tab color in response to user interaction.

**CSS**

/\* Style the buttons inside the tab \*/

**div***.tab* **button** {

background-color: *inherit*;

float: *left*;

. . .

}

/\* Change background color of buttons on hover \*/

**div***.tab* **button***:hover* {

background-color: *#ddd*;

}

/\* Change background color of buttons when tab is active \*/

**div***.tab* **button***.active* {

background-color: *#ccc*;

}

## PHP

PHP is used to build the page content from responses the code has received from the Streaming Manager APIs. PHP is server-side code, which has two implications. First, the code is not visible to the browser in the page source, and secondly there are challenges interacting with the page after it is initially loaded.

Like JavaScript, PHP code can placed inline or included from external files. The solution page will use both techniques. For the 1st lab the content block calls a script that uses pre-loaded client keys and populates the current block.

Load the client keys:

<body>

<div>

<?php

**include** 'php/set\_key.php';

?> <td>

<?php

For tab 1 pull in PHP script to generate content

<!--content for Content tab-->

<div id=*"access"* class=*"tabcontent"* style="display: *block*;">

<h3>Accessing content</h3>

<p>You can access a user's channels and videos from the API. … </p>

<?php

**include** 'php/channel-video\_api.php';

?>

</div>

--------------------------------------------

Contents of ***set\_key.php*:**

<?php

$myfile = fopen ( " client\_key.json", "r" ) or **die** ( "Unable to open file!" );

$apiParmFile = fread ( $myfile, filesize ( " client\_key.json" ) );

fclose ( $myfile );

$apiParms = json\_decode ( $apiParmFile, **true** );

$CLIENT\_KEY = $apiParms ['connKEY'];

$CLIENT\_SECRET = $apiParms ['connSECRET'];

?>

The inserted code builds the HTML page through the use of “**echo**” commands. Using variables generated from calls to the Streaming Manager API a complete HTML structure can be created. For example, the set of channel and video content is built using a combination of HTML text and API responses.

After calling the channel API this code starts a channel block and displays the channel id and title:

//start channel container

**echo** '<div class="channel-block">'."\n";

**echo** '<div><hr><p><span id="'.$channelId.'" class="channel">Channel: '.$channelTitle.'</span></p></div>';Builds this HTML structure:

This results in the following HTML:

<div class=*"channel-block"*>

<div><hr><p><span id=*"22725146"* class=*"channel"*>Channel: WW BP Live Demo</span></p></div>

After calling the video API the video entries can be retrieved and added:

**echo** "\n".'<div class="channel-video-block">';

**echo** "\n".'<img alt="" src="'.$videoThumb.'">';

**echo** "\n".'<div><a href="'.$videoUrl.'"><span id="'.$videoId.'" class="video" style="display: block;">'.$videoTitle.'</span></a></div>';

**echo** "\n</div>";

Which finishes the block as follows:

<img alt=*""* src=*"https://ustvstaticcdn1-a.akamaihd.net/i/video/picture/0/1/91/91405/91405344/1\_22725146\_91405344,192x108,b,1:2.jpg"*>

<div><a href=*"http://www.ustream.tv/recorded/91405344"*>

<span id=*"91405344"* class=*"video"* style="display: *block*;">playlist</span></a></div>

</div>

Four attributes are retrieved from the API for videos to build the HTML text:

1. Video thumbnail URL
2. URL to the video itself
3. A video id used by subsequent scripts to create selection lists
4. The title of the video

There are alternative methods for calling REST services in PHP. This workshop will use the built-in curl library to process REST calls to the Streaming Manager APIs. The login function provides a good example of building the HTTP request package and extracting the response.

$curl\_formData = array ('client\_id' => CLIENT\_KEY, 'grant\_type' => 'client\_credentials');

$curl = curl\_init();

curl\_setopt($curl, CURLOPT\_URL, "https://www.ustream.tv/oauth2/token");

curl\_setopt($curl, CURLOPT\_HTTPHEADER, array('Authorization: Basic ' . CLIENT\_SECRET));

curl\_setopt($curl, CURLOPT\_POSTFIELDS, $curl\_formData);

$result = curl\_exec($curl);

curl\_close($curl);

$resultJSON = json\_decode ($result, true);

## JavaScript and JQuery

JavaScript can also be used to call the Streaming Manager REST APIs and in its usual context of interacting with the web page. In these first labs JavaScript is used for two purposes. The 1st is to populate a selection field from the content lists that were generated by the PHP code. The 2nd is to provide control buttons that interact with the Streaming Manager Player and also display status and property data. Also, Streaming Manager provides a JavaScript library that contains the code used to connect to the embedded Player.

The 2nd set of code that creates the control buttons is implemented in a JQuery **$(document).ready()** code block. This ensures that the contained functions are ready as soon as the document is ready. In this lab that may not matter, but in a page that loads significant amounts of other data this could be necessary.

The use of the JQuery libraries is optional but it does provide some useful shortcuts. The workshop code will use the standard DOM objects with Javascript and will also use JQuery as alternative examples.

For example, the code that traps page events is written in JQuery:

**$('.video-button')**.click( /\* reference the class value \*/

Updating the visibility of status fields:

**$('#VideoDuration')**.val(''); /\* reference the tag’s id \*/

**$('.st-ended')**.hide(); /\* or class \*/

Standard manipulation of HTML DOM objects is also utilized

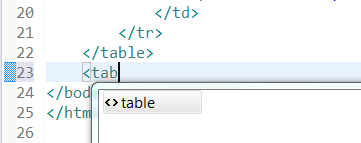
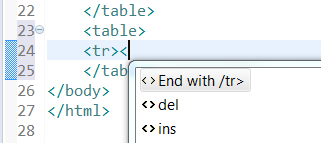
viewer.getProperty('progress', function(progress) {

var myProgress = timeFormat(progress);

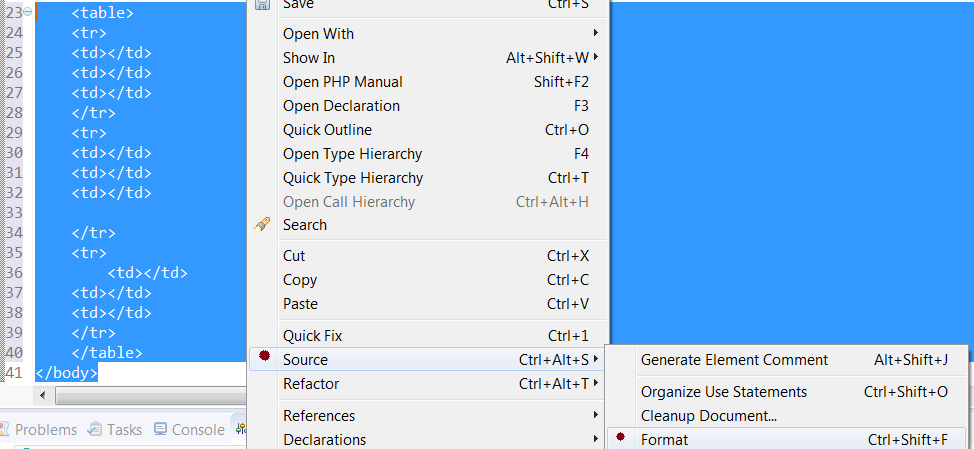
**document.getElementById("VideoProgress")**.textContent=myProgress;

## Eclipse

The editors in Eclipse provide code assists when entering various types of code. In HTML files it can offer choices for new tags and ending tags.

 🡺 

Eclipse will also reformat code according to the language standards, so indenting will automatically follow the normal rules and style guidelines.



Becomes



This is also valid for the PHP and JavaScript editors. Be aware though that while the indenting will generally be cleaned up the reformatting will sometimes make unexpected decisions and concatenate tags in illogical ways.

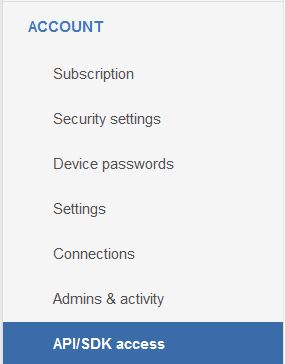
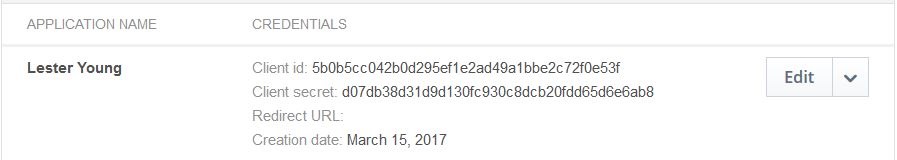
# Lab 1 – Build Web Page with Default Channel Data

## Update Default Page

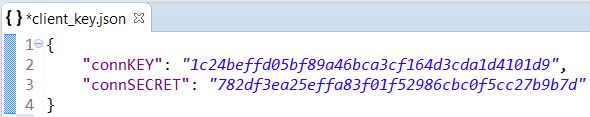
The 1st step is to replace the default home page with an updated workshop version. Delete the **index.php** file in the project and import the **Lab Artifacts/Lab1/index.php** version.

## Reset API Keys

The default keys file needs to be updated with your personal account values. Log onto your Streaming Manager account. At the bottom of the *Account* section is the *API/SDK access* option. That will display the current set of credentials. If no entry exists, click “Create Credentials”, then enter a name of your choosing. Otherwise edit the existing entry. Replace the values in the default project file with those in your account entry.

 Replace these values (Client id = connKEY, Client secret = connSECRET):

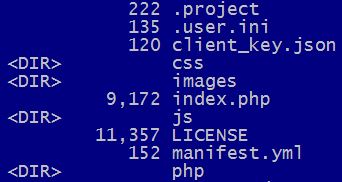


## First Build Checkpoint

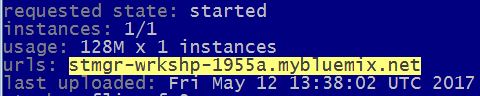
Now would be a good time to test the application before too many changes are made and the train goes unnoticed off the rails.

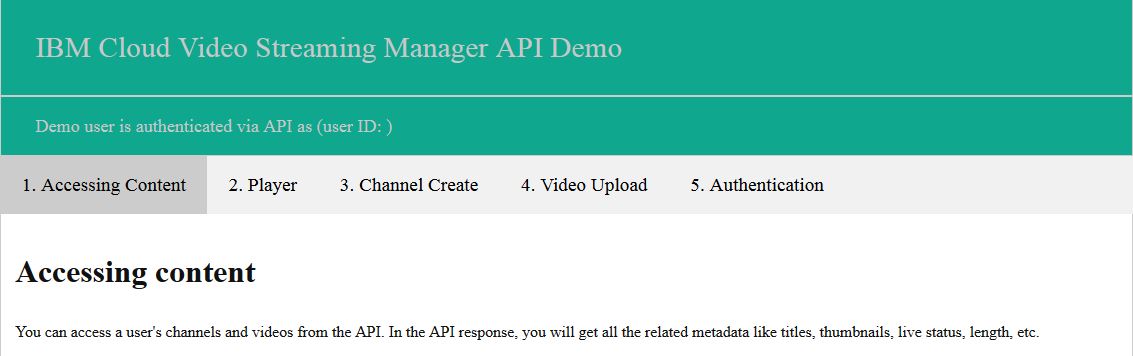


Switch to your local host command line and go to your workspace project folder. The contents should look like this:



Issue a **cf push** command to rebuild the application. Once it finishes, use the displayed application’s URL and open a browser window to the application. It should look like the following.





For troubleshooting there are two Cloud Foundry command to track the PHP server logs. From project the folder in the local host command window:

1. > cf logs <app name> this tails the running server log
2. > cf logs <app name> --recent displays the most recent log entries

## Build First Tab

The 1st step is to perform a login to the Streaming Manager site so that the channel and video content can be populated on the 1st tab.

Open the **php/set\_key.php** file in the Eclipse project. Notice that code is already in place to check for a session variable. This will be implemented in lab 5, but the initial structure has been created for you.

Scroll down to the first “placeholder” block.

### API Login

 The first operation creates an array of form parameters specifying the type of authentication to be performed. This will be a standard format for providing form parameters. The curl library will take care of converting the array to an appropriate request string. Replace “placeholder for code block 1” with this command.

|  |
| --- |
| $curl\_formData = **array** ('client\_id' => $CLIENT\_KEY, 'grant\_type' => 'client\_credentials' ); |

The next placeholder section sets up the rest of the HTTP request and makes the call.

Copy this code in the “placeholder for code block 2” location. The URL inserts a constant into the string for the sake of brevity. Here the authorization header dynamically creates the array, but it could have been predefined. The RETURNTRANSFER is enabled so that the access token will be returned to the script.



The result is decoded using an argument of **TRUE**. If **FALSE** was coded or no parameter was provided, then the values in the response would be accessed as “$auth\_token = $resultJSON->access\_token;”.

|  |
| --- |
| $curl = curl\_init ();  curl\_setopt($curl, CURLOPT\_URL, "https://".WWW\_URL."/oauth2/token");  curl\_setopt($curl, CURLOPT\_HTTPHEADER, **array**('Authorization: Basic ' . $CLIENT\_SECRET));  curl\_setopt($curl, CURLOPT\_POSTFIELDS, $curl\_formData);  curl\_setopt($curl, CURLOPT\_RETURNTRANSFER, 1);    $result = curl\_exec($curl);  curl\_close($curl);    // Convert result string to a JSON object  $resultJSON = json\_decode ($result, **TRUE**); |

The 3rd placeholder is also a short sequence.

 Here the retrieved authorization token is saved for subsequent use. Replace “placeholder for code block 3” with the following.

|  |
| --- |
| $\_SESSION["ustream\_token"] = $resultJSON ['access\_token'];  $auth\_token = $resultJSON ['access\_token']; |

At this point a valid access token should be present. Now issue a “cf push” in project’s local command window and refresh the browser page. If testing doesn’t show the expected result, place **echo** statements in the script to show the **$result** variable. *\*\* (See Debugging section in presentation for an example) \*\**

To help make subsequent request configuration less verbose, an authentication header can be saved here and used in later calls. .

 Go to the “placeholder for code block 4” location and substitute this command. This array will be used in the API requests with the current authorization token.

|  |
| --- |
| $API\_HEADER = **array**('Authorization: Bearer ' . $auth\_token, 'content-type: application/x-www-form-urlencoded'); |

Now that an authorization token is available, a call can be made to extract the current account’s email address and id for display in the headers.

### Extract account id

 Scroll to “placeholder for code block 5” and enter this request sequence. Notice that this request uses the API URL and the authorization header defined in the last step. The HTTP status is retrieved here but not checked. As another debugging tool, the status value can be **echo’d** to the browser in the event of unexpected results.

|  |
| --- |
| // Retrieve account information to display user's id and email address  $curl = curl\_init ();  curl\_setopt ( $curl, CURLOPT\_URL, "https://" . API\_URL . "/users/self.json" );  curl\_setopt ( $curl, CURLOPT\_HTTPHEADER, $API\_HEADER );  curl\_setopt ( $curl, CURLOPT\_RETURNTRANSFER, 1);  $result = curl\_exec ( $curl );  $response = json\_decode($result, **TRUE**);  $status = curl\_getinfo ( $curl, CURLINFO\_HTTP\_CODE );    curl\_close ( $curl ); |

 After the request finished the target values can be retrieved for displaying on the page at “placeholder for code block 6”.

|  |
| --- |
| $CLIENT\_ADDRESS = $response ['email'];  $CLIENT\_ID = $response ['id']; |

As noted before, variables defined in scripts can be referenced in later PHP scripts on the page. Thus, a later PHP code snippet can use these variables to build the page content.

<span>Demo user is authenticated via API as <?php **echo** "$CLIENT\_ADDRESS (user ID: $CLIENT\_ID)"; ?> </span>

## Lab 1 Checkpoint

Now would be a good time to check your progress. Save any edited pages in Eclipse, then go to the project’s command line folder and issue the **cf push** command. Once that finishes refresh the browser page and verify that your email address and id are displayed in the header section

.

Note: See **Lab Solutions/Lab1/set\_key.php**

## Acquiring Content

Now the code to acquire the channels and videos and build out the tab 1 content will be completed. In Eclipse open the file **php/channel-video\_api.php**.

There are five placeholders for code that will be inserted. There are two functions that will first be implemented, one to retrieve the channel data and one to retrieve the video data. After those functions are defined code will be added to call those functions and format the data on the page.

### Retrieve Channels

 Scroll down to “placeholder for code block 1” in the *getChannels*() function. Insert the following code to retrieve channel data from the account. Notice the URL now referencing the API URL with the channels URI. Also notice the use of the global variable that was defined with the authorization header.

|  |
| --- |
| $curl = curl\_init();  curl\_setopt($curl, CURLOPT\_URL, "https://".API\_URL."/users/self/channels.json");  curl\_setopt($curl, CURLOPT\_HTTPHEADER, $GLOBALS['API\_HEADER']);  curl\_setopt($curl, CURLOPT\_RETURNTRANSFER, **true**);    $result = curl\_exec($curl);  $status = curl\_getinfo($curl, CURLINFO\_HTTP\_CODE);  curl\_close($curl);    **if** ($status == 200) {  **return** json\_decode($result);  } |

If the function failed to return any results the web page will be empty, which is a clear indication of a problem. An alternative would be to echo the result string and status code in this routine so it would be visible on the page.

