Flickr Uploadr

Antero Duarte
40211946@live.napier.ac.uk
Edinburgh Napier University - Advanced Web Technologies (SET09183)

Abstract

Flickr Uploadr is an application that allows users of the photography oriented platform Flickr to schedule photos to be uploaded at a later date. The users conenct their flickr account to the app and can schedule photos to be uploaded and can change the title, description and tags of a photo.

Keywords – Advanced, Web, Technologies, Python, Flask, Flickr, Scheduler

1 Introduction

Flickr , like many social media platforms, prioritises content from users who post regularly. This means that the same user, with the same following and a comparable quality photo is more likely to get more exposure and better reception for a photo that is posted in the 10th day of a 10 day streak of posting 1 or more photos per day, than one posted on the first day.

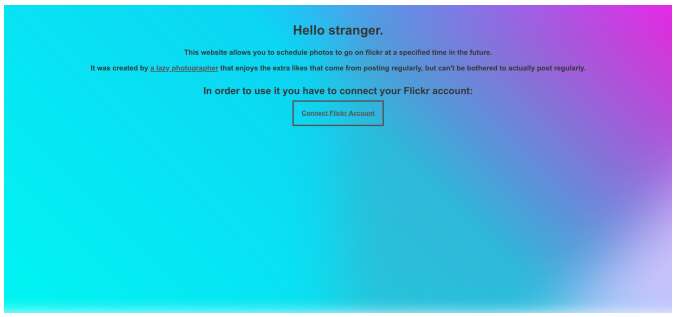

Figure 1: **Flickr Uploadr** - Home page

This application, by allowing users to schedule posts for later days, allows people who see photography as a hobby to enjoy the same level of exposure within the Flickr platform. Fot photography professionals, it allows them to batch edit photos and not have to worry about being consistent with their posts and focus more on their actual work.

Flickr Uploadr uses the Flickr REST API [1] to upload photos as the logged in user. It uses a microsevice oriented architecture and decouples the frontend from the backend [2].

2 Design

This web app was designed in line with modern standards and best practices.

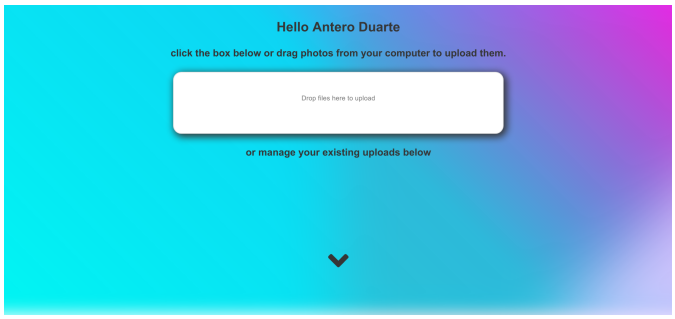

Figure 2: **Flickr Uploadr** - App home

2.1 Architectural Overview

The backend uses python Flask [3] for the web related activities, an embedded SQLite[4] database to store user information and uploading jobs (an uploading job is a photo and all of the associated settings needed to upload said photo to Flickr), and the python threading module to run a separate Thread that processes the uploading jobs.

The frontend uses a frontend javascript framework called Vue.js [5]. It is a framework that is increasing in popularity and is considered one of the top 3 javascript frontend SPA frameworks, alongside React.js [6] and Angular.js [7]. In my opinion, and the reason why I like it is that it stands exactly in the middle between the other two. Where Angular is too enterprise focused with little room for customisation in terms of code organisation and design patterns in favor of corporate-oriented standardisation, and React.js makes no assumptions and offers little in the way of suggesting best practice, leaving that for the developer to deal with. Vue.js offers a standardised way of organising code, by providing single-file components [8], which include markup, logic and presentation in the same file, but at the same time, it can be used in more complex applications, with state [9] and route [10] management or simpler applications as a substitute for older DOM manipulation based solutions like jQuery [11]

2.2 Database and data modelling

As mentioned before, the database is a SQLite presisted instance that is accessed from the code through the

Peewee Object Relational Mapper [12]. The database schema is as follows:

