Brandon Swanson
Assignment 3 - pt2 CS 496
4/24/16

Table of Contents

Webpage root address:

http://swansonbassign3.appspot.com/

Test Script

See included file: APItest.py

Test Script Results

See included file: testResult.txt

List of APIs

/playlist/ or /	
GET	Returns HTML of all playlists
POST	Create a new playlist, returns JSON containing new playlist object and its new URL -title=string -creator=string (optional)

/playlist.json	
GET	returns a JSON of all playlists in an abbreviated format
/playlist/<key>/	
GET	Returns HTML of playlist matching key
POST	Create a new snippet and associate it with playlist matching key, returns a json including snippets new url -title=string -notes=string (optional) -videoID=string -startTime=integer -endTime=integer
DELETE	Delete playlist matching key and delete all snippets associated with playlist
/playlist/<key>.json	
GET	Return a JSON representation of playlist matching key
/snippet/<key>/	
GET	Returns HTML of snippet matching key
PUT	Update properties of snippet from put data, maintains original values for all values not specified, return json of new values -title=string (optional) -notes=string (optional) -startTime=integer (optional) -endTime=integer (optional)
DELETE	Deletes snippet and removes reference from parent playlist
/snippet/<key>.json	
GET	Returns a JSON encoding of snippet matching key

RESTful constraints satisfied

- Client–server - This constraint was satisfied, the client API requires no knowledge of the data storage schema, and the server can return the stored data in different representations for different clients.
- Stateless - This constraint was satisfied, There is no state for the user or the entities being stored or queried, the api calls can be made at any time from any user as long as the request is on an existing entity.
- Cacheable - This constraint was not satisfied, mostly because the product is still under development and returning fresh results on every query or api call is very important when developing.
- Layered system - This constraint was satisfied, This constraint is satisfied by the GAE platform that provides intermediary servers opaquely to the user. The application code satisfies this constraint by interfacing with the GAE distributed datastore instead of saving and retrieving information locally on whatever server receives the client's request.
- Uniform interface
    - Identification of resources - satisfied, each playlist or snippet entity has its own unique url
    - Manipulation of resources through these representations - satisfied, POST to the page creates a new playlist, POST to that playlists url creates and associates a new snippet with its own url, PUT to the snippet url will update its attributes. Playlist entities or Snippet entities can be retrieved in HTML or JSON by GET requests to their urls and removed from datastore with DELETE to their url
    - Self-descriptive messages - satisfied, content-type of returned content is always specified
    - Hypermedia as the engine of application state - satisfied, HTML representations contain hyperlinks to resource URLs

Changes made to schema from previous weeks assignment

The `Selected_Thumbnails` was removed from the planned model for playlists. The selected thumbnails were primarily included in this web pages interface in order to have a property that suited having radio buttons as a form interface. They were included with the playlist model in order to cut down on the query size needed in order to generate a page with playlist previews. In order to correctly maintain them when snippets are deleted I would need to store the snippet reference keys and their selected snippet as a structured property. In order to focus on the important functions of the API this feature was removed.

Having finished the API, what would I have done differently

When I first began developing this API I wanted to use the same URLs for GET/POST/PUT/DELETE requests on an entity from both the browser (including from form submits) and from the API calls. The difficulty here is that if a request to update an object comes from an html form then after performing the operation the user will be expecting to receive an html page, and API users will be expecting a textual response. I at first tried to differentiate by the request headers accepts flags but this was problematic because Chrome included json in its accepts headers.

One solution would be to differentiate by a /api/ prefix as some websites do. If I go that route it would be a good idea to extract functions for some of the creating/updating/deleting of entities so that html routes and api routes can be only responsible for parsing headers and sending a response, with the common functionalities handled by separate functions. These functions would be well suited to be class methods of the models, I already have class methods for retrieving entities from the datastore, returning json representations, or urls from keys.