### Retrieve Videos

Next go to the *getVideos($channel\_id)* function. Here the channel id is provided as input.

 Enter this code for “placeholder for code block 2”. For the request there is a *filter* query parameter. It can be used to determine which videos in the channel should be returned. The parameter is defined in an array and then converted to a valid string by the curl *http\_build\_query()* function. Notice also the new URL that specifies the channel id and the generated query parameters.

|  |
| --- |
| $curl = *curl\_init*();    $fields = **array**('filter' => **array**('protect' => 'private,public'));  $field\_query = *http\_build\_query*($fields);    *curl\_setopt*($curl, *CURLOPT\_URL*, "https://".*API\_URL*."/channels/$channel\_id/videos.json/?$field\_query");  *curl\_setopt*($curl, *CURLOPT\_HTTPHEADER*, **$GLOBALS**['API\_HEADER']);  *curl\_setopt*($curl, *CURLOPT\_RETURNTRANSFER*, **true**);    $result = *curl\_exec*($curl);    $status = *curl\_getinfo*($curl, *CURLINFO\_HTTP\_CODE*);  *curl\_close*($curl); |

The basic parts of the two functions are now complete so code can be added to start calling them. Scroll back up to placeholder 3 in the **foreach** loop. Here the two routines can be executed to build out the page.

The result from the API call returns a channel structure like this:

{

"channels": {

"22953470": {

"id": "22953470",

...

The **foreach** syntax drills down to the array of objects in the “channels” entry and pulls out the key value (1) of each array entry. The value for each is itself an array of key-value pairs (2).

### Build HTML page

 The 1st step is to use the loop key to collect the current block of channel properties. From that block the channel title and id can be extracted. Insert this code into the “placeholder for code block 3” location.

|  |
| --- |
| // load the key to this channel object  $channel = $channel\_objects->channels->{$key};    // extract the id and title  $channelId = $channel->id;  $channelTitle = $channel->title; |

Now we are ready to start building out the web page. The format of the page is derived from the HTML code on the page plus the CSS styling applied to that code.

 In the “placeholder for code block 4” location enter this code to start a new channel block on the page. The 1st step is to open a new <DIV> block container for this channel. It will show the channel title and contain a set of video blocks. The container is assigned a CSS class of “channel-block”. Then a 2nd <DIV> block is generated only to hold the channel data. That data is included in a <SPAN> element with an id = to the channel id. This id can be used by subsequent Javascript to locate all of the channel elements on the page.

|  |
| --- |
| //start channel container  **echo** '<div class="channel-block">'."\n";  **echo** '<div><hr><p><span id="'.$channelId.'" class="channel">Channel: '.$channelTitle.'</span></p></div>';  getVideos($channelId);    **echo** "\n</div>"; /\* end of channel block \*/ |

After the channel data is built the video function can be called to display all of the videos for this channel. Then the channel container can be closed. To implement the rest of the *getVideos()* function scroll down to “placeholder for code block 5”.

 Insert this code for “placeholder for code block 5”. The result object from the API is a simple array of objects so no keys are needed to access them. The required attributes are extracted from each video block. A new <DIV> block contains all of the data for a single video entry. After the thumbnail image element is added a new <DIV> block is added to contain the hyperlink. That hyperlink contains the text in a <SPAN> element with an id. Like the channel elements, that id will be used by Javascript to find all of the video elements.

|  |
| --- |
| $response = json\_decode($result);    **foreach**($response->videos **as** $videoEntry) {  $videoTitle = str\_replace("\_", " ", $videoEntry->title);  $videoId = $videoEntry->id;  $videoUrl = $videoEntry->url;  $videoThumb = $videoEntry->thumbnail->default;  **if** ($videoEntry->protect != "private") {  **echo** "\n".'<div class="channel-video-block">';  **echo** "\n".'<img alt="" src="'.$videoThumb.'">';  **echo** "\n".'<div><a href="'.$videoUrl.'"><span id="'.$videoId.'" class="video" >'.$videoTitle.'</span></a></div>';  **echo** "\n</div>";  }  } |

The CSS class on the <DIV> container is how the video blocks are stacked horizontally and vertically. The **float** specification keeps the blocks left-justified up to the width of the containing <DIV>, and the dimension and margin specifications set the size and inter-block spacing relationships.

**div***.channel-video-block* {

float: *left*;

width: *190px*;

height: *190px*;

border: *1px* *solid* *#ccc*;

font-size: *12px*;

padding-left: *2px*;

padding-right: *2px*;

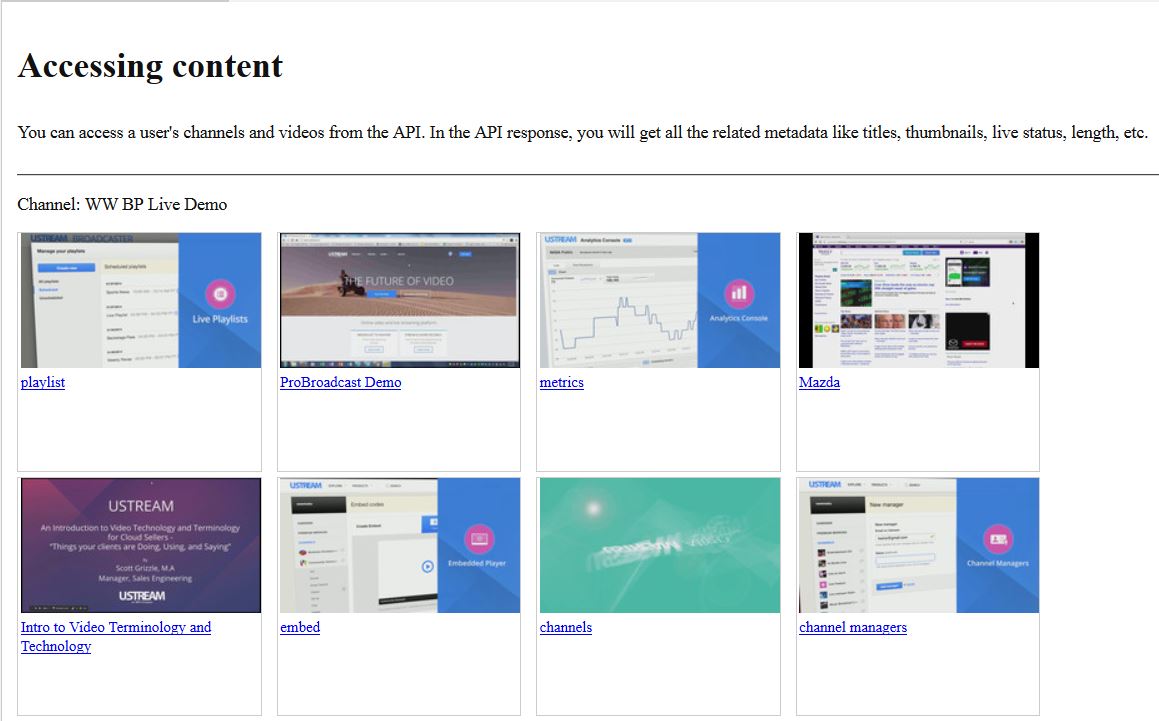
margin-right: *12px*;

margin-bottom: *4px*;

}

## Lab 1 Checkpoint

Go back to the command line to reload the application. Now when the browser page is refreshed there should be channel and video data on the 1st tab. In addition to verifying the page layout, also verify the functioning of the links. If any part does not match the solution, use the debugging techniques previously discussed to research possible causes.



End of Lab 1

# Lab 2 – Manage the Embedded Player

The first lab loaded the channel and video content for the default account. This lab will create the code behind the control buttons for the embedded player that will allow the user to:

1. Load new channel or video content from generated selection lists into the player
2. Play and pause the loaded content
3. Seek to a specific time position into recorded content
4. Display the duration and current viewing position

In Eclipse, import files **Lab Artifacts/Lab2/player-controls.js** and **ustream-embedapi.js** into the **js/** project folder and then open **player-controls.js**. This file contains code that will build selection lists from the channel and video lists of content. These selection fields will be used when content is loaded into the player or video files are loaded into a channel.



Let’s review the JavaScript code that builds the selection fields. This is complete code, so no modification will be needed.

## Lab 2 Initial Code Review

The script is executed when the HTML document is loaded. It depends on the server code from Lab 1 to have generated the elements for the channels and videos that are included in the account.

The first section of the script will first get sets of all of the “channel” and “video” elements, identified by a corresponding CSS class attribute.

|  |
| --- |
| **var** channels = document.getElementsByClassName('channel');  **var** videos = document.getElementsByClassName('video'); |

Then objects need to be created for the two selection fields and corresponding buttons.

|  |
| --- |
| // create targets for both select fields  **var** channelField = document.getElementById("load-channel");  **var** videoField = document.getElementById("load-video");  // create targets for both buttons  **var** channelButton = document.getElementById('channel-button');  **var** videoButton = document.getElementById('video-button'); |

The player will have a channel preloaded.

|  |
| --- |
| // load the first channel found as the default player content  **var** player = document.getElementById("UstreamIframe");  player.src = "https://www.ustream.tv/embed/" + channels[0].id + "?html5ui"; |

Any existing selection content is removed (only channel shown).

|  |
| --- |
| // clear the existing channel selection options  **for** (i = channelField.options.length - 1; i >= 0; i--) {  channelField.remove(i);  } |

A default selection is created and added to both selection lists. Then the set of discovered elements is loaded into the selection list (only channel shown).

|  |
| --- |
| // Create default entry  **var** newOptC = document.createElement('option');  newOptC.value = "";  newOptC.innerText = "Select a channel";  **var** newOptV = document.createElement('option');  newOptV.value = "";  newOptV.innerText = "Select a video";  // Prepopulate the Player channel and video selections  channelField.add(newOptC);  videoField.add(newOptV);  **var** newOpt;  // Create a selection option for each content entry  **for** (i = 0; i < channels.length; i++) {  newOpt = document.createElement('option');  newOpt.value = channels[i].id;  newOpt.innerText = channels[i].innerText;  channelField.add(newOpt);  } |

 This lab will build out the rest of the code in the script that will implement the various player controls. To start, add these entries inside the <head> </head> tag in **index.php**.

|  |
| --- |
| **<script type=*"text/javascript"* src=*js/player-controls.js*></script>**  **<script type=*"text/javascript"* src=*js/ustream-embedapi.js*></script>** |

## Player API Overview

The Streaming Manager Player API is documented here: <http://developers.ustream.tv/player-api/player.html>. There are 4 parts to the API. The entries in blue will be covered in this lab.

1. The embedded Player object runtime options:
   1. allowfullscreen
   2. autoplay
   3. controls
   4. offaircontent
   5. quality
   6. showtitle
   7. volume
2. Functional methods:
   1. play
   2. pause
   3. stop
   4. load
   5. seek
   6. volume
   7. quality
3. Properties:
   1. duration
   2. progress
   3. viewers
   4. content
   5. playing content
4. Listener Functions (Add and Remove):
   1. live
   2. offline
   3. finished
   4. playing
   5. size
   6. quality
   7. synchedmeta
   8. content

The Player Object

The player object is embedded in an HTML iframe tag. The default setup used in the lab is as shown.

<iframe id=*"UstreamIframe"* src=*"https://www.ustream.tv/embed/22752241?html5ui"*

style="border: *0 none transparent*;" webkitallowfullscreen

allowfullscreen width=*"360"* height=*"220"*></iframe>

Additional options could be added as required:

allowfullscreen autoplay=*"false"* width=*"360"* height=*"220"*></iframe>

Which options to implement would depend on how the player is to be utilized. For example, since the lab will provide player controls that are part of the page design, it makes sense to accept the default **controls** option to hide the built-in player controls.

Player Methods

The call to the player methods comes in two versions. Some only need the method command while others require input arguments. See the documentation page listed above for details.

The simple methods:

viewer.callMethod(<method operation>): play, pause and stop

Methods with arguments:

viewer.callMethod(<method operation>, <method arguments>): load, seek, volume and quality

Property Methods

The getProperty method is called with an anonymous function as the argument after the property name. That function will have the property object returned as the function parameter.

The simple properties:

viewer.getProperty(<property name>, function(<function name>) {}): **duration**, **progress** and **viewers**

Example:

viewer.getProperty(<property name>, function(“duration”) {

control.log(“duration = ”, duration);

})

The properties with complex results:

viewer.getProperty(<property name>, function(<function name>) {}): **content**, **playingContent**,

Example:

viewer.getProperty(<property name>, function(“content”) {

control.log(“content type = ”, content[0]);

control.log(“content id = ”, content[1]);

})

Listener Functions

Various dynamic events are triggered by the player. For each that needs processing a listener function is defined. The format is the same for all, the only difference is in the data package that is sent with the event.

To define a listener function two parameters are required.

viewer.addListener(<event>, <function name>);

var <function name> = function(event, data) {

console.log(“event ‘“ + event + “’ has data: “, data); // debugging

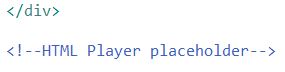
}

## Defining HTML Structure

Before the script can be loaded the supporting HTML structure needs to be built.

### Load Container Structure

Open the **index.php** web page file. Currently there is a title and two radio buttons in the Player content tab. Scroll down to the Player tab placeholder.



The Player tab hides the embedded player and controls until a selection is made. Activating a radio button will expose a corresponding selection field and load button. The radio button fields call a Javascript function to manage the display of the other elements. Import file **Lab Artifacts/Lab2/manage-player-selects.js** into the **/js** project folder. This file contains a number of functions that are used to change the appearance of the Player tab.

 Add this entry inside the <head> </head> tag in **index.php** along with the other script includes.

|  |
| --- |
| **<script type=*"text/javascript"* src=*js/manage-player-selects.js*></script>** |

 Replace “HTML Player placeholder” with the following HTML code. It defines the two selection fields and load buttons.

|  |
| --- |
| <div>  <select id=*load-channel* onchange=*"sourceSelect(this)"*>  <option>Select a channel</option>  </select>  <select id=*load-video* onchange=*"sourceSelect(this)"*>  <option>Select a video</option>  </select>  <div id=*"video-button"*>  <button type=*"button"* class=*"video-button loader control-load-video"*>Load Video</button>  </div>  <div id=*"channel-button"*>  <button type=*"button"* class=*"video-button loader control-load-channel"*>Load Channel</button>  </div>  </div>  <!--end of Player select buttons--> |

Remember that you can reformat the source to standardize indentation (although it’s not perfect and some cleanup will still be needed):

1. Select all the code (Ctrl+A or Edit->Select All)
2. Right-click the selected text
3. **Source** -> **format**

### Player Tab JavaScript Review

Two JavaScript functions are now in effect. When the default radio buttons are activated the related onclick() function will expose the appropriate selection fields.

|  |
| --- |
| **function** sourceRadio(rad)  {  **var** type = rad.value;    channelSelect = document.getElementById('load-channel');  videoSelect = document.getElementById('load-video');  // now hide or show the other elements  **if** (type == "channel") {  channelSelect.style.display = "inline";  videoSelect.style.display = "none";  } **else** {  channelSelect.style.display = "none";  videoSelect.style.display = "inline";  }  . . . |

Similarly, when a selection is then made of either a channel or a video the corresponding load button will be exposed by the field’s *onchange()* function.

|  |
| --- |
| **function** sourceSelect(sel) {    **var** whichButton = sel.id;  /\*  \* Player tab  \*/  **var** channelButton = document.getElementById('channel-button');  **var** videoButton = document.getElementById('video-button');    **if** (whichButton == "load-channel") {  channelButton.style.display = "block";  videoButton.style.display = "none";  **return**;  }  **if** (whichButton == "load-video") {  channelButton.style.display = "none";  videoButton.style.display = "block";  **return**;  }  . . . |

As with the selection fields, the corresponding button will be displayed while the other button will be hidden. This function also has code for the Upload Video tab which will be reviewed in a later section.

### Update tab Management

Now the script that manages behavior on tab switches can be updated. The new selection fields and buttons on the Player tab should be hidden by default whenever the tab is opened. Open the **js/tab-manage.js**.

 The current code in **js/*tab-manage.js*** only manages the top-level container block for each tab so they don’t all display concurrently. The Player tab has objects that will be displayed in sequence, so code must be added to hide those individual fields. Insert this code in place of the “tab-manage placeholder 1 for Player” comment..