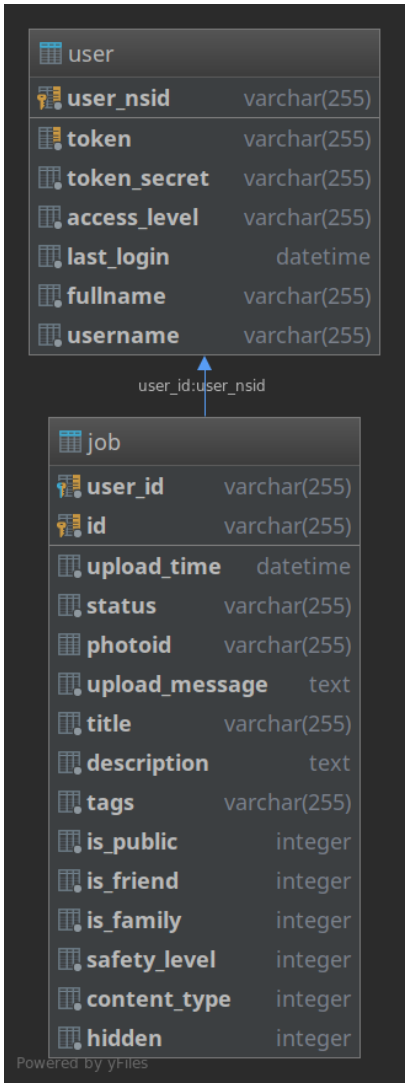

Figure 3: **Database Schema**

Figure 3 shows 2 main tables, user and job. Peewee recommends that auto-incrementing IDs are used as primary keys for the database tables, but since this is still widely debated and a divisive issue [13] I decided to go with what we learned in the 2nd year Database model, that when natural primary keys exist, they should be used in favor of surrogate keys.

Although this is an advantage in terms of data design, it introduced a workaround, since peewee supports the opposing view.

```
1         class BaseModel(Model):
2             # We override save because we use self managed↩
      primary keys, and peewee docs
3             # say that we must always call it with ↩
    force_insert=True when that is the case
4             # So I'm expecting this to save countless ↩
    hours of debugging
5             def save(self, *args, **kwargs):
6                 if "force_insert" not in kwargs:
7                     kwargs["force_insert"] = True
8                 return super(BaseModel, self).save(*args, ↩
    **kwargs)
9
```

Listing 1: Peewee save function workaround

The code excerpt at 1 showsthea workaround for peewee's default save function. As explained in the documentation [14], self managed primary keys need the argument `force_insert` to be true for every save.

2.3 Authentication/Authorisation

2.4 Decoupled frontend

2.4.1 URL Structure

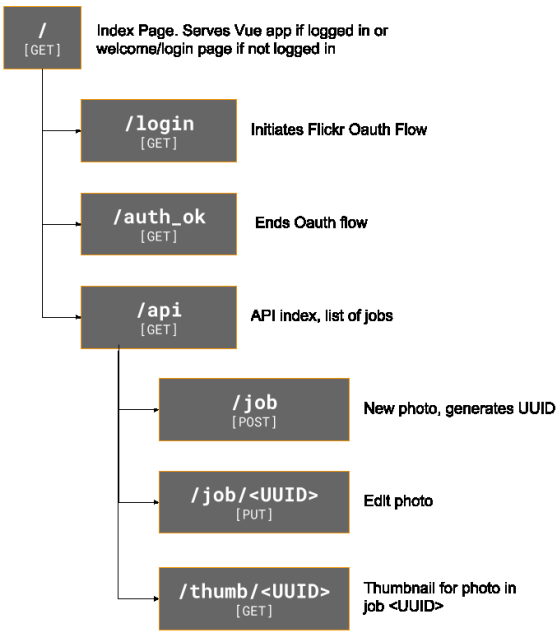

Figure 4: **URL map**

2.4.2 Navigation map

As for navigation, the application presents the map in figure 5.

One extra that is not present in the maps is that when a release page is accessed by clicking a track name, said track is highlighted in the release page. This happens because clicking on a track does not make it obvious that the user will navigate to the release page.

3 Enhancements

"If it ain't broke, don't fix it. Do improve it though..."

