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 connect 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 [1] , 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, than one posted on the first day.

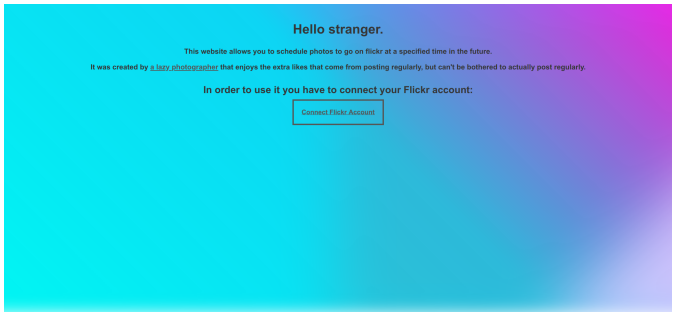

Figure 1: **Flickr Uploadr** - Home page

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

Flickr Uploadr uses the Flickr REST API [2] to upload photos as the logged in user. It uses a microsevice-like architecture and decouples the frontend from the backend [3].

2 Design

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

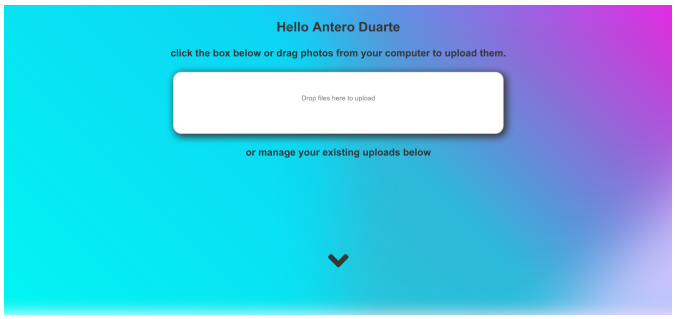

Figure 2: **Flickr Uploadr** - App home

2.1 Architectural Overview

The backend uses python Flask [4] for the web server activities, an embedded SQLite[5] database to store user information and uploading jobs [1].

The frontend uses Vue.js [6], a javascript framework. It is a framework that is increasing in popularity and is considered one of the top 3 javascript frontend SPA frameworks, alongside React.js [7] and Angular.js [8]. 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 [9], 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 [10] and route [11] management or simpler applications as a substitute for older Document Object Model manipulation based solutions like jQuery [12].

2.2 Database and data modelling

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

[1] An uploading job is a photo and all of the associated settings needed to upload said photo to Flickr

the Peewee Object Relational Mapper [13]. The database schema is as seen in figure 3.

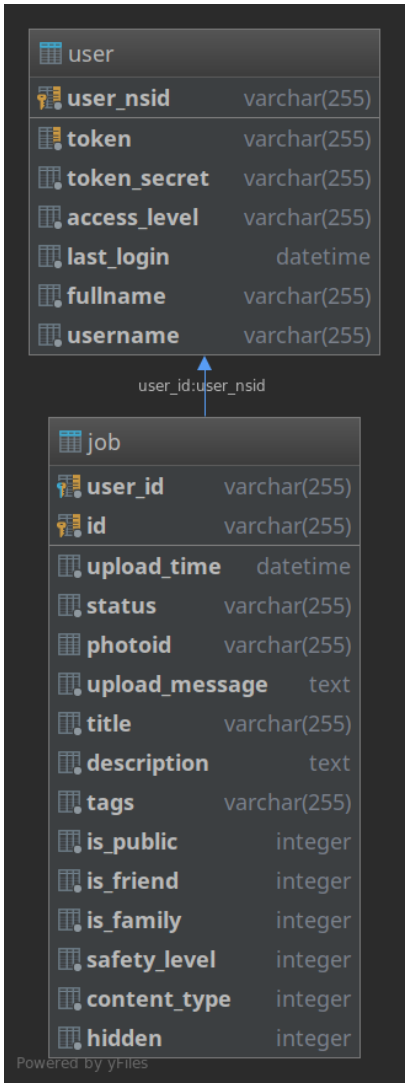

Figure 3: **Database Schema**

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 there is no canonical answer to this question [14], I decided to go with what we learned in the 2nd year Database module, 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 1 shows the workaround for peewee's default save function. As explained in the documentation [15], self managed primary keys need the argument `force_insert` to be set to `True` on every save.