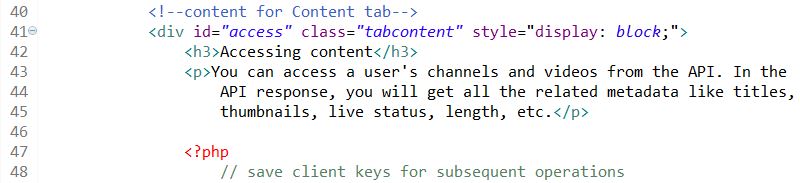
|  |
| --- |
| /\*  \* Player tab  \*/  **if** (apiFunction == "player") {  // Reset the radio buttons and selection fields  document.getElementById('radio-channel').checked = **true**;  document.getElementById('load-channel').style.display = "inline";  document.getElementById('load-video').style.display = "none";    // Hide the load buttons and the player block  document.getElementById('channel-button').style.display = "none";  document.getElementById('video-button').style.display = "none";  document.getElementById('player-block').style.display = "none";  } |

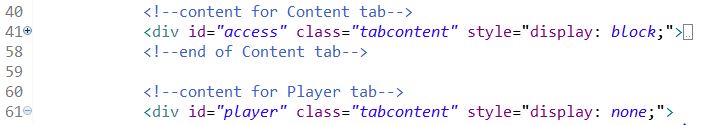
As you can see, the radio button is set to “channel” by default and the channel selection field is displayed. The other elements previously defined will be hidden until referenced by interaction with the user.

### Defining the Player Tab Embed Container

The next step is to create the HTML that will contain the player and associated controls. Go back to the **index.php** file. The next steps will create a container for the player object on the page and the associated controls.

***Tip***: you can collapse blocks of code in Eclipse to make it easier to work with a particular section.





 Insert this HTML between the end of the button container and the end of the tab content.

|  |
| --- |
| </div> \* only for position reference  <!--end of Player select buttons--> \* only for position reference  <div id=*"player-block"*>  <div class=*"player-container"*>  <div class=*"player-title"*>  </div>  <div class=*"player-embed-left"*>  </div>  <!--end of player-embed-left block-->    <div class=*"player-embed-right"*>  <div id=*"player-status"*>  </div>  <div id=*"player-stats"*>  </div>  <div class=*"player-quality"*>  </div>  </div>  <!--end of player-embed-right block-->    </div>  <!--end of embed player-container block-->    <div class=*"player-container"*>  </div>  <!--end of controls player-container block-->    </div>  <!--end of player-block block-->  </div> \* only for position reference  <!--end of Player tab--> \* only for position reference |

This HTML code creates the HTML structure that will contain the player object and controls.



### Defining the Title and Player Elements

Fill in the *"player-title"* and *"player-embed-left"* containers with these code blocks.



**Title**

|  |
| --- |
| <hr>  <span id=*"player-title"*>Some title of a video</span> |

**Player**

|  |
| --- |
| <iframe id=*"UstreamIframe"* width=*"360"* height=*"220"*  allowfullscreen=*""* webkitallowfullscreen=*""*  style="border: *0 none transparent*;" src=*""*></iframe> |

The title element can be anything at this point as it will be replaced when content is loaded into the player.

### Defining the Status and Stats Elements

The status container contains various elements that display the current status of the content, such as “OFFLINE” or “PLAYING” or “LIVE”. The elements will be selectively exposed in response to events initiated in the player. Initially they will all be hidden.

 Insert this code into the *"player-status"* container.

|  |
| --- |
| <span style="display: *none*;" class=*"st-offline label"*>OFFLINE</span>  <span style="display: *none*;" class=*"st-live label"*>LIVE</span>  <span style="display: *none*;" class=*"st-playing label"*>PLAYING</span>  <span style="display: *none*;" class=*"st-ended label"*>ENDED</span> |

The next container contains the fields for showing the duration of the content and current progress, plus a quality field.

 Insert this code into the *"player-stats"* container.

|  |
| --- |
| <div class=*"spacer"*></div>  <div><span id=*"VideoDurationTitle"* class=*"stats"*>Duration</span></div>  <div><span id=*"VideoDuration"* class=*"stats"*></span></div>  <div class=*"spacer"*></div>  <div><span id=*"VideoProgressTitle"* class=*"stats"*>Progress</span></div>  <div class=*"player-status"*><span id=*"VideoProgress"* class=*"stats"*></span>  </div> |

### Define Quality Element

The last element in the right-hand block is the quality element. It will show a dynamically-generated selection list of video quality options and the current quality setting

 Insert this code into the *"player-quality"* container.

|  |
| --- |
| <div class=*"spacer"*></div>  <label>Video quality</label>  <div class=*"control-select"*>  <select class=*"quality-selector"*></select>  <div class=*"select"*><span class=*"label"*>Video Quality</span></div>  </div> |

### Define Player Controls

The last section contains the controls that Play, Pause and Seek the player.

 Insert this code into the 2nd *"player-container"* block.

|  |
| --- |
| <div class=*"player-control"*>  <button type=*"button"* class=*"video-button control-play"*>Play</button>  </div>  <div class=*"player-control"*>  <button type=*"button"* class=*"video-button control-pause"*>Pause</button>  </div>  <div class=*"player-control"*>  <input class=*"player-control-input"* type=*"text"* id=*"Seek"*  maxlength=*"4"* />  </div>  <div class=*"player-control"*>  <button type=*"button"* class=*"video-button control-seek"*>Seek</button>  </div> |

If you are in doubt about the result, refer to **Lab Solutions/Lab2/index.php**.

## Supporting JavaScript

### Loading the Player Object

 This 1st part is one simple command. Open the *player-controls.js* file in Eclipse if it is not already open. Scroll down to “player-controls placeholder 1”. Insert this code in its place.

|  |
| --- |
| // load Streaming Manager viewer object for the Player  viewer = UstreamEmbed("UstreamIframe"); |

The UstreamEmbed function is defined in the file that was downloaded from GitHub. The value is the id of the HTML iframe element that contains the player. This object will provide all of the methods and properties to manage the player’s operation.

### Trapping the Button Control Events

The function template has already been populated in the file.

|  |
| --- |
| /\* set up function to trap click events \*/  $(".video-button").click(  **function**(e) { |

This is JQuery syntax that will capture the click events of any element with the class “video-button”, as in:

<button type="button" class="**video-button** control-play">Play</button>

The next set of commands gets the element that issued the event and attempts to pull out another class name that identifies the specific Streaming Manager operation.

|  |
| --- |
| **var** elementTarget = $(e.currentTarget);  **var** control = elementTarget[0].className.match(/control-([a-zA-Z0-9\-]\*)/);  **if** (control && control[1]) { |

The **Play** button itself would be identified using the 2nd class in the CSS specification.

<button type="button" class="video-button **control-play**">Play</button>

The regular expression will result in control = “control-play” and control[1] = “play”.

The **Play** and **Pause** buttons are straightforward. The player object is simply called with the appropriate option. Note that a **break**; command should terminate each case block.

Scroll down to “player-controls placeholder 2”.

 Insert the 1st two traps for **Play** and **Pause**.

|  |
| --- |
| **case** "play":  viewer.callMethod('play');  **break**;    **case** "pause":  viewer.callMethod('pause');  **break**; |

The **Seek** operation needs to know the position to seek. Using the page element’s **id** attribute, insert a new **case** block, loading the input field’s value and then calling the player object.



HTML source: <input class=*"player-control-input"* type=*"text"* **id=*"Seek"***maxlength=*"4"* />

|  |
| --- |
| **case** "seek":  seekField = document.getElementById("Seek");  position = seekField.value;  viewer.callMethod('seek', position);  **break**; |

The **Load** operation is a bit more complicated, as it needs to know the type of content, **channel** or **video**, and the specific id of the content. Plus, in this application, different HTML elements will be exposed depending on the choice made. Some of the code can be shared by the two options, so a common case block can be defined.

 Insert a new **case** block for “**load-channel”** and “**load-video”**.

|  |
| --- |
| **case** "load-channel":  **case** "load-video": |

Since there is no “**break**;” command the load-channel case will drop down into load-video.

 The 1st step will be to call a function that will reset the various fields that display status and runtime properties of the current player content. Insert this command as the first instruction in the “**load-video” case**.

|  |
| --- |
| clearFields(); |

That function is already defined in the script file. It resets the display attribute of various status elements.

|  |
| --- |
| **var** clearFields = **function**() {  populateQualitySelector({});  $('#VideoProgress').val('');  $('#VideoDuration').val('');  $('.st-ended').hide();  $('.st-playing').hide();  $('.st-offline').hide();  $('.st-live').hide(); |

The 1st command calls a separate function to clear the quality element, as it has a selection field and display field. The 1st two elements are referenced by JQuery syntax only for the purposes of demonstration, the others use standard JavaScript.

The next step is to retrieve the respective selection list choice. The assignment of ids and class names helps tie the various elements together.

 Insert this code after the clearFields() function call.

|  |
| --- |
| **var** resourceField = document.getElementById(control[1]);  **var** resourceId = resourceField.options[resourceField.selectedIndex].value;  **var** resourceTitle = resourceField.options[resourceField.selectedIndex].text; |

Recall that “control[1]” was extracted from the class of the button element and will be either “**load-channel**” or “**load-video**”. The corresponding select field looks like this with a matching id:

<select **id=*load-video*** onchange=*"sourceSelect(this)"*>

So the first command can use the control variable as the identifier for finding the selection field.

 The next commands load the title of the content into a display field and then the selected content into the player.

|  |
| --- |
| document.getElementById("player-title").textContent = resourceTitle;  **var** contentType = control[1].substring(5);  viewer.callMethod('load', contentType, resourceId); |

If the “control[1]” value was “load-**channel**” or “**load-video**” then the substring value for content type will be either “**channel**” or “**video**”. The resource id is extracted from the selection list entry. At this point the player would be populated and running, but the controls are not yet visible.

 Finish the load **case** block with this code.

|  |
| --- |
| // Expose controls based on content type  **if** (contentType == "channel") {  document.getElementById('player-status').style.display = "block";  document.getElementById('player-stats').style.display = "none";  } **else** {  document.getElementById('player-status').style.display = "none";  document.getElementById('player-stats').style.display = "block";  // Set up the recurring progress indicator  **var** refreshId = setInterval(progress\_monitor, 10000);  }    // Now expose the player block  document.getElementById('player-block').style.display = "block"; |

Note the call to setInterval(progress\_monitor, 10000); when a video is loaded. This application will update the progress every 10 seconds. It is also possible tie the update to a button if you wanted to retrieve it dynamically.

The function for the progress indicator is set up to run every 10,000 ms. The property API calls define an inline function as the 2nd parameter: viewer.getProperty('progress', **function**(progress) {} ). Inside the function you can access the property value and display as required.

|  |
| --- |
| **var** progress\_monitor = **function**() {  viewer.getProperty('progress', **function**(progress) {  **var** myProgress = timeFormat(progress);  document.getElementById("VideoProgress").textContent=myProgress;  });  }; |

### Trapping Player Events

The player can issue different events depending on the particular content and operation of the player. The script can define listener functions that will intercept the events and take action on the web page.

The first step is to set up the various listeners and then define the function that will process the various events.

 Scroll down to “player-controls placeholder 3” and insert the following code.

|  |
| --- |
| viewer.addListener('ready', onEmbedEvent);  viewer.addListener('quality', onEmbedEvent);  viewer.addListener('live', onEmbedEvent);  viewer.addListener('offline', onEmbedEvent);  viewer.addListener('playing', onEmbedEvent);  viewer.addListener('finished', onEmbedEvent);  viewer.addListener('content', onEmbedEvent);  viewer.addListener('duration', onEmbedEvent); |

In general the listener functions will simply hide or expose different page elements to reflect the updated state of the player. Scroll back up to “player-controls placeholder 4” and insert these next blocks of code. They are all very similar, primarily the target elements change.

**Ready** event

 Insert this **case** block. This event precedes other events that will follow so it’s basically a reset event.

|  |
| --- |
| **case** "ready":  clearFields();  **break**; |

**Quality** event

Insert this **case** block. The quality function will be reviewed at the end of this section.



|  |
| --- |
| **case** "quality":  populateQualitySelector(data);  **break**; |

**Live** event

 Insert this **case** block. For the various status changes, you have to make certain the correct state is displayed and any contradictory states are hidden.

|  |
| --- |
| **case** "live":  $('.st-live').show();  $('.st-offline').hide();  **break**; |

**Offline** event

 Insert this **case** block. For the various status changes, you have to make certain the correct state is displayed and any contradictory states are hidden.

|  |
| --- |
| **case** "offline":  $('.st-offline').show();  $('.st-live').hide();  $('.st-playing').hide();  **break**; |

**Playing** event

 Insert this **case** block. The **Playing** event is a bit different. The **data** argument will be TRUE or FALSE to indicate whether or not the content is playing.

|  |
| --- |
| **case** "playing":  **if** (data) {  $('.st-playing').show()  $('.st-offline').hide();  $('.st-ended').hide();  } **else** {  $('.st-playing').hide();  $('.st-offline').show();  }  **break**; |

**Finished** event

 Insert this **case** block.

|  |
| --- |
| **case** "finished":  $('.st-ended').show();  $('.st-playing').hide();  **break**; |

**Duration** event

 Insert this **case** block. **Duration** is also a bit different, as this is returning a property, not indicating a change in the player’s state. The value is returned in seconds, so it is converted by a separate function to show minutes and seconds.

|  |
| --- |
| **case** "duration":  viewer.getProperty('duration', **function**(duration) {  console.log("playing duration function", duration);  myDuration = timeFormat(duration);  document.getElementById("VideoDuration").textContent=myDuration;  });  **break**; |

### Displaying and Changing the Player Quality

The web page has a selection field for changing the video quality. It is populated by the quality event that was just defined.

|  |
| --- |
| <label>Video quality</label>  <div class=*"control-select"*>  <select class=*"quality-selector"*></select>  <div class=*"select"*><span class=*"label"*>Quality</span></div>  </div> |

The function called by the event updates the selection field with the contents of the quality event. It takes the quality data passed in and builds a new selection list. The incoming data is an array of bitrate values:

[ {"active": true, "bitrate": 32, "height": 480", "id": 0, "label": "480p", "selected": false}, {...} ]

The function pulls out the “label” and “selected” values to build a new list and indicate which one is active (it’s not necessary to understand the syntax, leave that for insomnia therapy).

|  |
| --- |
| /\* use quality event data to repopulate the quality selection options \*/  **var** populateQualitySelector = **function**(qualityData) {  **var** list = $(".quality-selector");  list.empty();  /\* the quality data is an array of current quality values \*/  **for** ( **var** id **in** qualityData) {  list.append('<option value="' + id + '" ' + (qualityData[id].selected ? 'selected="selected"': '') + '>' + qualityData[id].label + '</option>');  }  // Updating value of the custom select gui element  list.next().find('span').text("Current quality: " + list.find('option:selected').text());  } |

Selecting an option from the list will update the player’s quality setting. The function is already set in the script file to trap changes to the selection list.

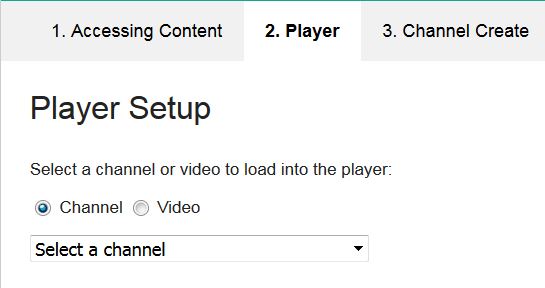
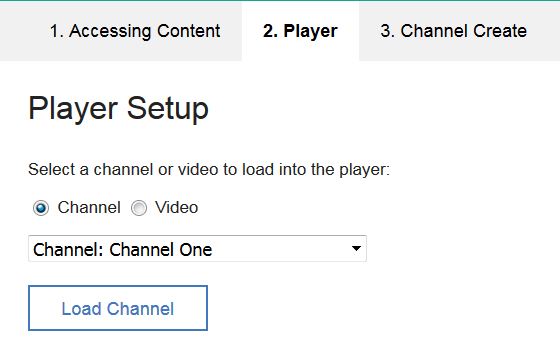
|  |
| --- |
| $(".quality-selector").change(**function**() {  viewer.callMethod("quality ", $(**this**).val());  } |

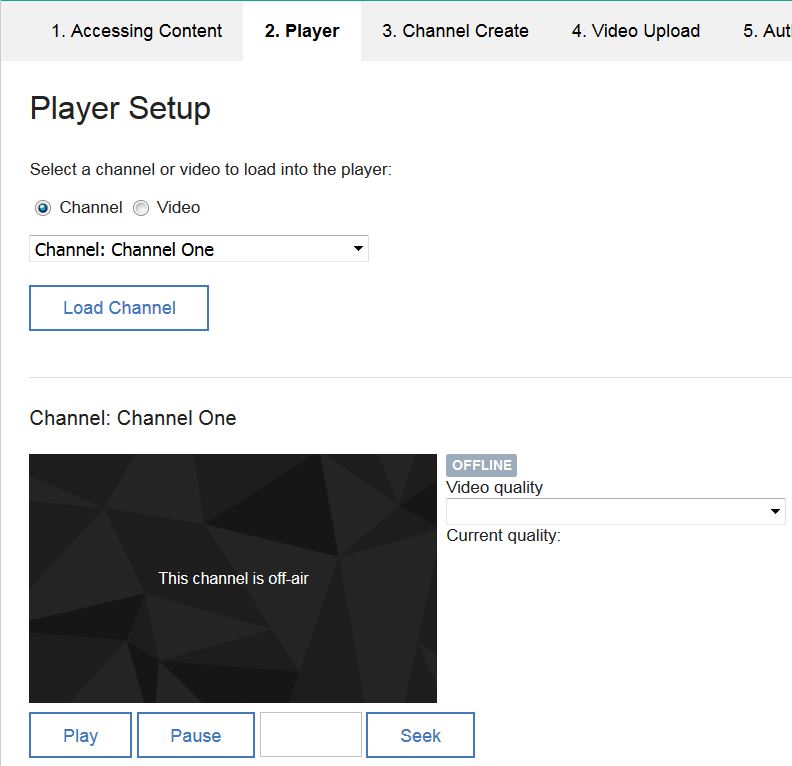
## Lab 2 Checkpoint

On the command line, push the updated application to Bluemix and then refresh the browser page.

