University of Leeds **School of Computing**

COMP3011, 2023-2024

Web Services and Web Data

A RESTful API for News Aggregation

By

Tejaswa Rizyal

201484983

sc20tar@leeds.ac.uk

Date: 5 April 2024

1. Introduction

This document discusses the implementation of the coursework 1 for COMP3011. The aim as described is designing "A RESTful API for News Aggregation". I have managed to implement **all** parts of the courseworks as they were described in the assignment brief. All the requests that the server needs to process have been implemented and same with the client; all the commands that the client needs to perform and interact with the API have been implemented.

The Django server code has been uploaded to “pythonanywhere.com” under the domain “sc20tar.pythonanywhere.com” and works as expected. The Client code has been written and tested in Python 3.12.2. As asked by the assesors, the URL, Username and Password for the Admin site are as follows:

Domain: sc20tar.pythonanywhere.com

Username: ammar

Password: asdf@1234

2. The Database

The implementation of the Django Database Model was done following the steps as taught in Lecture 6 of the module. At first I had implemented a simple *Author* class and *News* class with all the fields as specified in the assignment brief. Soon this had to be updated as the coursework needed the use of *auth.models.User* functions like, *login*, *logout*, *authenticate* etc. Hence the *Author* Model was updated to inherit from the *User* Model and this rendered the fields I had created redundant as the *User* Model already has all the required fields. Although I did make a Meta class to change what name for the model is displayed in the admin site to Authors as it was showing to be Users as a result of the inheritance.

For the *News* Model, the implementation was direct. I created each field as the brief mentioned it, but did have to make a tuple for Category choices as they needed to be iterable for my implementation of filtering the news.

3. The APIs

For the implementation for the features, I will discuss their implementation individually in the following sections:

Log In: I created a *view_login()* function to process this request. In the function, the first part of the logic was to only allow POST requests, and hence if the request method was anything else, the server responds with 405: Method Not Allowed. Secondly, the server needs to authenticate the user credentials received. As mentioned in the database section, the *User* Model function *authenticate* was useful for this. I implemented this step in a try except block as in my original implementation, the server was crashing if the database did not have any user with the provided username. Hence, I put the except block in case the authenticate function fails and causes any issues, the server will respond with 401: Unauthorized. Finally

if the authentication is successful, I use the *login()* function to remember the logged in user, and the server responds with 200: OK.

Log Out: This function, after checking the request method, also checks whether the user is logged in, using the *is_authenticated* function available to us from the *User* model. If the user is not logged in the server responds with 401: Unauthorized, otherwise I call the *logout()* function in a try-except block to log the user out and respond with 200: OK. In any case, if the logout fails, the server responds with 500: Error in the except block.

Post a Story: For this feature I made the function *story()* which firstly checks the request method, and then checks if the user is logged in with the same logic as previously mentioned. Then in a try except block, it parses the payload received, if the payload is not in JSON type or the content in the payload is not appropriate, the try block fails and in the except block, the server responds with "400:Bad Request, Error in Payload". If the payload parsing is successful, a new *News()* record is created and saved, and the server responds with "200:OK".

Get Stories: As this service uses the same URL but a different request method, I put an *elif* block in the *story()* function to see if the request method is GET. If so, the function parses the payload received into the *story_cat*, *story_region* and *story_date*. If all of them are '*' then *News.objects.all()* is called to return all stories. Otherwise, I create a dictionary to store the filter values. For the date field, I implemented an extra step to convert the date received in format "dd/mm/yyyy" to "yyyy-mm-dd" as the database stores the dates in this format. The *dict* for the filter only gets appended if the filter option was provided in the payload and hence if the filter was '*' for any key, that key does not get added to the dict. This *dict* with the correct filter values then gets used as the argument for the *News.objects.filter()* function to return all the stories that are filtered. If no stories are found with the filter, the server responds with "404:No News Found". Otherwise, the news objects get appended to a list and returned as a JSON payload with "200:OK" response.

Delete Story: Firstly as this function required the key of the story to be a part of the URL, I added "*api/stories/<str:key>*" in the urls.py to have this url pattern be directed to my *delete()* function and the *key* with become an argument for the *delete()* function. Firstly the function checks the request method and if it is DELETE, it then checks if the user is logged in. The server responds appropriately for these possibilities. If so far everything has passed, the function retrieves the news record with the key and checks if the news author is the request user and if so, deletes the story. This happens in a try block and if the record retrieval fails, the server responds with "404:Story Not Found" in the except block. Also if the user is not the author of the story, the server responds with "401:Unauthorised".

All these functions have the "*@crsf_exempt*" decorator as that is needed to have these request methods. I have also put every interaction with any model in a try except blocks as, that makes sure if the retrieval fails, the server does not get any issues and instead goes to the except error blocks. This enables the server to easily interact with the database with having to always check if the list or object returned by the database is not *None*.

4. The Client

I implemented the client code simultaneously as I was doing the server side code to be able to test it. Firstly I made a while loop that only terminates when the user enters *'exit'*. Then I made a function to process user inputs and for the first command (login) the client is required to parse user input to get the URL of the new agency.

login: When the command received starts with the string "login", the function checks for the second argument, if there isn't any, the input processing function prints the syntax for the user. If a second argument is received too, the function will prompt the user to enter the username and password, and sent to a *login()* function. This function converts the domain url to the correct path with "http://" and "/api/login" and sends a post request. If the response code received is 200, the URL is stored in a session object as this was needed to implement later functions.

logout: This function just uses the session object stored to send a post request to "/api/logout" and prints the response received. In case the session object is empty, it prints an error, telling the user that they have not sent a login request yet.

post: For this command, I put another clause in the input processing function if the "*post*" command is received. It prompts the user to input a headline, category, region and details which then get put in a payload and sent to the *post_story()* function. This function then posts this payload to the URL stored in the session object with "/api/stories" appended to it. Then it prints the response received from the server

list: This function is implemented first because we need the agency ID and URLs. If the command "*list*" is entered, the *list_agencies()* function is called. Which sends a get request to 'http://newssites.pythonanywhere.com/api/directory/' and returns the data received back to the *input_processing()* function where the printing happens.

news: Similarly as before, if this command is received, the input processing function checks if there are any other arguments that the user has put in. These then get sent to a *news_switch()* function that parses the switches and if any switch is missing, it stores "*" as its key-value pair. In this function, we check if any “-id” is received, if so, we call the *list_agencies()* functions and it returns a list of all agencies which is iterated over to find the ID entered. When the corresponding URL and agency is found. The function prints the Agency Name, and passes the URL and payload to the get_news function to request and print the response appropriately. If “-id” is omitted, we call the *list_agencies()* and iterate over the list returned for 20 items and call the *get_news()* function on each iteration, getting stories and responses from 20 news agencies.

delete: Lastly, when the input processing function receives the *delete* command, it checks for the next argument, if there isn't any, it prints the syntax for the user. When the key is received, it passes the key to the delete_story function. Here, we send a delete request to the "/api/stories/<key>" URL. The response is then printed.

I implemented clauses for each of these commands that if the user makes a syntax error or enters \<command> -help, the syntax and use of the function is provided to the user. For the sessions, I also created a UserSession class to store the URL so that the client knows where to send logout, post, etc, requests. I also created a welcome section using ASCII art to print a

title and let the user know about the help command, which prints out all the available commands for the user.