2.3 Authentication/Authorisation

This application extends Flickr, which means users need a Flickr account to use the application. Flickr provides an Oauth [16] server as part of their API [17], which means that the amount of data this application needs to keep about a user is minimal. It also takes the pressure away from this app to manage users and passwords. The application links a user account to a Flickr `user_nsid`, which is Flickr's unique identifier for users. It also stores an Oauth Token, so it can upload photos as the user when scheduled, even if the user is offline. Although it is generally discouraged to store access tokens, it is the only option when scheduling activities on behalf of the user.

2.4 Decoupled frontend

As mentioned before, this application uses Vue.js to provide a modern, feature-rich interface that connects to the server using a RESTful API [18]. Vue.js is a reactive library, which means that changes to the application state will trigger components that receive the state to re-render. In practice this means that a developer does not have to manually listen to events in the DOM[2], to trigger changes to the DOM[2]. This enables faster development and less time debugging/dealing with DOM[2] quirks.

2.5 Application functionality

Upon logging in, a user can upload images to the server by using the file drop box, provided by `vue-dropzone`[19], which is a plugin that implements Dropzone.js as a Vue component. Uploaded images automatically generate a job ID (v4 UUID[20]) and the job fields are set to their defaults (`title='name_of_the_uploaded_file'`, `description=''`, `tags=''`, `upload_time=current_time + 1 day`). After creating, a job's status is set to `WAITING`, meaning it is on the queue to be processed shortly after `current_time > job.upload_time` (See figure 4).
Shortly after, because there are no guarantees that a job will run exactly on the specified time, for example, if there's a queue of jobs all set to run at 5pm, they will run in the order they come from the database, presumably, row creation time, which essentially means first come, first served.

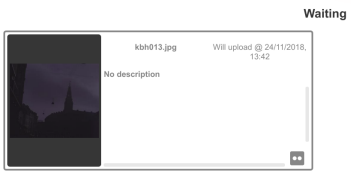

Figure 4: **Jobs waiting to be processed**

A job can be edited, so a user can update the photo's title on Flickr, the photo's description on Flickr, the

[2]Document Object Model

photo's tags on Flickr, and the uploading time (See figure 5).

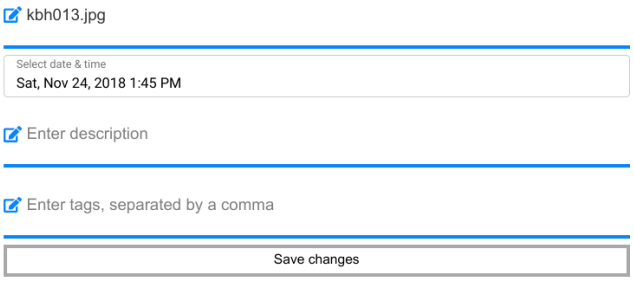

Figure 5: **Jobs editing form**

The uploading time is displayed in the form of a calendar, where past dates are blocked (See figure 6). As shown in the screenshot, taken on the 23rd, days 23 and before are blocked. The reasoning behind blocking the 23rd as well stems from a belief that people using a scheduler should not be putting pressure on the server for immediate/almost immediate uploads.

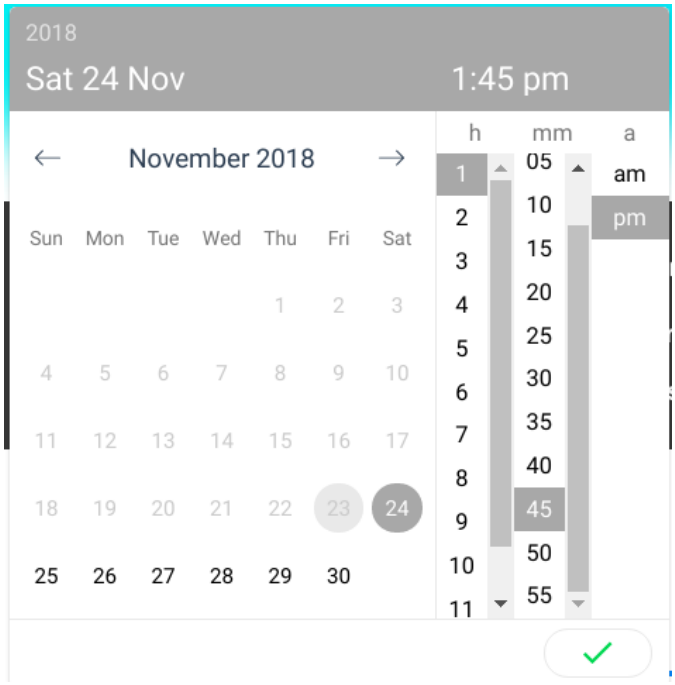

Figure 6: **Calendar**

This means that a user posting a job for now + 5 minutes is essentially shifting the effort on to the server, rather than waiting 5 minutes and posting the photo manually on Flickr. This is not a hard constraint, and is not checked on the server side, but by introducing an element of friction on the UI [21], it discourages users from doing it[3]. The server will not allow past dates though, validating input to check if `upload_time < now`.