There are several things I would have done if I had more time to develop this project. A lot of the data that I gathered, I ended up not using, or at least in the way I intended to use it.

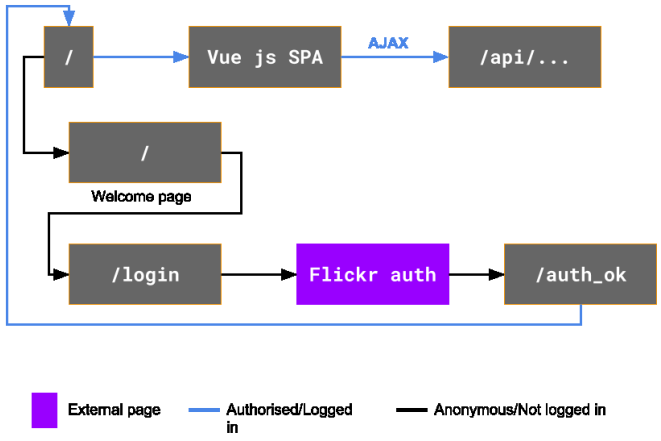

Figure 5: **Navigation map**

3.1 Track similarity

An example of this can be seen in a comment in the model.py file:

This function was meant to return the euclidean distance between 2 tracks that are placed on an 8th dimensional plane. It's a very rough attempt at creating a track similarity function. Had I had more time, I would have implemented this function in suggesting tracks to add to the setlist, based on the tracks that a user added previously.

3.2 Setlist flow

An idea I had since the start was to measure a setlist's flow.

By flow here, I mean how one song progresses into the next one. Key changes, beats per minute, loudness and other acoustic metrics that I have access to in the data could be used to calculate how easily one can move from one song to the other. Other data could also be gathered to try to make the flow calculations better.

For example, songs in different tunings sometimes mean that artists have to switch instruments or tune them differently, which breaks the flow in a live concert scenario. Songs with different instruments than the previous one might mean that a band member has to switch instruments or come on/off stage. Songs with a low dynamic complexity (low variation between calm/agitated periods) and high bpm, or high danceability and bpm, might make the band tired after playing a lot of them in a row, which means they would need to rest between songs.

These and other conditions could be made into a program that tries to leverage the elements of different songs to keep a crowd's engagement.

3.3 Data collection

Finally, a feature that could be extremely useful in growing an app like this one would be to collect data about how people are using the website. This could become a big advantage when developing new functionality, and could also enable machine learning based features, if said data was used to train models.

3.4 Dedicated setlist page

When creating this application, the initial idea involved a page dedicated to managing the songs in the current setlist. This idea was dropped when due to time constraints I realised that I would not be able to create the necessary mechanisms to have enough functionality that makes it worth having such a page.

The setlist page would include the previously mentioned track suggestion feature, a track reordering (using drag and drop) and overall setlist metrics, such as total running time, and possibly even a dashboard type interface that would show the characteristics of the current playlist.

3.5 Dedicated track page

Similarly to the setlist page, the amount of effort needed to create a track page, when compared to the functionality that would be in that page, just did not justify its creation.

3.6 Direct connection to data APIs

All of the data collection process could have been avoided if a direct connection was made to the APIs offered by both data sources. This would have opened up a lot of possibilities, considering that the artist catalogue is quite limited, but on the other hand, it would have made the range of different outcomes too unwieldy, which would have slowed development because the code would have to accommodate for gaps in the data that were filled during the data gathering stage.

4 Crtitical evaluation

While I'm happy with the current state of Set.lystr, several things about it make me want to spend more time on it, even if just for fun/portfolio building.

4.1 Architecture

One of the things I enjoy the most about developing projects such as this one is to look at the holistic system architecture. While being a relatively simple project that is composed of a single codebase, it's architecture is also rather simplistic. While there are things about this project that might not pass the scalability test, for example not splitting different routes into different files, I personally consider this a good architecture when taking into account the wider context. This application and its development have a scope, and one of the best skills when thinking about the architecture of an application is to take into account the balance of time vs features.

I could have made this application more scalable. I could have decoupled the frontend from the backend, which would mean less work on the server side. I could have deployed n instances of the application and load balance them. But calling that overkill would be an understatement.