The checkpoint items for this lab cover all the interactions with the embedded player.

1. Are the channels and video selection lists populated correctly
2. Are the appropriate buttons displayed in response to changing selections
3. Does the **Load** button change the player content
4. Do channels and videos show different status fields on the right of the player
5. Do **Play** and **Pause** perform as expected
6. Does the **Status** indicator match the player mode
7. Does **Seek** move to the specified position
8. Do **Duration** and **Progress** match the current content
9. Is the **Quality** list populated and does it show the current setting

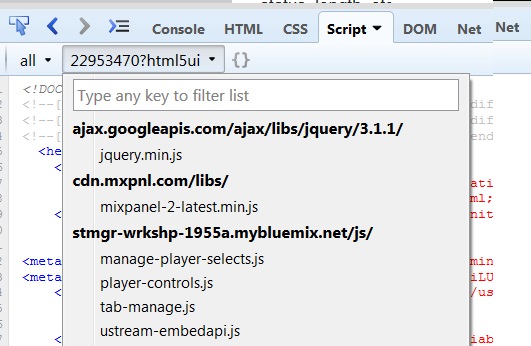
 



## Lab 2 Debugging

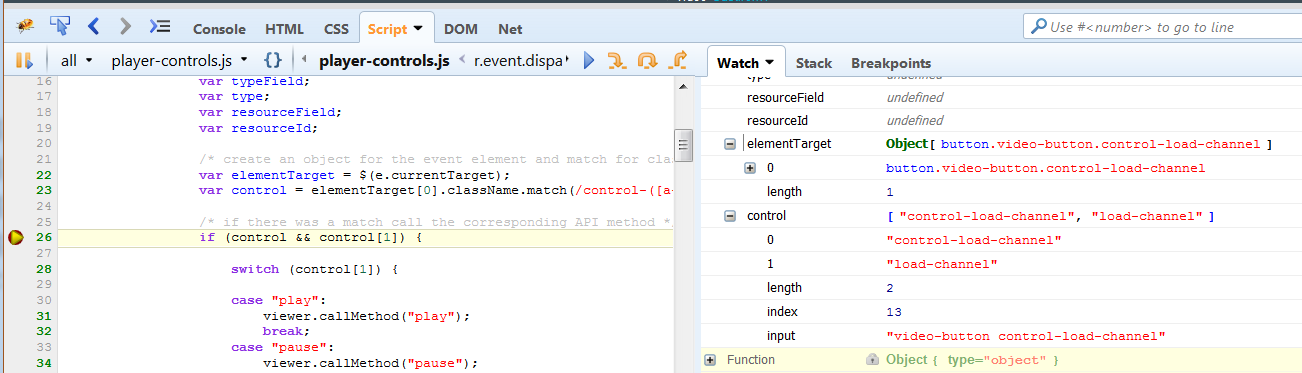
Lab 2 consists primarily of JavaScript programming to manage the functionality of the web page and CSS to manage the look of the page. The CSS definitions are all predefined so element styling should be correct. Debugging problems in the scripts was discussed in the presentation. If any function does not work then here are the two places to start.

In FireFox, once you open the FireBug tool, check that all the scripts have been loaded by the page:



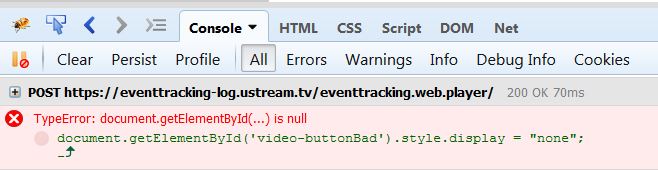
If some operation is simply unresponsive then this list will verify that the scripts have been successfully loaded.

To check the operation of a particular script a breakpoint can be set. Select the script from the dropdown as shown above and click to the left of a script command to set a breakpoint.

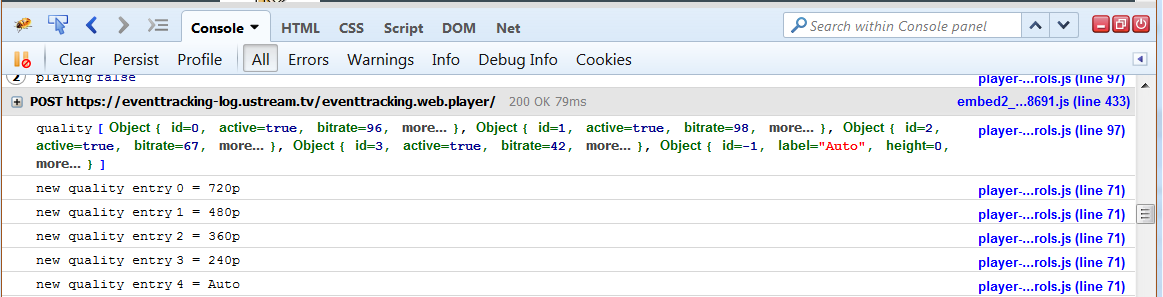


The right frame shows the content of all the variables in this context. Here you can see the **elementTarget** and **control** variables that were created from the object sent to the function.

If there are syntax errors in the script the **Console** tab will show them on the **All** or **Errors** page.



Also, the **Console** tab’s **All** page shows the output of “console.log(“new data:”, data)” commands inserted into a script.



End of Lab 2

# Lab 3 Creating a Channel

The Streaming Manager API provides the ability to manage the channels, including the ability to create one from outside the account UI. The request should provide a title and optionally a description.

## Updating the Home Page

In Eclipse go to the **index.php** file. The HTML code for the “3. Create Channel” tab is already in place, which consists of a standard HTML form element calling a PHP script as its POST action.

|  |
| --- |
| <form **action=*"php/channel-create.php"*** method=*"post"* enctype=*"multipart/form-data"*>  <p>Channel Title</p>  <div>  <input type=*"text"* name=*"channelTitle"*>  </div>  <p>Description</p>  <div>  <input class=*"channel-description"* type=*"text"* name=*"channelDescription"*>  </div>  <div>  <input type=*"submit"* value=*"Create Channel"*>  </div>  </form> |

There is one addition to the page that can be made now. When the create script finishes it redirects itself back to the home page. While not required, it is possible to show the results of the create task on the home tab as verification to the user.

In the Player tab find the “<!--HTML channel create placeholder-->” comment. Insert this code in its place to display the results of the script that it saved in a session variable.

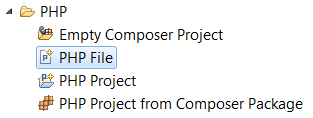


|  |
| --- |
| <div>  <h4>  <?php  **if** (**isset** ( $\_SESSION ["channel\_create\_error"] ) && strlen ( $\_SESSION ["channel\_create\_error"] ) > 0) {  **echo** "<hr>";  **echo** "Channel create response: " . $\_SESSION ["channel\_create\_error"];  $\_SESSION ["channel\_create\_error"] = "";  }  ?>  </h4>  </div> |

## Add a Channel Create script

Right-click the ***php*** folder and select **New / Other / PHP / PHP File** (or **New / PHP File** if it’s already in the list). Name it **channel-create.php**.





 The default file will have an opening PHP tag but no closing tag, so fix that now.

|  |
| --- |
| <?php  ?> |

 Start by creating a new session and then check for an access token. Recall that the token session variable was populated in the **set\_key.php** file as part of the login task. If for some reason that variable is empty then a message can be displayed on the Content tab using a new session variable in the PHP script you just added to the page. The “header ()” command will switch location back to the host server name that in this application is the base URL.

|  |
| --- |
| session\_start ();  $home\_page = "https://" . $\_SERVER ["SERVER\_NAME"];  **if** (strlen ( $\_SESSION ["ustream\_token"] ) == 0) {  $\_SESSION ["channel\_create\_error"] = "Missing access token";  header ( "Location: " . $home\_page );  **exit** ();  } **else** {  } |

The request for this function is very simple. Provide the authorization header and the two create parameters from the form. The **$\_POST** array contains all of the input fields contained in the HTML form.

Pull out the title field and then create two parameter arrays, one for the request and one for the headers. Make sure the code is inside the ‘else {...}’ code block.



|  |
| --- |
| $CHANNEL\_TITLE = $\_POST ['channelTitle'];  $curl\_formData = **array** (  'title' => $\_POST ['channelTitle'],  'description' => $\_POST ['channelDescription']  );  $headerFields = **array** (  'Authorization: Bearer ' . $\_SESSION ["ustream\_token"],  'content-type: application/x-www-form-urlencoded'  ); |

 Then make the request. This should look quite familiar from previous labs. The authorization header retrieves the token saved in the session environment, and the form data pulls in the two input fields.

|  |
| --- |
| $curl = *curl\_init* ();  *curl\_setopt* ( $curl, *CURLOPT\_URL*, "https://api.ustream.tv/users/self/channels.json" );  *curl\_setopt* ( $curl, *CURLOPT\_HTTPHEADER*, $headerFields );  *curl\_setopt* ( $curl, *CURLOPT\_POSTFIELDS*, *http\_build\_query* ( $curl\_formData ) );  *curl\_setopt* ( $curl, *CURLOPT\_RETURNTRANSFER*, 1 );  $result = *curl\_exec* ( $curl ); |

 Decode the results and also retrieve the HTTP status code. The code will need to be checked to verify the operation was successful.

|  |
| --- |
| $resultJSON = *json\_decode* ( $result );  $status = *curl\_getinfo* ( $curl, *CURLINFO\_HTTP\_CODE* );    *curl\_close* ( $curl ); |

 The expected return code is a 201. Anything else should send a response message back to the home page. The **hint** attribute isn’t always populated, so make a check so it can be properly formatted in the message. For both paths, update the session variable so the home page can display the results.

|  |
| --- |
| **if** ($status == 201) {  **$\_SESSION** ["channel\_create\_error"] = "Channel '" . $CHANNEL\_TITLE . "' successfully created";  } **else** {  **if** (*strlen*( $resultJSON->hint ) > 0) {  $error\_hint = ": " . $resultJSON->hint;  } **else** {  $error\_hint = "";  }    **$\_SESSION** ["channel\_create\_error"] = "(HTTP $status) " . "$resultJSON->error" . $error\_hint;  } |

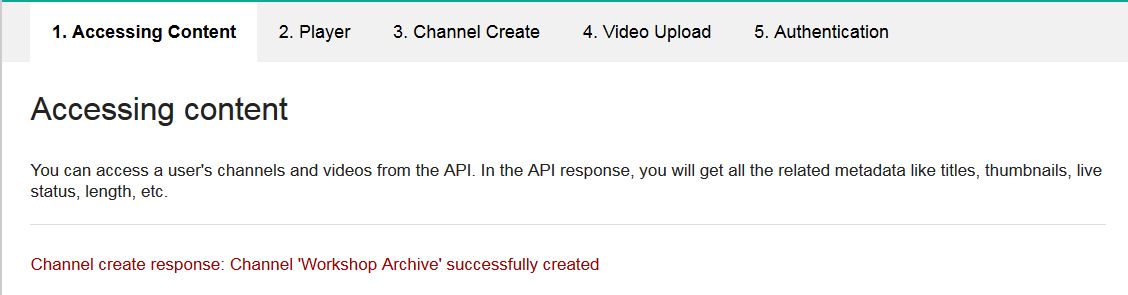
Note that the channel title that was initially retrieved can be made part of the status message.

 And finally close the script with a location switch and an exit.

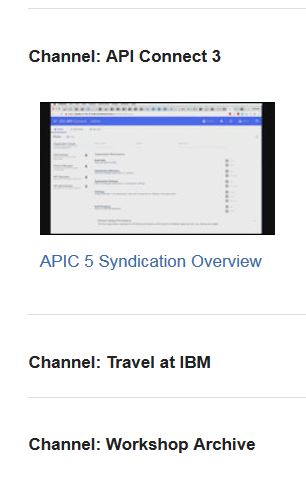
|  |
| --- |
| *header* ( "Location: " . $home\_page);  *exit*(); |

## Lab 3 Checkpoint

Issue you **> cf** push an refresh the browser page. For this lab the UI is unchanged. Go to tab 3, enter values for the title and description and click the **Create** button. After a brief pause the home tab should be redisplayed with a message from the create script.



In addition, the new channel should now be visible at the bottom of the channel listing.



.

# Lab 4 – Upload Video Files to Channel

A fundamental function for a video channel is the ability to upload new video content to the channel. The Streaming Manager Channel API provides a request sequence that provides for the uploading and management of new videos.

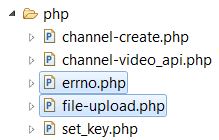
The upload API has a sequence of 3 requests required to complete the uploading process.

1. Call for an FTP URL and internal video id
2. FTP the file
3. Set the uploaded video to a ‘ready’ state

## Lab Setup

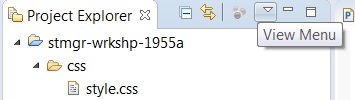
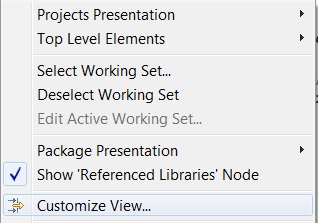
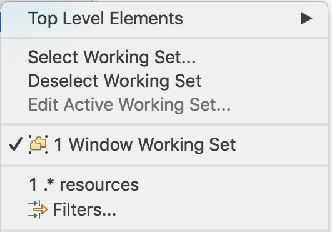
This lab requires a generous amount of PHP coding. As such, a template for the PHP upload script has been provided. There is also a PHP file that includes a common set of FTP error codes. There are many more possible codes but these are the most likely for an FTP connection.

 Import the files ***Lab Artifacts/Lab4/file-upload.php*** and ***errno.*php** into the Eclipse project’s ***php*** folder.

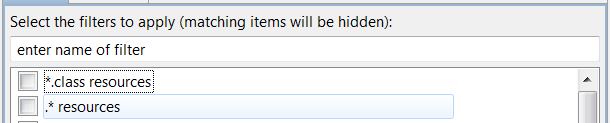


The presentation also mentioned the need for possible server adjustments. In any environment these attributes would likely need to be adjusted to accommodate the expected size of files to be uploaded.

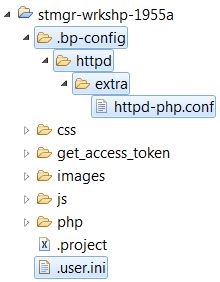
The new files are named “.\*” so they aren’t visible by default in Eclipse. Open the View menu in the Project Explorer tab. On Windows select **Customize View** from the menu, on Mac select **Filters**.

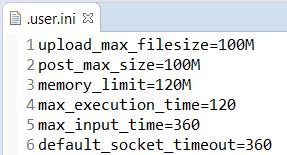
Resources checked on the list will be excluded from any views. Uncheck the “.\* resources” entry and close the window.



 Select the project folder in Eclipse and import ***Lab Artifacts/Lab 4/.user.ini***. Then select the project folder again and create a new folder ***.bp-config*** (right-click folder -> New -> Folder). Select ***.bp-config*** and create a new folder ***httpd***. Select ***httpd*** and create a new folder ***extra***. Import ***Lab Artifacts/lab 4/httpd-php.conf*** into ***extra***.

