Walkthrough on Writing Applications that Interact with YouTube in NodeJS

Author:
Savvas Kastanakis
2018

##
YouTube Data API Overview
##

This document is intended for developers who want to write applications that interact with YouTube.
It explains basic concepts of YouTube and of the API itself.

Before you start:

You need a Google Account(https://accounts.google.com) to access the Google Developers Console, request an API key, and register your application.

1) **Create a project** in the Google Developers Console(https://console.developers.google.com/) and obtain authorization credentials so your application can submit API requests(https://developers.google.com/youtube/registering_an_application).

- After creating your project, make sure the YouTube Data API is one of the services that your application is registered to use:
- Go to the Developers Console(https://console.developers.google.com/) and select the project that you just registered.
- Open the API Library in the Google Developers Console. If prompted, select a project or create a new one. In the list of APIs, make sure the status is ON for the YouTube Data API v3.
- If your application will use any API methods that require user authorization, read the authentication guide to learn how to implement OAuth 2.0 authorization(https://developers.google.com/youtube/v3/guides/authentication).

2) **Select a client library** to simplify your API implementation.

##################################
Node.js Quickstart
##################################

Complete the steps described in the rest of this page(https://developers.google.com/youtube/v3/quickstart/nodejs), and in about five minutes you'll have a simple Node.js command-line application that makes requests to the YouTube Data API.

##

Quota Usage and Calculation

##

1)Quota usage

- The YouTube Data API uses a quota to ensure that developers use the service as intended and do not create applications that unfairly reduce service quality or limit access for others. All API requests, including invalid requests, incur at least a one-point quota cost. You can find the quota available to your application in the Developers Console.

- Projects that enable the YouTube Data API have a default quota allocation of 1 million units per day, an amount sufficient for the overwhelming majority of our API users. Default quota, which is subject to change, helps us optimize quota allocations and scale our infrastructure in a way that is more meaningful to our API users. You can see your quota usage on the Usage tab for the API in the Google Developer's Console.

2)Calculating quota usage (https://developers.google.com/youtube/v3/getting-started#calculating-quota-usage)

- Google calculates your quota usage by assigning a cost to each request, but the cost is not the same for each request.

- Different types of operations have different quota costs.
  - A simple read operation that only retrieves the ID of each returned resource has a cost of approximately 1 unit.
  - A write operation has a cost of approximately 50 units.
  - A video upload has a cost of approximately 1600 units.
  - Read and write operations use different amounts of quota depending on the number of resource parts that each request retrieves.
    - Note that insert and update operations write data and also return a resource. So, for example, inserting a playlist has a quota cost of 50 units for the write operation plus the cost of the returned playlist resource.

- Each API resource is divided into parts. For example, a playlist resource has two parts, snippet and status, while a channel resource has six parts and a video resource has 10.
- Each part contains a group of related properties, and the groups are designed so that your application only needs to retrieve the types of data that it actually uses.

- An API request that returns resource data must specify the resource parts that the request retrieves. Each part then adds approximately 2 units to the

request's quota cost. As such, a videos.list request that only retrieves the snippet part for each video might have a cost of 3 units. However, a videos.list request that retrieves all of the parts for each resource might have a cost of around 21 quota units.

3)Quota Calculator

Calculator URL =
https://developers.google.com/youtube/v3/determine_quota_cost

This tool lets you estimate the quota cost for an API query. All API requests, including invalid requests, incur a quota cost of at least one point.

##################################

Scripts workflow

##################################

Since you have installed nodejs on your computer, you need a folder which contains the following things:

a) a "client_secret.json" to authorize your requests
b) a "scripts" folder, including all the scripts provided
c) a "Results" directory to store all files from scripts' execution

In order for the scripts to run properly, create a directory named Results and create 5 sub-directories inside:

a)Related_to_ID_Videos
b)Most_Popular_Videos
c)DataSets_perID
d)DataSets_perRegion
e)Cache_Bitmaps

You run a nodeJS script by typing:
"node script_location/script_name.js arguments"

The scripts provided in the scripts folder use the client_secret.json to authorize their requests (thus the client secret must be located to the same directory as the nodeJS executable).

