1. Introduction

I have a database with complete API server code, hosted on pythonanywhere.com, it was written in Python 3.12.2 but hosted on a Python 3.10 virtual machine. The accompanying client code was also written and tested in Python 3.12.2.

The URL of my news agency is: [redacted]

2. The Database

I implemented the database as two models, one for Authors and one for News.

The "Author" model has a one-to-one relation with the default Django user, this makes it easier to link an account directly with an author name instead of separating authors from the users.

The "news" model comprises a uniquekey, headline, category, region, author, date and details. Apart from uniquekey, these are all required by the web service and directory service to have valid and robust API communication that is also descriptive of a story on a news site.

The author field is a foreignkey of the Author model, this makes it so that you cannot have an author that does not exist as a user, preserving database integrity, and the one-to-one relationship makes it easier to find more stories by a specific author, or related author information.

3. The APIs

I created the first API, HandleLoginRequest with a @csrf_exempt decorator to stop cookies being required, less work for client. All my APIs first check that they received a valid POST, GET or DELETE request. In the login API I extract the username and password from an incoming POST request, and use Django authenticate to return if the account details are actually in the database, then using Django auth login to maintain a session info to the client-side request. Reference below: (https://docs.djangoproject.com/en/5.0/topics/auth/default/#django.contrib.auth.views.LoginView)

To be able to authenticate and login, I had to use a Django sessions backend in the settings alongside the authentication middleware. For logging out I used a similar Django logout on the specific session.

My post API, used to post stories was the first to use a @login_required decorator, ensuring only verified users who have an author name can post stories. It first loads JSON data from the client request body, and extracts the headline, category, region, and details from the request, checking all are present. Once it retrieves the author obj. of the user, and the current date, it inserts a new row into the news database (db), returning an appropriate HTTP success (201) or error (503) code depending on the outcome.

The API for getting stories extracts a story category, region, and date from a URL payload, defaulting to wildcards if no value was given. It uses this information to first collate all rows in the news model, and then selectively filtering depending on the given information, with dates handled in the format YYYY/MM/DD, or the Django default.

Afterwards, it makes a JSON payload out of the results of the db filters, this payload contains all the information required to identify and describe a story. This JSON payload is then returned to the request caller (client in my case) so it could locally handle the stories.

Finally, the API for deleting a story tries to filter for news objects that match a given uniquekey, then deleting it if the caller of the command is the original author. The key is passed through the URL as an integer variable.

All the APIs are given a specific URL pattern they match against, with the API for getting stories, posting stories and deleting stories all matching to the same URL but being differentiated by the different HTTP request method used to invoke the URL.

When hosting to pythonanywhere, there was also the extra requirement of making my website URL an allowed host in the settings, otherwise I could only run on localhost.

4. The Client

For the client code, I made a Client class that stores the current URL the user is working with, and a requests library session instance. This requests session is used to attach the appropriate cookies and login data on the server-side code, allowing login permanence for specific sessions.

The login code first asks for a URL that the user wants to connect to, appending “https://” if necessary. It then constructs a plaintext payload of a username and password to POST to the server. If it receives a 200 status code, it has worked, if not, it will set the URL to none as this is used to verify locally if a user is logged in.

For the logout code it first checks if the user is logged in, and if so, removes the session data and host URL data.

When the client wants to post a story, they are prompted to enter a headline, category, region, and details. The information prior forms a JSON payload which is sent to the appropriate host URL as a POST request.

For deleting a story, the client code converts the string representation of the key to an integer and posts it to “api/stories/{key}” as a DELETE request.

The client code for the news command, was the most complicated. It first aims to take a user input and to parse/sanitise the input to see if any switches or flags such as “-id=RRB03” are used to narrow down search requests. Now the client must make a GET request to the directory service to be able to get all the active news agencies in the directory; once all news agencies are extracted, they can be queried.

From the agencies, 20 are randomly chosen, of these 20, each has a GET request sent to its own API for getting stories, if the news agency gave a false positive status response of 200, it is hidden. If a true positive status response of 200 is received by the client, it formats all the appropriate information passed back from the news agency into an easily understandable explanation of its stories. If there is an error trying to retrieve any story, if a news agency has no stories at all, this is shown as a clearly defined error message, alongside the error code.

The client code for listing all agencies uses the same directory service with a GET request to obtain all active news agencies, 20 of which are randomly selected and then displayed in a formatted approach, albeit not in the same way as the news command.

The invocation of all these commands is in the main function which just has an infinite loop asking for one of the listed commands, separating extra information if necessary, and finally informing the user if they used a bad command.

There is a hidden command for registering your service, but this will do nothing as the details of the registration have to be manually set within the source code and cannot be changed in the client code.