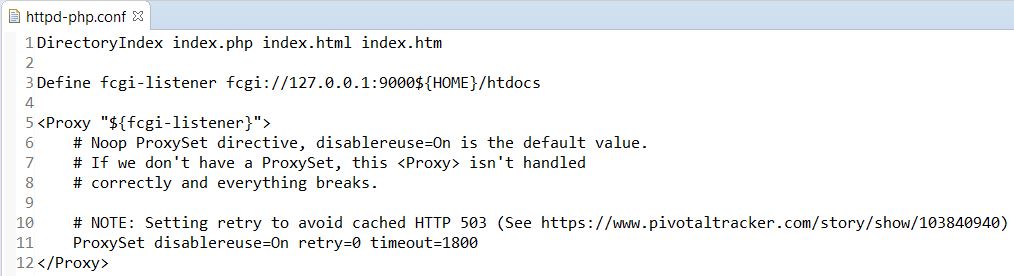


These custom PHP parameters in **.*user.ini*** increase the server limits on memory, file size and POST data size, overriding the default parameters in the server’s ***php.ini*** file.



***\* Note for Apple Mac users :***by default Mac reserves file names starting with “.” for system files. After importing **.user.ini** the system may have renamed it to **\_user.ini**. Right-click **\_user.ini** and select either **Rename** or **Refactor/Rename** to change it back to **.user.ini**.

For the Apache server itself the configuration file will be replaced entirely by ***httpd-php.conf***. The only change being implemented is the addition of a proxy timeout.



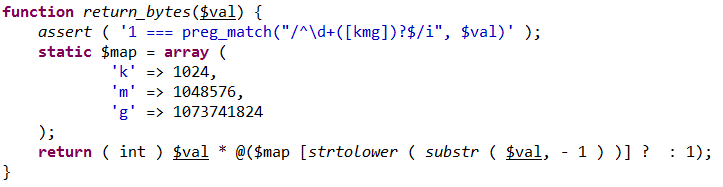
## File Upload Overview

As before the file upload script has markers in the comments to assist with inserting the lab code.

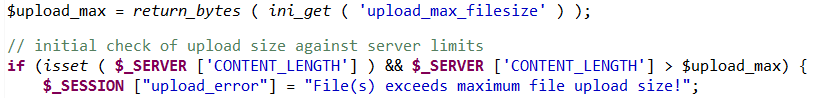


There are 6 snippets of code flagged in this way that will need to be replaced.

A conversion function is included in the script to convert a value like “**100M**” to “**100000000**” for comparison with dynamic values in the script.

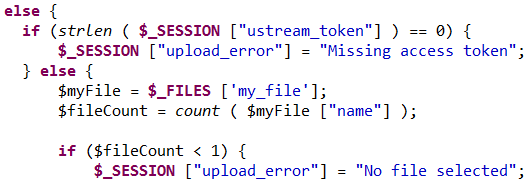


PHP provides a function *ini\_get()* to retrieve the values of the server’s initialization parameters. The script will use this function to validate the size of the POSTed data against the server limits before proceeding with the upload.

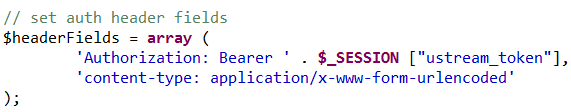


If the limit is exceeded a session variable is updated and control will be returned to the home page. This is the same technique used to send channel create status messages back to the home page

The script makes a couple of other checks before proceeding with the upload, to verify an access token and validate the input.



As in previous scripts a common authorization header array is defined. There are multiple API requests that will be made so using a variable name in the requests helps reduce code clutter.



The file selections are provided in a PHP variable named **$\_FILES** ['my\_file']. The array entry name comes from the HTML element on the page. The structure is a collection of attribute arrays; names, temporary names, sizes and such. The name of an individual file is referenced first by its attribute name and then the index.

**$\_FILES** ['my\_file'] ['name'] [0]

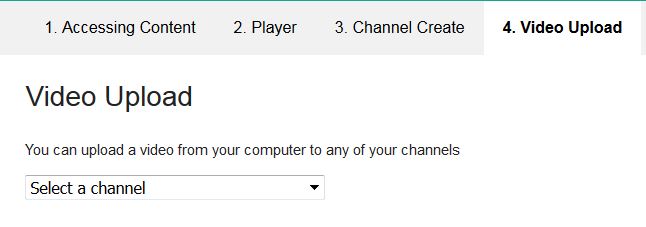
The other fields in the HTML form are referenced via a **$\_POST** variable, like the channel create script.

**$\_POST** ['title'] or **$\_POST** ['description']

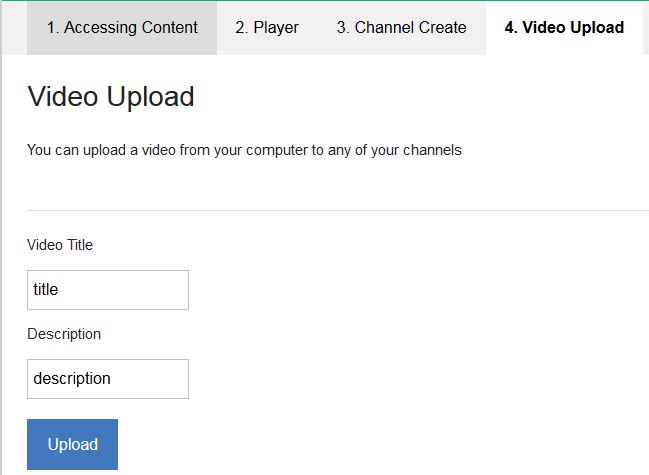
## Updating the home page

The home page needs new elements to provide input elements and to display results from the upload process. Some of the HTML code has been prepopulated. The Javascript tab code has not been updated so some of the fields displayed will be hidden once that code is changed.

This image shows the final design of the Upload tab.



This is what your application’s Upload tab should look like now.



### Display Status Message

Open the main ***index.php*** home page. As was done for channel create, code to catch status messages from the upload process can be included.

 Scroll to the top of the Content tab, about line 55. A new <DIV> with an <H4> heading element was inserted to show the channel create message. Insert this PHP code inside <?php ?> script block that is inside the <H4> block, to display messages generated by the Upload script.

|  |
| --- |
| **if** (**isset** ( $\_SESSION ["upload\_error"] ) && strlen ( $\_SESSION ["upload\_error"] ) > 0) {  **echo** "Video upload response: " . $\_SESSION ["upload\_error"];  $\_SESSION ["upload\_error"] = "";  } |

### File Selection Fields

Scroll down to find “<!--HTML Upload placeholder 1-->”. The new code is inside a <FORM> element that is already defined. Note that the action value calls the php script that was uploaded in the initial part of this lab.

 To provide a selection list of video names replace the 1st placeholder with the following code. Note the “onchange=*"sourceSelect(this)"*>” event specification. This is the same function called on the Player tab. That function has code to manage elements on both tabs, but having separate functions would be an option.

|  |
| --- |
| <select id=*upload-channel* name=*"channel"*  onchange=*"sourceSelect(this)"*>  <option></option>  </select> |

 Scroll slightly down to “<!--HTML Upload placeholder 2-->”. Here will be the file control that allows the user to browse for files. Styling a file input element has some limitations and challenges, so a normal button is created that calls the hidden input element.

|  |
| --- |
| <button type=*"button"* class=*"video-button loader"* onclick="document.getElementById('uploader').click();">Choose a file ...</button>  <input type=*"file"* style="display: *none*;" name=*"my\_file[]"* id=*"uploader"* onchange=*"sourceSelect(this)"* multiple /> |

 The last change to the page at “<!--HTML Upload placeholder 3-->” is a display element that can be populated with the set of selected filenames.

|  |
| --- |
| <ul id=*"upload-selects"*>  </ul> |

### Building the channel selection field

In Lab 2 the **js/player-controls.js** script was used to automatically populate the channel selection list for the Player tab. That same script can be used to populate the channel selection list for the Upload tab. This will be an exercise without any code provided.

|  |
| --- |
| <select id=*upload-channel* name=*"channel"* onchange=*"sourceSelect(this)"*>  <option></option>  </select> |

 Using the existing **player-controls.js**, add code to populate this field. It will be very similar to the code for the Player channel field. Note, it will be tempting to share variables for the two channel elements, but that may not work as expected. You will have to duplicate the Player channel code for the Upload channel element. If you get stuck, refer to **Lab Artifacts/Lab4/player-controls-codeblock.js**.

### Update the CSS Styling

By default the Upload tab will only show the channel selection list, the other fields will be hidden until a channel is selected. Open the ***js/tab-manage.js*** file.

 Replace the “tab-manage placeholder 2 for Upload” placeholder with this code.

|  |
| --- |
| /\*  \* Upload tab  \*/  **if** (apiFunction == "upload") {  // Hide the upload browse button and video parameter fields  document.getElementById('upload-browse').style.display = "none";  document.getElementById('upload-parameters').style.display = "none";  document.getElementById('upload-show').style.display = "none";  } |

## Building out the upload script

There are 6 sections where code will be inserted. As you will see in the script template, in addition to the markers for new code that are present, the conditional structure is in place to accommodate responses from the various requests.

### Prepare for FTP connection Request

The 1st step is to get an FTP connection. This request will return two important pieces of information.

1. FTP URL for video file
2. Uploaded video id

Open the ***php/file-upload.php*** file. Scroll down to the “file upload placeholder 1” and replace it with this code.

 First add code to retrieve the POST parameters.

|  |
| --- |
| // retrieve channel id, title and description  $CHANNEL\_ID = **$\_POST** ['channel'];  $video\_title = **$\_POST** ['title'];  $video\_description = **$\_POST** ['description']; |

 Next, go to “file upload placeholder 2” and set up the query fields for the POST. This parameter will direct the site to return the required FTP connection values.

|  |
| --- |
| // set query fields  $fields = **array** ( "type" => "videoupload-ftp" );  $queryFields = http\_build\_query( $fields ); |

 Go to “file upload placeholder 3” and set up the form fields for title and description, and a fixed value for the ‘protect’ option. This will automatically publish the video in the channel when it is finished transcoding. This is optional and could be removed to allow the video to be manually published at a later time.

|  |
| --- |
| // set POST Form fields  $curl\_videoFormData = **array** (  'title' => $video\_title,  'description' => $video\_description,  'protect' => "public"  ); |

The rest of the code is contained in a loop of the file selection elements. The current design only allows a single title and description field, so if there are multiple files they will all use the same values, if entered. The first snippet of code in the loop will substitute the current filename for the title if it is still the default “title” or has been removed.

### Pick Up the Title

 Move next to “file upload placeholder 4”. This is a small snippet to conditionally extract the filename as the title. If there is no title text or it’s still the default, update the value in the form parameters array with the filename extracted from the PHP files array.

|  |
| --- |
| // if no title was entered, use the filename as the title  **if** (strlen($video\_title) == 0 || $video\_title == "title") {  $curl\_videoFormData["title"] = pathinfo($myFile ['name'] [$i], PATHINFO\_FILENAME);  } |

### Request an FTP Connection

Uploading a file is a 2-step process. This initial request provides connection information used to perform the actual upload.

 Now replace “file upload placeholder 5” with the request code. This code completes the settings for the request. Take note of the PHP function used to convert the array of form data fields into the correct format.

|  |
| --- |
| // first get an FTP connection from Streaming Manager  $curl = curl\_init ();  curl\_setopt ( $curl, CURLOPT\_URL, "https://api.ustream.tv/channels/$CHANNEL\_ID/uploads.json/?$queryFields" );  curl\_setopt ( $curl, CURLOPT\_HTTPHEADER, $headerFields );  curl\_setopt ( $curl, CURLOPT\_POSTFIELDS, http\_build\_query( $curl\_videoFormData ) );  curl\_setopt ( $curl, CURLOPT\_RETURNTRANSFER, 1 ); |

The request is successful if an HTTP status code of 201 is returned. Now the upload request can be made.

### Make an FTP Request

 After the result is formatted find “file upload placeholder 6”. The initial request returns the target URL for this file. However, it needs to have the file extension type appended to it. Add these 3 instructions to extract the extension, append it to the returned URL, and retrieve the video id.

|  |
| --- |
| // pull the extension from the form field  $ext = pathinfo($myFile ['name'] [$i], PATHINFO\_EXTENSION);    // the URL is provided, the only addition is the extension of the original file  $ftp\_url = $response->url . ".$ext";  // the video id will be used when updating the status of the uploaded video  $video\_id = $response->videoId; |

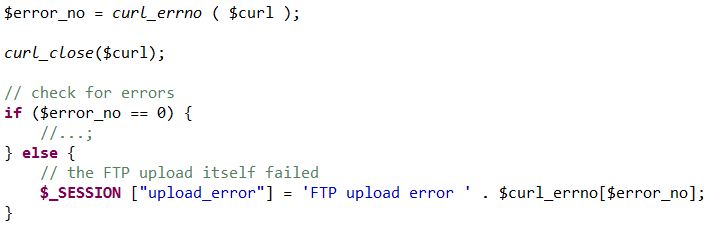
 Placeholder “file upload placeholder 7” includes code to generate a ‘read’ file descriptor for the file copy that exists on the server. The PHP files array includes an attribute that provides the name of the local temporary file.

|  |
| --- |
| // get a file descriptor to the local file  $localFile = $myFile ['tmp\_name'] [$i];  $fp = *fopen* ( $localFile, 'r' ); |

 Finally, “file upload placeholder 8” part contains the request. These options are different than ones used previously, although they should be fairly self-explanatory.

|  |
| --- |
| // set up the FTP parameters.  $curl = *curl\_init* ();  *curl\_setopt* ( $curl, *CURLOPT\_UPLOAD*, **TRUE**);  *curl\_setopt* ( $curl, *CURLOPT\_URL*, $ftp\_url );  *curl\_setopt* ( $curl, *CURLOPT\_INFILE*, $fp );  *curl\_setopt* ( $curl, *CURLOPT\_INFILESIZE*, *filesize* ( $localFile ) ); |

The test for success of this step is of the curl error number. If the error number is 0 then the next steps will be invoked, otherwise the error number file that was included will be used to map the number to a message string that can be returned to the home page.



### Set the video to a ready state

The uploaded video has to be set to a ready state before transcoding will start. The request URL will contain the channel and video ids to make the connection.

 Scroll down to “file upload placeholder 9” after the error check. First set up the form parameters array, and then create the URL. The URL for this request will be used in the final status request, so a variable is created for reuse.

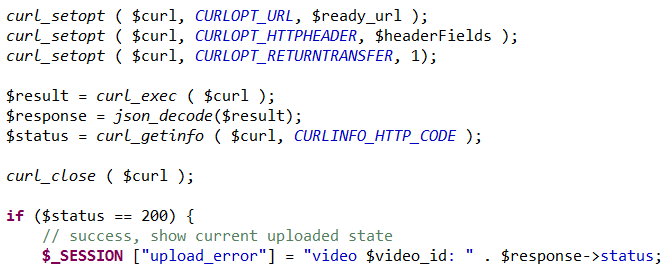
|  |
| --- |
| // for a successful upload go set the video to a ready state so it's visible  $curl\_readyFormData = **array** ( "status" => "ready" );    $ready\_url = "https://api.ustream.tv/channels/$CHANNEL\_ID/uploads/$video\_id.json"; |

 Move down to “file upload placeholder 10”. Now the request is ready to be made. Unlike previous requests, this is a PUT request as it is updating the state of the video object.

|  |
| --- |
| // make request to set ready state  $curl = *curl\_init* ();  *curl\_setopt* ( $curl, *CURLOPT\_URL*, $ready\_url );  *curl\_setopt* ( $curl, *CURLOPT\_CUSTOMREQUEST*, "PUT");  *curl\_setopt* ( $curl, *CURLOPT\_HTTPHEADER*, $headerFields );  *curl\_setopt* ( $curl, *CURLOPT\_POSTFIELDS*, *http\_build\_query*($curl\_readyFormData) );  *curl\_setopt* ( $curl, *CURLOPT\_RETURNTRANSFER*, 1); |

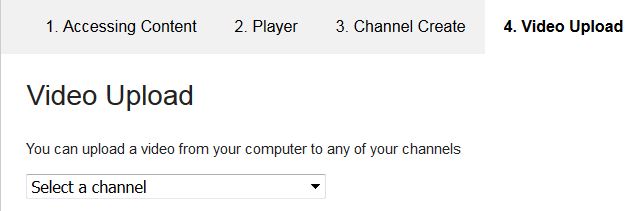
### Verify the uploaded video state

The optional step to retrieve the status of the uploaded video is already in place. A return code of 202 signifies a successful setting of the ready state. If that’s the case the same URL can be used in a GET request to return the current status. At this point the status will be “transferred” if the upload was successful.

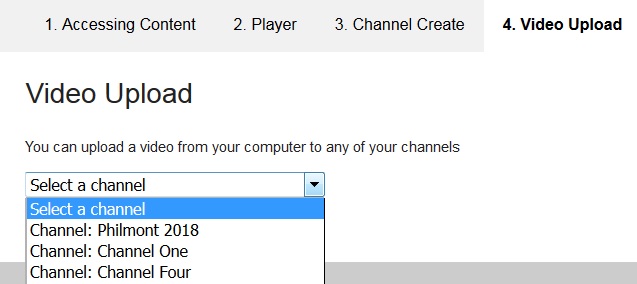


## Lab 4 Checkpoint

After pushing the changes to Bluemix, check the refreshed UI against the target design. Are the input fields and load button hidden by default?