Scripts:

1)authorization.js

This script obtains authorization credentials so our application can submit API requests in YouTube API.

It's a script used as a library for the rest of the scripts.

Notes:

- Authorization information is stored on the file system, so subsequent executions will not prompt for authorization.
- The authorization flow is designed for a command line application.
- For information on how to perform authorization in a web application that uses the YouTube Data API, see Using OAuth 2.0 for Web Server Applications (https://developers.google.com/youtube/v3/guides/auth/server-side-web-apps).

2)findMostPopular.js

This script finds most popular videos of a given region and writes them to a file.

Example: node scripts/findMostPopularVideos.js US 10
The above example requests the 10 most popular videos for region of US.
The result is written in Results/Most_Popular_Videos
The size of the region could be a parameter in the interval:[5, 50].

3)findRelatedVideos.js

This script finds related videos of a given video ID and writes them to a file.

Example: node scripts/findRelatedVideos.js 'GicXcgBt2bc' 50
The above example requests the 50 related videos for a given video id.
The result is written in Results/Related_to_ID_Videos
The size of the related list could be a parameter in the interval:[5, 50].

4)BFStraversal.js

This script applies BFS traversal algorithm over a single video ID.

The result is a graph with depth d and width w that contains related videos from every depth. Every new depth contains the related videos of the previous depth results.

For example, given a video id:vi, width 50 and depth 2, this

script requests D1 = 50 related videos of vi and then requests 50 related videos for each video id in D1. This results to 2550 contents directly and indirectly related to vi.

Workflow:

- Width, depth and seed ID are provided via the args list.
- The script requests w videos related to the seed ID and adds them to the graph in the order they are returned from the YouTube API.
- For each video w returned, the w' related videos of that are calculated, so on so forth, until depth d is reached, and the graph is filled with all necessary related videos.

Example: node scripts/BFStraversal.js "-9rdDeWzvsU" 50 2
The result is written in Results/DataSets_perID.

5)simulation_script.js

This script corresponds to section 4.1 in [1].

The arguments passed in this script are:

a) the seed id of our simulations (consider this id as the first video that someone chooses to watch in YouTube),
b) the width of BFStraversal
c) the depth of BFStraversal
d) the size of the recommendation list size for each simulation run (consider this list as the related contents for each video that the user is currently watching(seed id))
e) the cached contents file - this is a json file with video ids that correspond to a YouTube cache (based on this list we can decide whether a video id that a user selected to watch is a cache hit or a cache miss respectively)
f) the number of simulation steps that we want to operate (how many videos will the user watch)
g) a vector with weights that apply on each content on the recommendation list, so that we can simulate the possibility of user picking a video from that list. If uniform distribution is followed then all contents have the same probability, otherwise top-k videos are more likely to be chosen (rather than bottom-k videos)

- For a given number (sim_runs) of iterations we calculate the recommendation list(based on BFStraversal) of a given seed id.
- More specifically, we apply BFStraversal on the id that the user is currently watching (seed id) and we pick from this dataset only the cached contents (ids that can be found inside the provide cached contents list).

- These contents will be the top k videos of our recommendation list, while the rest videos to fill the list are the related videos that YouTube would offer to the user.
- The next step is to choose a random video as the new seed id for the new iteration(randomness is based on the possibility mode provided).
- If the chosen video is a cache hit we score an 1 on a bitmap size of sim_runs.
- If the chosen video is a cache miss we score an 0 on a bitmap size of sim_runs.

Example:
node scripts/simulation_script.js "fKFbnhcNnjE" 10 2 10 Results/Most_Popular_Videos/50_mostPopular_GR_\(21_Jun_2018\).json 5 "zipf1"

Accepted values for the weights mode are: zipf1, zipf2 or otherwise uniform distribution will be applied.
The result is a bitmap, written in Results/Cache_Bitmaps (this file also contains the recommendation lists and seed ids for each sim run).