That being said, I am particularly happy about features such as the streaming search results function, which is backed by a generator function that yields results as they become available. This improves the perceived loading times, and when matched with client-side functionality that sorts the results as they come, creates a good User Experience.

Another example that I am particularly happy with is the balance between AJAX functionality for improved user experience and the fall-backs that are in place for the case where a user is not running javascript. While it could be argued that users whose browsers don't support javascript would not benefit from the modern CSS in the page, having these fall-back capabilities means that even with a broken layout and a few ugly pages (I'm looking at you, default Flask redirect-after-form-submit page), a user is fully capable of using the system. This is also true for screen readers, which makes the web application more accessible.

4.2 Responsiveness

While responsive to a point, this web application still has some pages that are not optimised for mobile view. One of the worse elements being the setlist, which should be made to take the whole screen when on mobile views.

This not only makes the website less appealing, but also harder to interact with when on mobile devices. This becomes a major problem, when considering that upwards of 60% of the web's traffic according to statistics from April 2018, are from mobile devices.

4.2.1 Sub par search results

When searching for a set of keywords, a user can clearly see that the search results are just not up to the standard that a user has come to expect when using a website. And while this is partly due to users being "spoiled" by great user experiences, such as the ones that big companies/products provide, the truth is that the technology for achieving such functionality is more accessible than ever. Besides dedicated search engines that index all pages in a website, there are libraries that could be used to improve string comparison, which in itself would have made search results better.

While string comparison using the naive python builtin *SequenceMatcher* stems from a limitation imposed as part of the coursework, of not being able to install additional libraries, if I had put some more effort into the search functionality, better results would have been possible. For example, for the cases where an artist has a self-titled release (which is very common, e.g: Pearl Jam by Pearl Jam), due to the text index being just a dictionary of *string* to *string*, whatever comes first is overridden by what comes after, which means searching for Pearl Jam would never return the artist, when more often than not, that is the desired outcome.

This could be fixed quite easily by just making the text index a dictionary of *string* to *list of string*, but I decided that this would come on a following iteration of the development, due to time constraints.

5 Personal Evaluation

Web development is an area of computing that has always fascinated me. Partly because of how easy it is nowadays to create something that can be accessible by the whole world, but also because it can be quite challenging to adhere to living standards and create a web application that follows all best practices in the industry. That being said, I work as a part-time web developer, so a lot of these best-practices are ingrained in the way I develop a project such as this one.

This can be a blessing and a curse. It is very good to know my way around a codebase and to be able to understand the basics of a new framework in a programming language that I have barely used before in a matter of minutes, as was the case with Flask, simply because I understand the underlying concepts that are common to most web applications. But this brings with it a sense of perfectionism and always trying to push myself past the next boundary that can work against me. I am guilty of having developed coursework applications that go beyond demonstrating that I am capable of achieving the level sought after by the marker, simply because that is the bar I have set for myself and my work outside of university.

In this coursework in particular, I would say that I was able to strike a balance between achieving the learning outcomes proposed in the coursework description and achieving a level of self-fulfilment that comes from doing the best I can. As mentioned in the enhancements section3, I realise that there are things I could have made prettier or more functional, or better in general. But I also realise that I introduced challenges for myself that were out of the scope of the coursework but that I decided to do in order to improve my abilities (for example, not using frontend libraries).

In the end, I am happy about where I got to because I can see a clear path to improve the things I am not completely happy about with this coursework. I was able to manage the whole project from start to end, balance that with the rest of my university workload and a particularly busy period at my job.

References

[1] “Flickr app garden - rest api.”

[2] “Microservices and spas,” Feb 2017.

[3] “Flask (a python microframework).”

[4] “Sqlite.”

[5] “Vue.js.”

[6] “React.js.”

[7] “Angular.js.”

[8] “Single file components.”

[9] Vue.js, “Vuex.”

[10] Vue.js, “Vue router.”

[11] “Replacing jquery with vue.js,” Feb 2018.

[12] “Peewee orm.”

[13] “Choosing a primary key: Natural or surrogate?.”

[14] “Models and fields — peewee 3.7.1 documentation.”

6 Appendices

6.1 Data gathering code