Is the channel selection list populated and is the default entry selected?



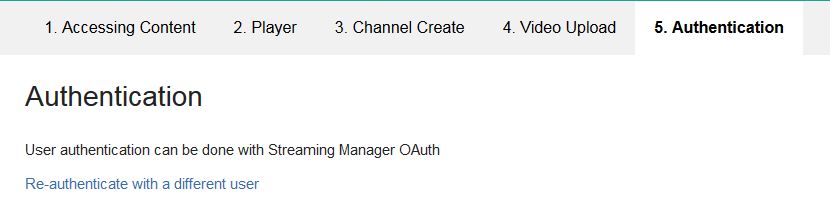
Test the title field. If new title text is not entered the new video should use the filename as its title, otherwise it should reflect the entered text.

# Lab 5 – Change Account Access

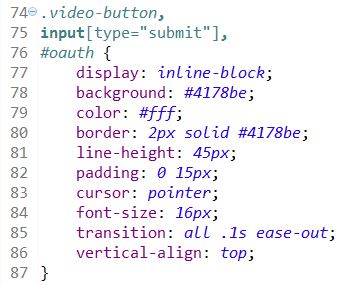
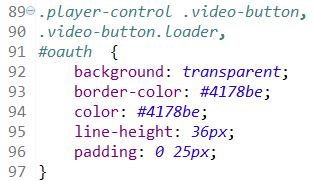
By default the application home page shows the channel and video content for a pre-configured account. This lab will demonstrate how to provide the ability to change which account’s content is displayed.

## Authentication Tab Formatting

The initial format of the Authentication tab already shows the link to the authorization site. However, it looks like a standard hyperlink like those seen on the Player tab.



To improve the visual look of the tab there is styling that can be applied to this link to make it look more like the buttons that exist on the other tabs. The styling definitions already exist in the default *style.css* file.

To activate the styling the current hyperlink definition needs to be updated.

 Open the **index.php** file, scroll to the bottom and insert **id=*"oauth"*** into the existing <a> hyperlink tag as shown below. This id matches the “#oauth” specification found in **css/style.css** (in CSS, “#” is for ids and “.” Is for class names).

|  |
| --- |
| <a **id=*"oauth"*** href="*https://www.ustream.tv/oauth2/authorize?...* </a> |

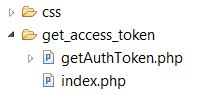
## Setup for OAuth Call

The setup for the OAuth request requires a change to the server structure. After the authentication is complete a redirect call is made to a specific location in the application, so that location needs to be defined. In addition, the redirect URL must match the value specified in the account’s API access definition.

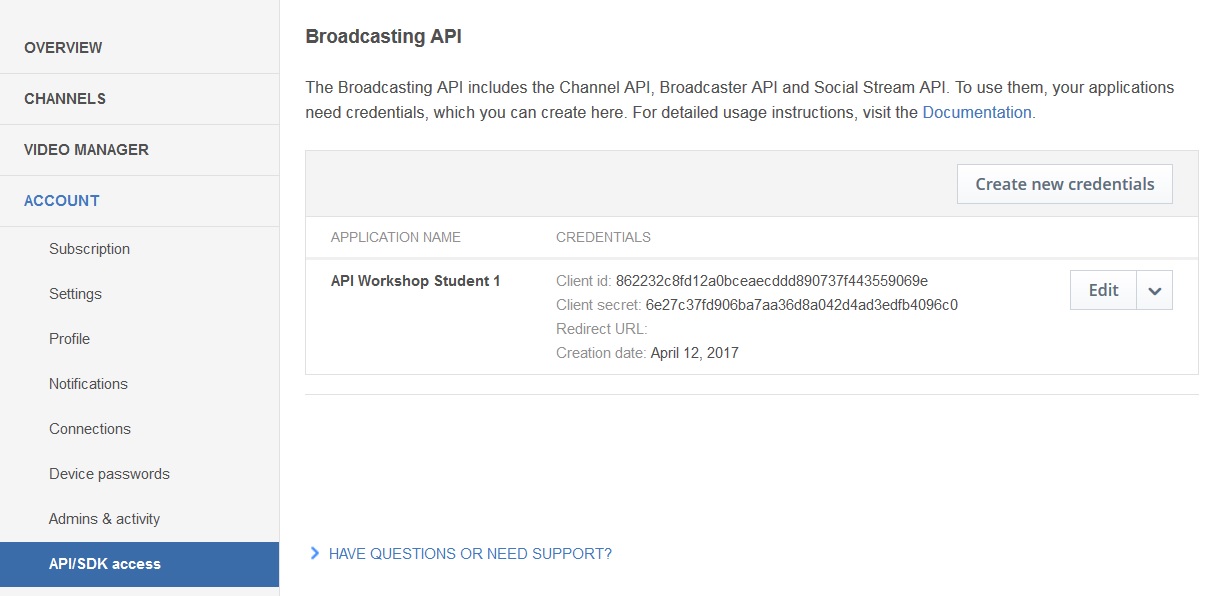
Create a new project folder at the top of the project named **get\_access\_token**.



 Import the files **Lab Artifacts/Lab5/index.php**and **getAuthToken.php** into the new folder.



Now log onto your Streaming Manager account. Under the **ACCOUNT** section click **API/SDK access**.

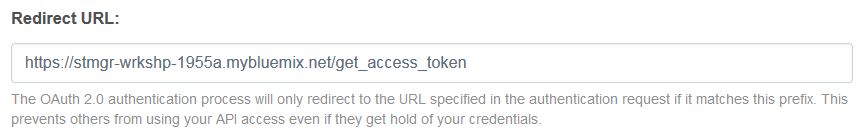


The entry contains a “Redirect URL” attribute that needs to be updated with the URL of the Bluemix application URL. Click on the **Edit** button next to the application entry.

Go back and open the command window where the application was pushed to Bluemix. The final summary includes text similar to the following.

|  |
| --- |
| requested state: started  instances: 1/1  usage: 128M x 1 instances  urls: **stmgr-wrkshp-1955a.mybluemix.net**  last uploaded: Fri May 19 01:47:36 UTC 2017  stack: cflinuxfs2  buildpack: php 4.3.10 |

Copy the “urls” string and use it to update the **Redirect URL** field. The “get\_access\_token” portion of the URL is fixed and matches the new folder created at the beginning of this lab.



Similarly, the re-authentication hyperlink on the home page must be updated in the same way.

Open the **index.php** file and scroll to the bottom. Update the hyperlink URL to match the Streaming Manager account property. If it remains as



“*redirect\_uri=https://****my-app****.mybluemix.net/get\_access\_token*”

then the script will fail as soon as it hits the token site without even providing the login fields.

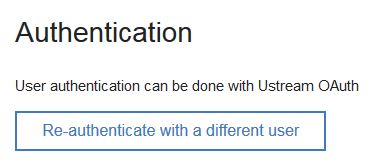
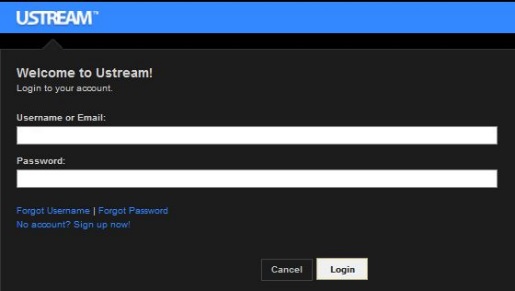
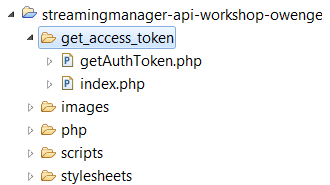
|  |
| --- |
| *... redirect\_uri=https://stmgr-wrkshp-1955a.mybluemix.net/get\_access\_token ...* |

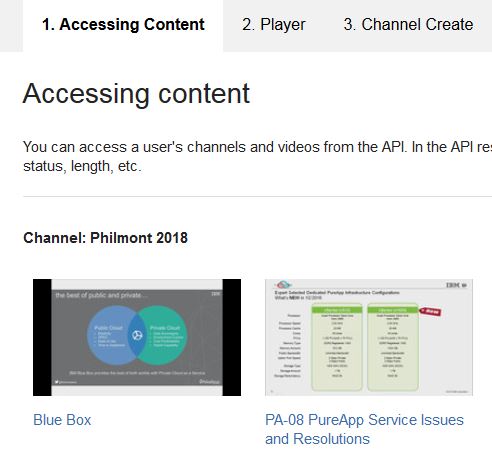
## OAuth Overview

Let’s review the basic sequence in the OAuth path:

App Home Page 🡺 Login Page 🡺 App Redirect Page 🡺 App Home Page

Or for the image-oriented:

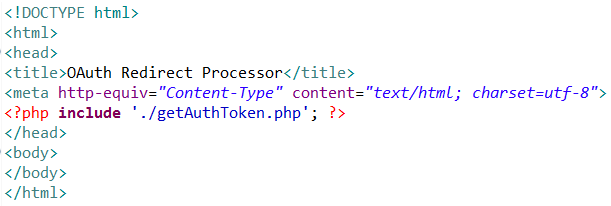


## Build OAuth Redirect Structure

The PHP script that will handle the redirect initially needs authentication values like the client key and secret. Because those values were initially loaded into session variable, they will be accessible by this script even though it is not running in the context of the initial home page.

### Review Redirect Home Page

The home page of the redirect was loaded at the beginning of this lab. It’s a minimal page that is only designed to hold and execute the PHP script that will acquire a new authorization token.



### Build redirect script

The *getAuthToken.php* script’s primary task is to extract the new access token from the OAuth site’s redirect and make it available to the main page. The script will utilize two built-in PHP variables: **$\_SERVER** and **$\_SESSION**. The **$\_SERVER** variable provides a number of useful values.

* $\_SERVER['SERVER\_PROTOCOL'] Returns the name and revision of the information protocol (such as HTTP/1.1)
* $\_SERVER['REQUEST\_METHOD'] Returns the request method used to access the page (such as POST)
* $\_SERVER['QUERY\_STRING'] Returns the query string if the page is accessed via a query string
* $\_SERVER['HTTP\_HOST'] Returns the Host header from the current request

 Open the **get\_access\_token/getAuthToken.php** script and insert this code in place of “oauth request placeholder 1”.

|  |
| --- |
| /\* parse the query string into variable $queryParm \*/  *parse\_str*(**$\_SERVER**['QUERY\_STRING'], $queryParm); |

When this script receives control from the Streaming Manager OAuth site it will find an authorization code in the query string. The PHP “*parse\_str*()” command formats that string into an array that is named in the 2nd argument.

As noted, the initial client key and secret can be retrieved from the session environment.

 For “oauth request placeholder 2”, enter commands to load the client keys.

|  |
| --- |
| // retrieve client keys for subsequent requests  $CLIENT\_KEY = **$\_SESSION** ["client\_key"];  $CLIENT\_SECRET = **$\_SESSION** ["client\_secret"]; |

This script will be making a request similar to the one that logged in with the client key in Lab 1. The difference will be in the credentials type which will be ‘authorization\_code’ instead of ‘client\_credentials’, using the new OAuth token instead of the client key. The one major difference is that this request must also include the same redirect URL that was provided to the OAuth token site.

 Scroll to “oauth request placeholder 3” and insert this code to create a parameter array for the token request.

|  |
| --- |
| $curl\_formData = **array** (  'client\_id' => $CLIENT\_KEY,  'redirect\_uri' => "https://".**$\_SERVER**['SERVER\_NAME'].*substr*(**$\_SERVER**['REQUEST\_URI'], 0, *strpos*(**$\_SERVER**['REQUEST\_URI'], '/?')),  'grant\_type' => 'authorization\_code',  'code' => $queryParm['code']  ); |

Take note of the 'code' parameter that is referencing the 'code' property of the parsed query string. For the intellectually inquisitive, compare this parameter array with the one constructed in lab 1.

 Insert the request code in place of “oauth request placeholder 4”. The request format is the same as in lab 1, it’s the combination of form parameters that is the difference.

|  |
| --- |
| // set required request options  *curl\_setopt* ( $curl, *CURLOPT\_URL*, "https://www.ustream.tv/oauth2/token" );  *curl\_setopt* ( $curl, *CURLOPT\_HTTPHEADER*, **array** ('Authorization: Basic '. $CLIENT\_SECRET) );  *curl\_setopt* ( $curl, *CURLOPT\_POSTFIELDS*, $curl\_formData );  *curl\_setopt* ( $curl, *CURLOPT\_RETURNTRANSFER*, 1 ); |

The final steps will save the token in a **$\_SESSION** entry and send control back to the main page.

 Save the new token in place of “oauth request placeholder 5”.

|  |
| --- |
| /\* save access token for main site redirect \*/  **$\_SESSION**["ustream\_token"] = $resultJSON['access\_token']; |

### Review Before Leaving

A couple of notes before validating other parts of the application.

The OAuth request response could encounter an error. If so, part of that response object will be populated with an error description. The script already includes a check for possible errors and will use the existing session variable to send that back to the home page.

|  |
| --- |
| **if** (*isset*($resultJSON['error'])) {  **$\_SESSION**["token\_error"] = $resultJSON['error\_description' ;  } **else** { |

\* First and only test: add code to display this message on the home page

For all called PHP scripts, it’s important to explicitly end the code execution. Here the script switches the location to the home page, that in the application is at the server root, and exits.

|  |
| --- |
| *header*("Location: https://" . **$\_SERVER**["SERVER\_NAME"]);  *exit*; |

## How Does the Home Page Know?

The **php/set\_key.php** file already had code in place to accommodate an external authorization token. If the session variable is populated by this script then it will be discovered by *set\_key.php* and a login using the default client key and secret will be bypassed.

|  |
| --- |
| // If the access token has not been preloaded then a normal login is required  **if** (*isset*(**$\_SESSION** ["ustream\_token"])) {  // Save token from session environment  $auth\_token = **$\_SESSION** ["ustream\_token"];    } **else** { |

## Lab 5 Checkpoint

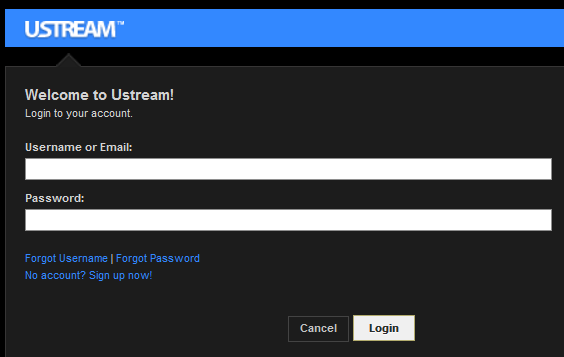
The application should now be able to switch accounts. Republish the application and refresh the browser. Open tab “5. Authentication”, then click the button to initiate the re-authentication.



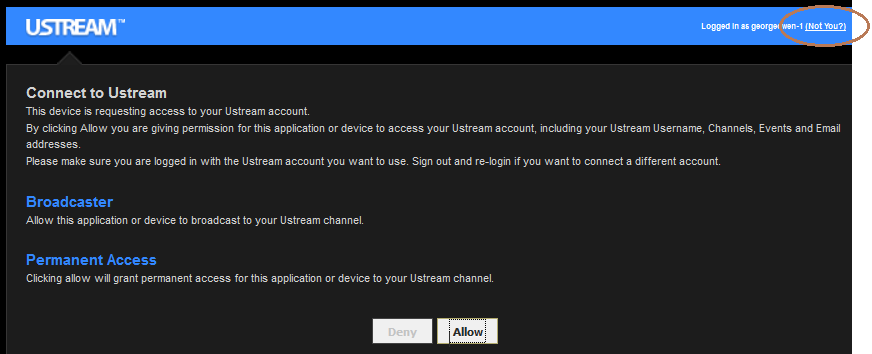
It will first show a login screen like this.