6)SuperScript.js

This script requests the most popular video ids of a region and runs simulation_script over all these ids.

The arguments passed in this script are:

a) the region from which we request the most popular videos
b) the size of the region (these contents will be considered as our cache in our scenario)
c) the width of BFStraversal
d) the depth of BFStraversal
e) the number of simulation steps that we want to operate (how many videos will the user watch)
f) a vector with weights that apply on each content on the recommendation list, so that we can simulate the possibility of user picking a video from that list. If uniform distribution is followed then all contents have the same probability, otherwise top-k videos are more likely to be chosen (rather than bottom-k videos)

Example: node scripts/SuperScript.js US 50 50 2 5 "zipf1"
Results are written under the Results/Cache_Bitmaps directory.

7)DataSetProducer.js

This script finds most popular videos for a region.

Then applies BFS traversal for each most popular video, and constructs a graph (dataset) with all the produced related videos(with depth d and width w).

Example: node scripts/DataSetProducer.js US 10 20 3
The above example calculates the 10 most popular videos for US and then applies BFS on them with width = 20 and depth = 3.

8)WorldDataSetGenerator.js

This script produces the datasets for each region (executes dataset producer script for each region provided).

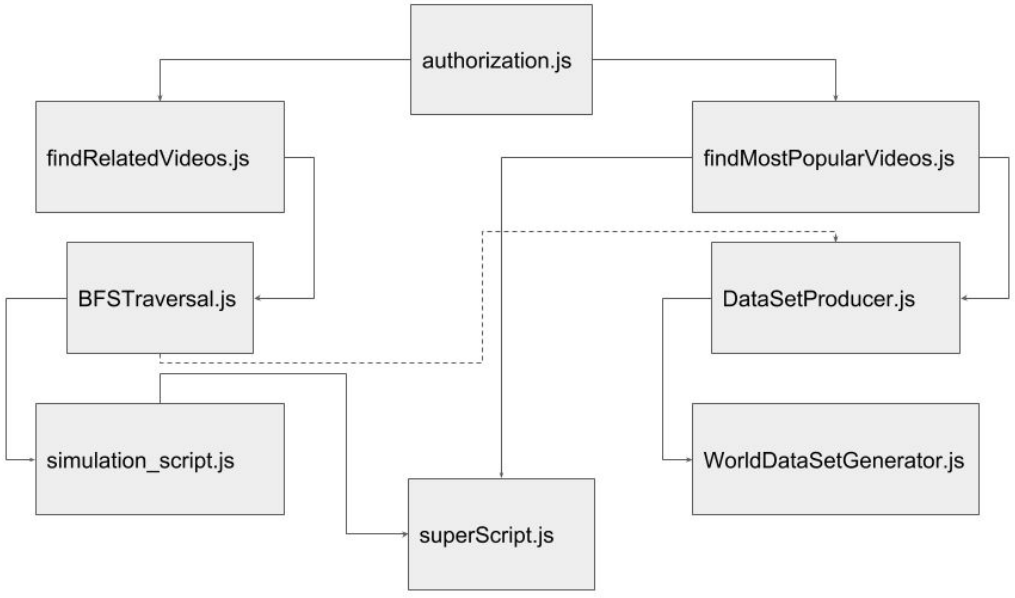
The arguments passed in this script are:

a) the regions - Regions are provided from a file in json format (e.g [GR, US, FR])
b) the size of the region
c) the width of BFStraversal
d) the depth of BFStraversal

Example: node scripts/WorldDataSetGenerator.js fileWithCountries.js 10 20 3

Note:Further explanation on the scripts is given in comments included in the source code.

Below you can find a graph with the dependencies between scripts, for better understanding of the architecture.



References

[1] Savvas Kastanakis, Pavlos Sermpezis, Vasileios Kotronis, Xenofontas Dimitropoulos, " CABaRet: Leveraging Recommendation Systems for Mobile Edge Caching", in **ACM MECOMM (SIGCOMM workshops)**, August 2018.