(click the ‘Not You?’ link at the top-right of the page if the authentication has been cached and the password prompt page is not presented)

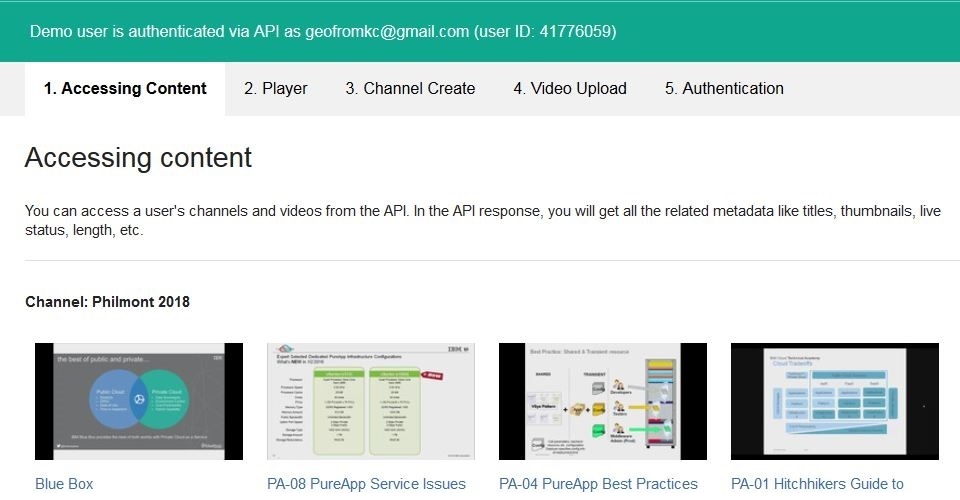


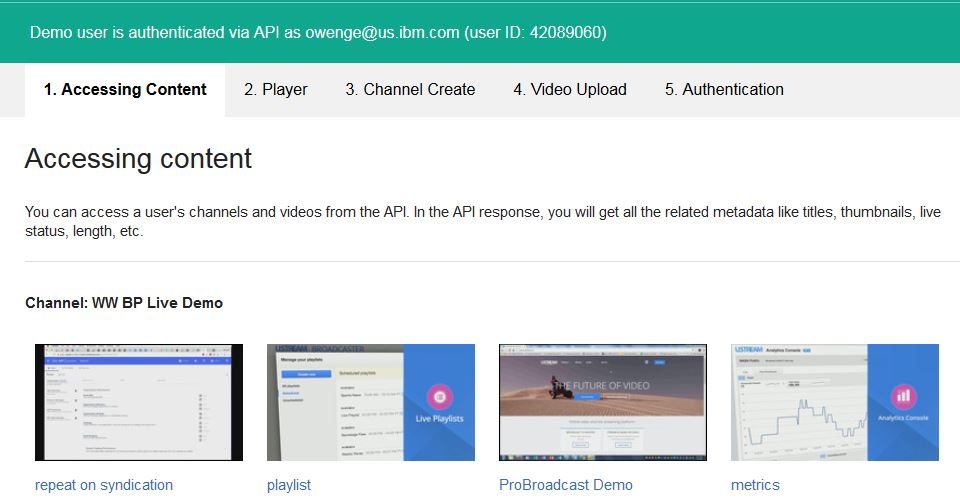


If the login is accepted, then the next page is presented. Click “Allow” to return to the application home page.



If the token was processed correctly by the **getAuthToken.php** script the user information should change and the content of the channel and video lists.





## Lab 5 Debugging

If an empty page like this is displayed then the redirect URL on the application home page probably doesn’t match what is configured in the Streaming Manager account. The actual format will change depending on the browser but the content will be the same.

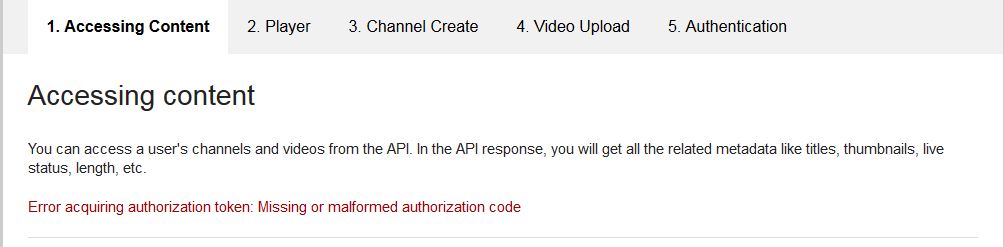


In the Firefox Firebug console there will also be an error message:

“NetworkError: 400 Bad Request - https://www.ustream.tv/oauth2/authorize?response\_type=code&client\_id=1c24beffd05bf89a46bca3cf164d3cda1d4101d9&redirect\_uri=https://**my-bad-1955a.mysbluemix.net/get\_access\_token**&... "

If the content doesn’t change with a login to a different account, then either the request for an access token using the OAuth token failed, or the new token isn’t being picked up by the main page PHP code. It would be possible to populate the session error variable with more than only the error message. You could generate a more involved HTML string that included input values that were part of the request.

If the home page was updated correctly, it would show whatever was placed into that session variable after returning from the re-authentication request.



It is also possible to generate messages in the server log with these PHP commands.

|  |
| --- |
| *error\_log*("flamethrower", 0);  *error\_log*("client\_id = $CLIENT\_KEY", 0); |

Starting with a unique string helps find these entries in the log. To monitor the log, go to the project folder on the command line and issue a cloud foundry command to start tailing the logs.

C:\...\workspace\stmgr-wrkshp-1955a>**cf logs stmgr-wrkshp-1955a**

Connected, tailing logs for app stmgr-wrkshp-1955a in org rtk@us.ibm.com / space dev as rtk@us.ibm.com...

2017-05-19T17:09:10.12-0400 [RTR/1] OUT stmgr-wrkshp-1955a.mybluemix.net - [19/05/2017:21:09:09.943 +0000] "GET /get\_access\_token?code=

0da283ec5c29162ddcf2f367&state=video HTTP/1.1" 301 0 327 "https://www.ustream.tv/oauth2/authorize?response\_type=code&client\_id=1c24beffd05bf

...

Using the commands noted above as an example, the logs would show entries like this (the line was separated for clarity):

2017-05-19T17:09:10.65-0400 [APP/0] OUT 21:09:10 httpd | [Fri May 19 21:09:10.632337 2017] [proxy\_fcgi:error]

[pid 54:tid 140217361053440] [client 169.45.162.88:36374]

AH01071: Got error 'PHP message: **flamethrower**\n

PHP message: **client\_id =** 1c24beffd05bf89a46bca3cf164d3cda1d4101d9\n

# Tool-based API Calls

The PHP library hides some of the actual structure of the API calls. This section will provide request and responses for some of the APIs used in this lab using curl, the Chrome Poster browser extension, and the Postman application.

## Retrieve Auth Token

### Curl

**URL, Headers and POST Fields**

C:\...\bin>curl.exe --verbose \

--url "https://www.ustream.tv/oauth2/token" \

--header "Authorization:Basic 382df3ea25effa83f01f52986cbc0f5cc27b9b7d" \

--header "content-type:application/x-www-form-urlencoded" \

--data "client\_id=9c24beffd05bf89a46bca3cf164d3cda1d4101d9&grant\_type=client\_credentials"

\* Trying 199.66.238.212...

\* TCP\_NODELAY set

\* Connected to www.ustream.tv (199.66.238.212) port 443 (#0)

...

> POST /oauth2/token HTTP/1.1

> Host: www.ustream.tv

> User-Agent: curl/7.54.0

> Accept: \*/\*

> Authorization:Basic 782df3ea25effa83f01f52986cbc0f5cc27b9b7d

> content-type:application/x-www-form-urlencoded

> Content-Length: 80

>

\* upload completely sent off: 80 out of 80 bytes

< HTTP/1.1 200 OK

< Server: ustream/web

< Set-Cookie: PHPSESSID=UstreamGuestSession; path=/; secure; HttpOnly

< Expires: Thu, 19 Nov 1981 08:52:00 GMT

< Cache-Control: no-store

< Pragma: no-cache

< Set-Cookie: ustream-guest=sWt2HH9rce8UWRwaHAAtfTyN2tcxFstDrKHuSQZiPefQanubA3FR%252BS%252BhparsXPHOBG57Y3zk7%252Bl6WYWAS3m0LYnP8IIfmLa4bIbpp3yz32VC12mo1Mq4VCJO5652QoF9ZUAeBqOChkeiOaL2%252F24w5czoGbMg2AIMQP6h6P8hn0o4AxCxfDLX1W4IAJnp7pEgwn72cKrvejfeXc6gmscWhibn1EFY2DIMwAtCPSzWUS8mErrxvqBBrmJM; path=/; httponly

< X-RSRC: sjc-web002

< Content-Type: application/json; charset=UTF-8

< Content-Length: 100

< Accept-Ranges: bytes

< Date: Fri, 16 Jun 2017 17:21:42 GMT

< Age: 0

< X-Cache-Info: 0 (-1)

< X-Domain: www.ustream.tv

< X-Cache: MISS

< X-BackendOrigin: www.ustream.tv-bck/sjc-web-cache05

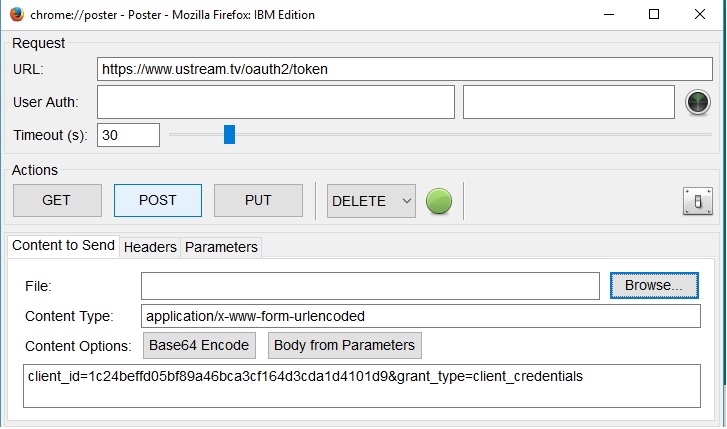
<

**Response:**

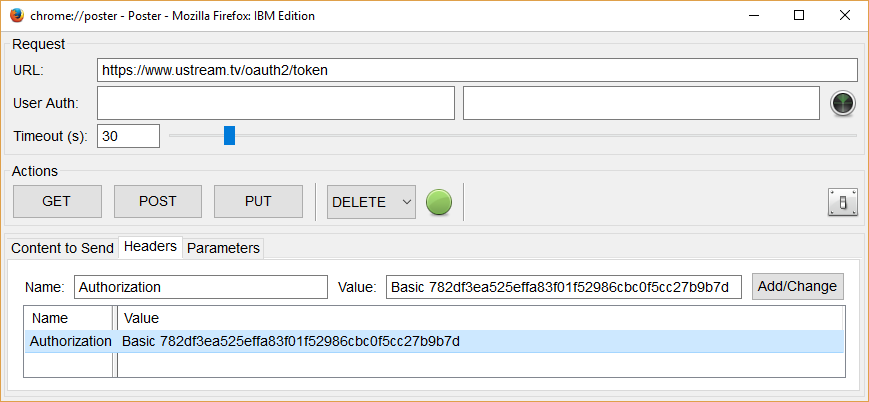
{"access\_token":"3006d00360fadd757e7cd87d2da9233becce6ef9","token\_type":"bearer","expires\_in":86400}\* Connection #0 to host www.ustream.tv left intact

### Chrome Poster

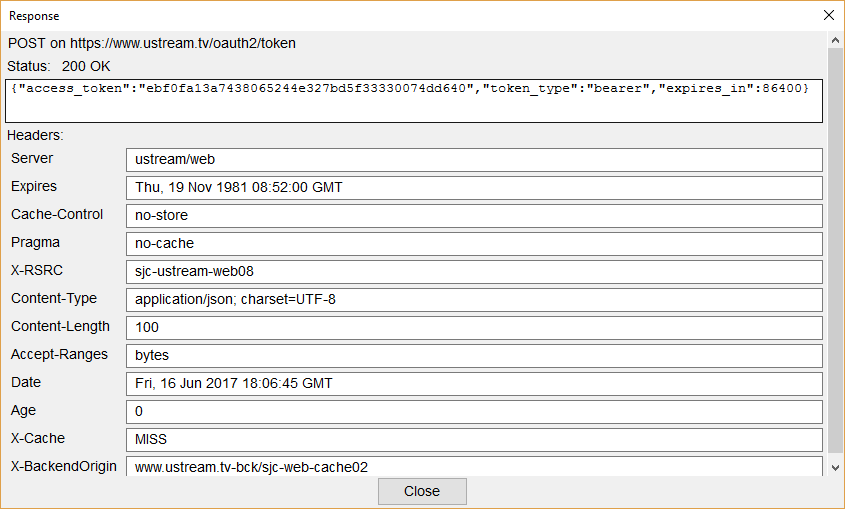
**URL, POST fields:**



**Headers**:

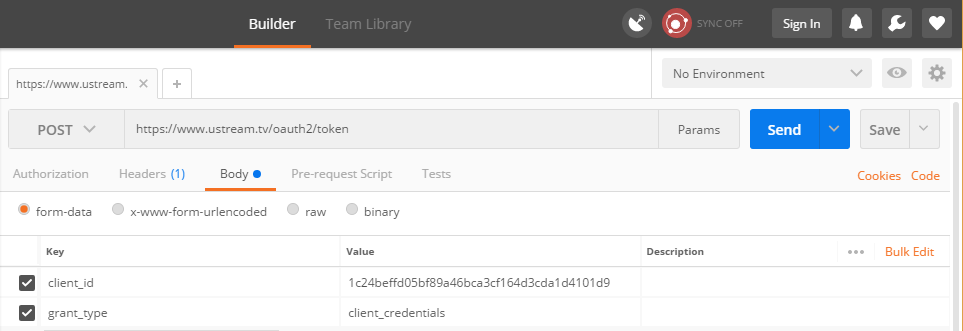


**Response**:

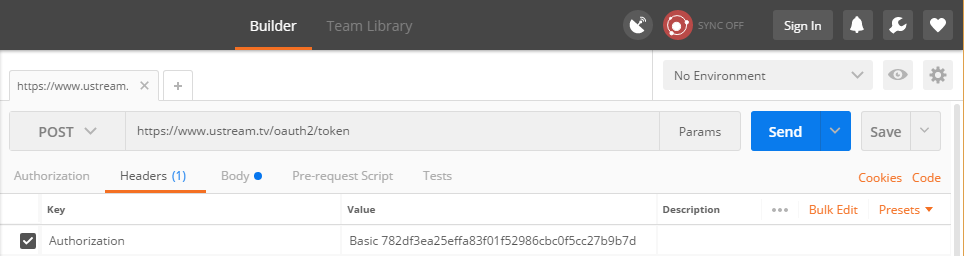


### Postman

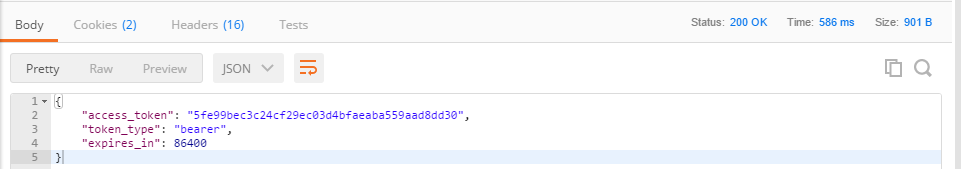
**URL, POST fields:**



**Headers**:



**Response**:



## Retrieve User Data

### Curl

**URL, Headers:**

C:\...\bin>curl.exe --verbose --url "https://api.ustream.tv/users/self.json" \

--header "Authorization:Bearer 3002d00360fadd757e7cd87d2da9233becce6ef9" \

--header "content-type:application/x-www-form-urlencoded"

\* Trying 199.66.238.68...

\* TCP\_NODELAY set

\* Connected to api.ustream.tv (199.66.238.68) port 443 (#0)

...

> GET /users/self.json HTTP/1.1

> Host: api.ustream.tv

> User-Agent: curl/7.54.0

> Accept: \*/\*

> Authorization:Bearer 3002d00360fadd757e7cd87d2da9233becce6ef9

> content-type:application/x-www-form-urlencoded

>

< HTTP/1.1 200 OK

< Server: Ustream/Api

...

< X-Domain: api.ustream.tv

< X-Cache-Info: 0 (-1)

< Age: 0

< Content-Length: 138

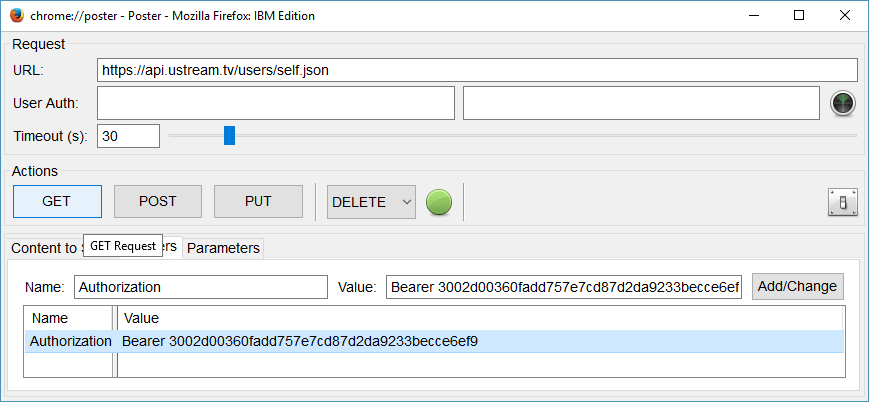
<

**Response:**

{"id":"41786059","username":"milesdavis-1","first\_name":"Miles","last\_name":"Davis","email":"mdavis@jmr.org","is\_adfree\_viewer":true}\*

### Chrome Poster

**URL, Header:**

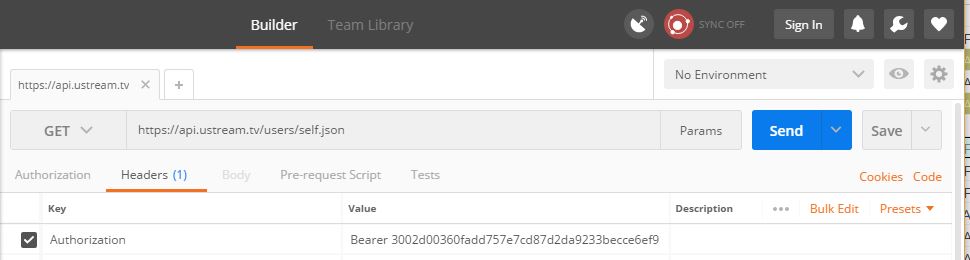


**Response**:

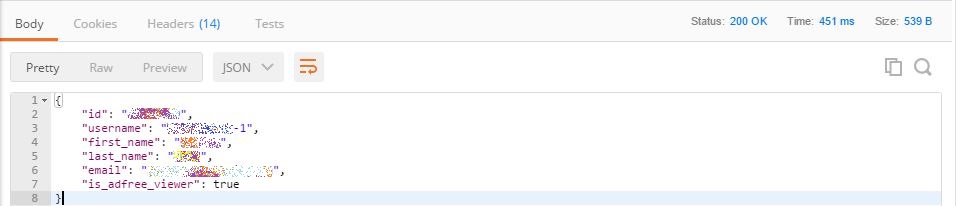


### Postman

**URL, Header:**



**Response**:



## Retrieve Channel Data

### Curl

**URL, Headers:**

C:\...\bin>curl.exe --verbose --url "https://api.ustream.tv/users/self/channels.json" \

--header "Authorization:Bearer 3002d00360fadd757e7cd87d2da9233becce6ef9"

--header "content-type:application/x-www-form-urlencoded"

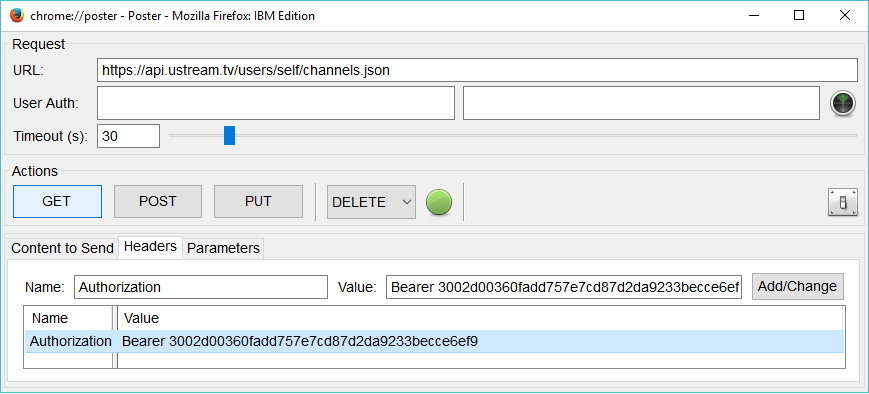
<verbose output unchanged>

**Response:**

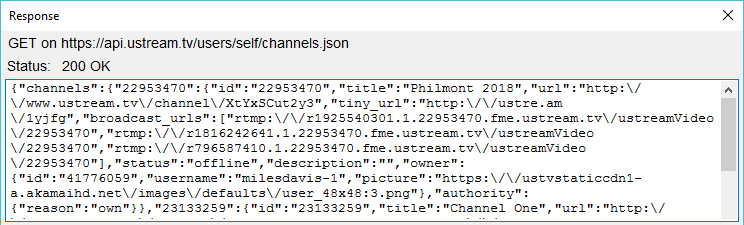
{"channels":{"22953470":{"id":"22953470","title":"Philmont\_2018","url":"http:\/\/www.ustream.tv\/channel\/XtYxSCut2y3","tiny\_url":"http:\/\/ustre.am\/1yjfg","broadcast\_urls":["rtmp:\/\/r2003162296.1.22953470.fme.ustream.tv\/ustreamVideo\/22953470","rtmp:\/\/r261464636.1.22953470.fme.ustream.tv\/ustreamVideo\/22953470","rtmp:\/\/r1550592237.1.22953470.fme.ustream.tv\/ustreamVideo\/22953470"],"status":"offline","description":"","owner":{"id":"41786059","username":"milesdavis-1","picture":"https:\/\/ustvstaticcdn1-a.akamaihd.net\/images\/defaults\/user\_48x48:3.png"},"authority":{"reason":"own"}}, …

### Chrome Poster

**URL, Header:**

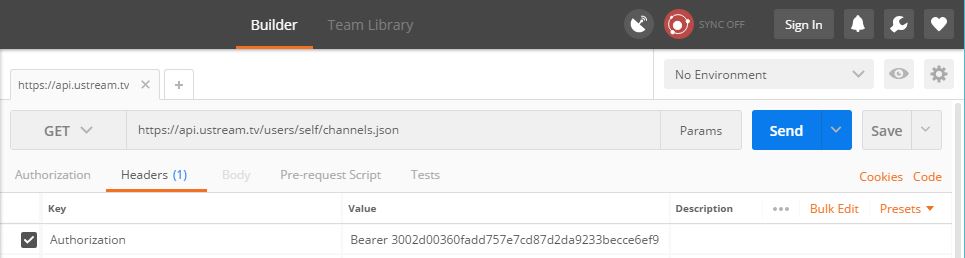


**Response**:



### Postman

**URL, Header:**



**Response**:



## Retrieve Videos

### Curl

**URL, Headers:**

C:\...\bin>curl.exe \

--url "https://api.ustream.tv/channels/22953470/videos.json?protect=private,public" \

--header "Authorization:Bearer 3002d00360fadd757e7cd87d2da9233becce6ef9"

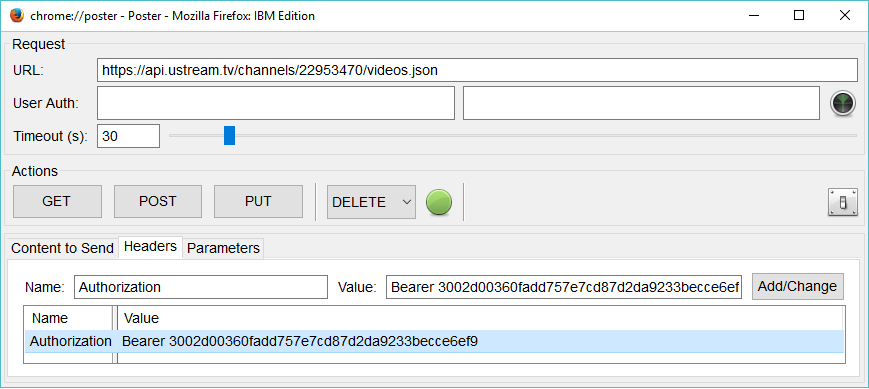
--header "content-type:application/x-www-form-urlencoded"

**Response**:

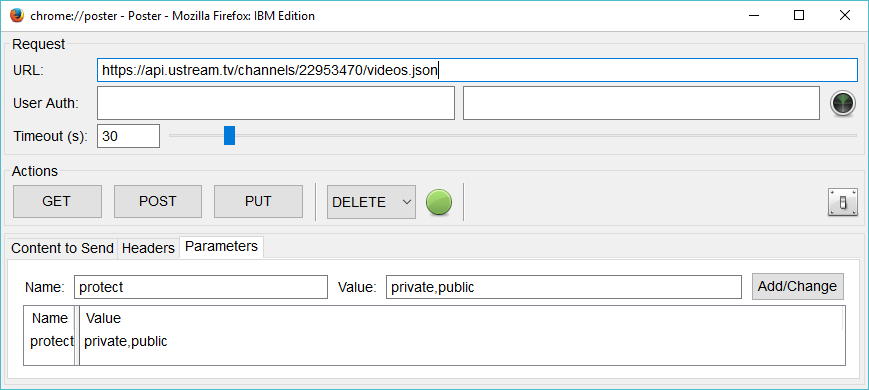
{"videos":[{"id":"102002811","title":"Blue Box","description":"Description of test video","url":"http:\/\/www.ustream.tv\/recorded\/102002811","length":"2542.85009765625","created\_at":1491935671,"custom\_created\_at":1491935671,"file\_size":"53548495","views":14,"protect":"public","thumbnail":{"default":"https:\/\/ustvstaticcdn2-a.akamaihd.net\/i\/video\/picture\/0\/1\/102\/102002\/102002811\/1\_22953470\_102002811,192x108,b,1:2.jpg"},"media\_urls":{"flv":null},"links":{"channel":{"href":"https:\/\/api.ustream.tv\/channels\/22953470.json","id":"22953470"}},"chapters":[],"tinyurl":"http:\/\/ustre.am\/:6TZAD","schedule":null,"owner":{"id":"41786059","username":"milesdavis-1","picture":"https:\/\ …

### Chrome Poster

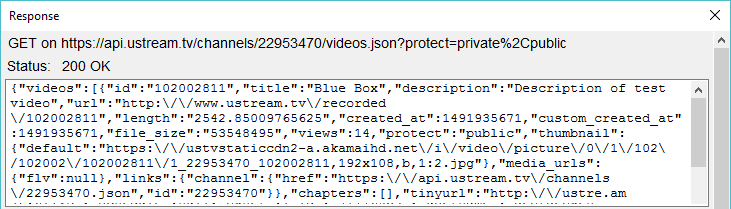
**URL and Header:**



**Query Parameters:**

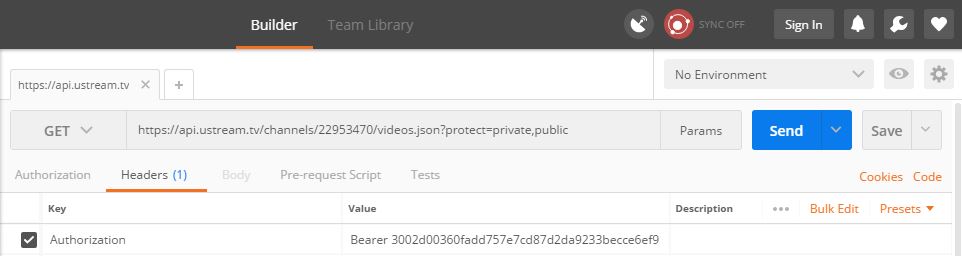


**Response**:

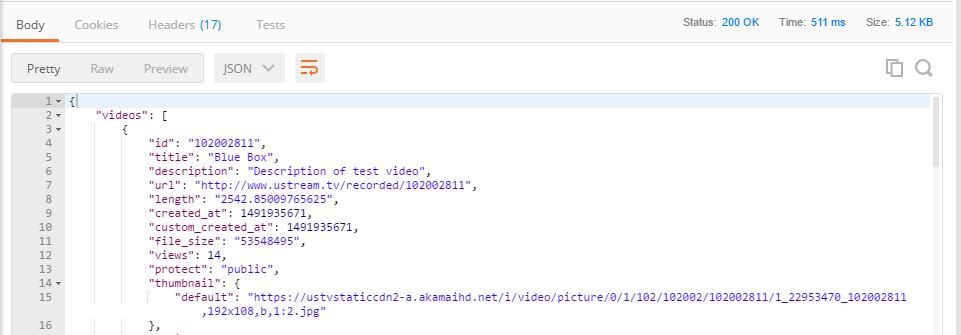


### Postman

**URL, Query Parameters and Header**



**Response**:



## Create Channel

### Curl

**URL, Headers:**

C:\...\bin>curl.exe --url "https://api.ustream.tv/users/self/channels.json" \

--header "Authorization:Bearer 712325fd1e0d281d4b93e739e762162e18556afd" \

--header "content-type:application/x-www-form-urlencoded" \

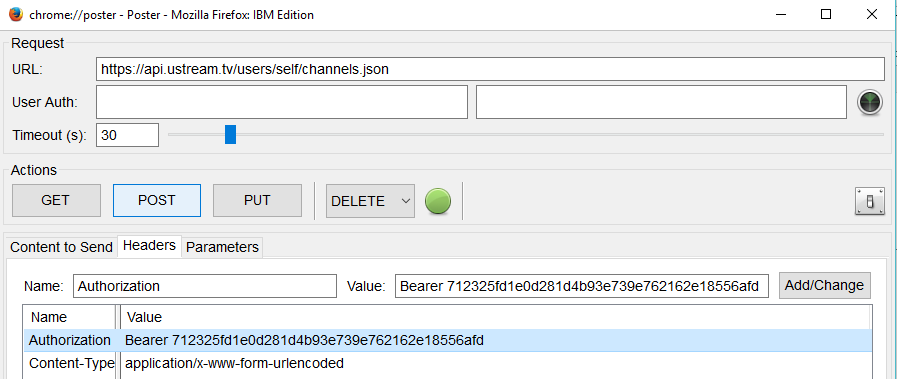
--data "title=Blockchain"

**Response**:

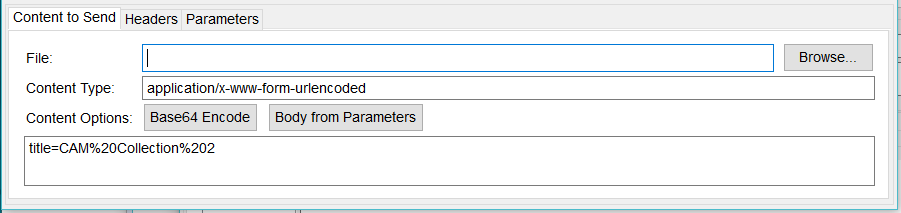
{"channel":{"id":"23294795","title":"Blockchain","url":"http:\/\/www.ustream.tv\/channel\/gMqvRYqPx39","tiny\_url":"http:\/\/ustre.am\/1zK2v"}}

### Chrome Poster

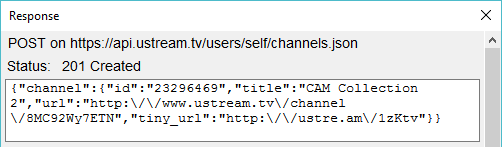
**URL and Header:**



**Content:**

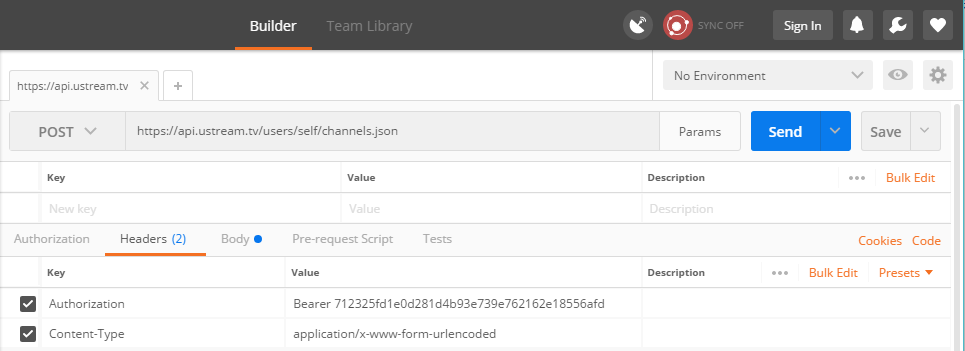


**Response:**

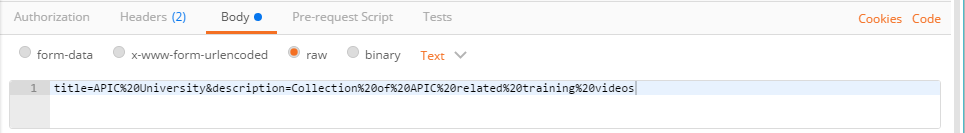
****

### Postman

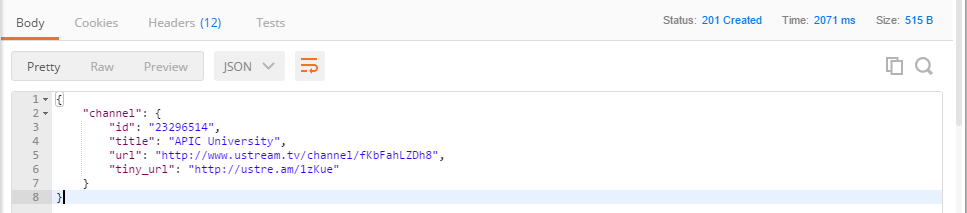
**URL and Headers:**



**Body:**

****

**Response:**



End of Workshop