[3] In fact, only a user with knowledge of programming or at least HTTP/REST could manipulate the request being sent to set a photo to be uploaded on the same day

Once the upload time comes for a photo, it is uploaded to flickr and it's job status updated to `COMPLETED`. There are also intermediate stages between `WAITING` and `COMPLETED`. During the upload a job is marked as `UPLOADING`, and if there's an error, the job is marked as `ERRORED`.

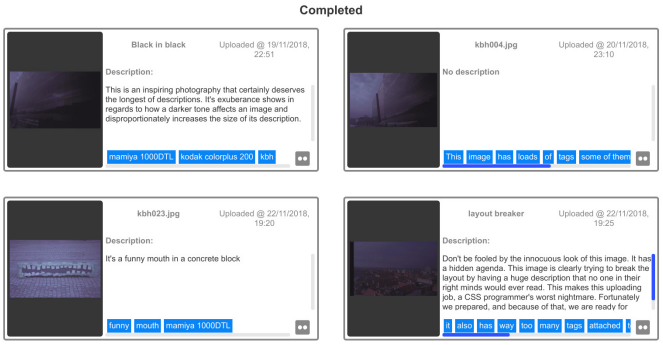

Figure 7: **Completed Jobs**

2.6 URL Structure

2.6.1 URL Map

Since this application is based on a REST API, the URL structure is quite simple, as can be seen in Figure 8. The only page generated server-side is the welcome page. All other routes are either redirects in or out of the oauth flow, or API endpoints that serve data for the Vue frontend to render.

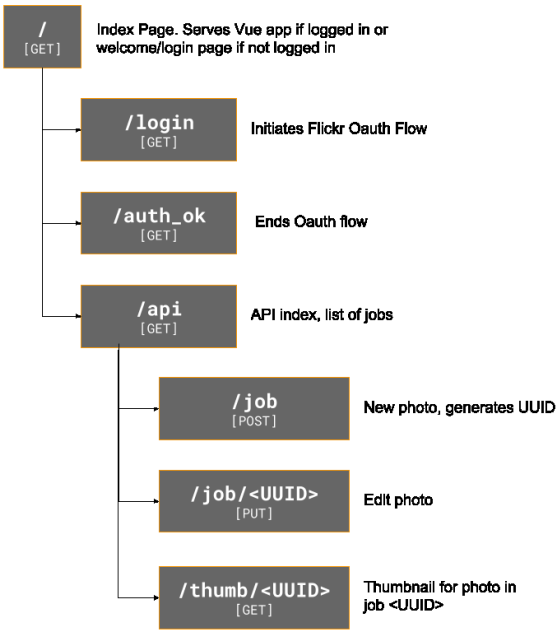

Figure 8: **URL map**

2.6.2 Navigation map

As for navigation, the application presents the map in figure 9.

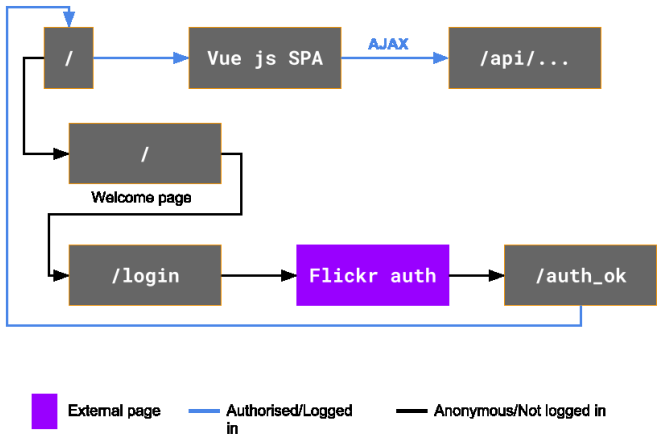

Figure 9: **Navigation map**

Note the navigation around the Login flow, which as per the oauth spec [16], sends a user to Flickr to authenticate (figure 10[4]), then send the user to a consent page (figure 11) which authorises the application to perform activities on behalf of the user, and then back to the application, which receives the necessary parameters to request an access token via url query parameters, and finally redirects the user to the application home page, where the application already has access to user data, as seen before on figure 2, where the user's Flickr full name is shown.

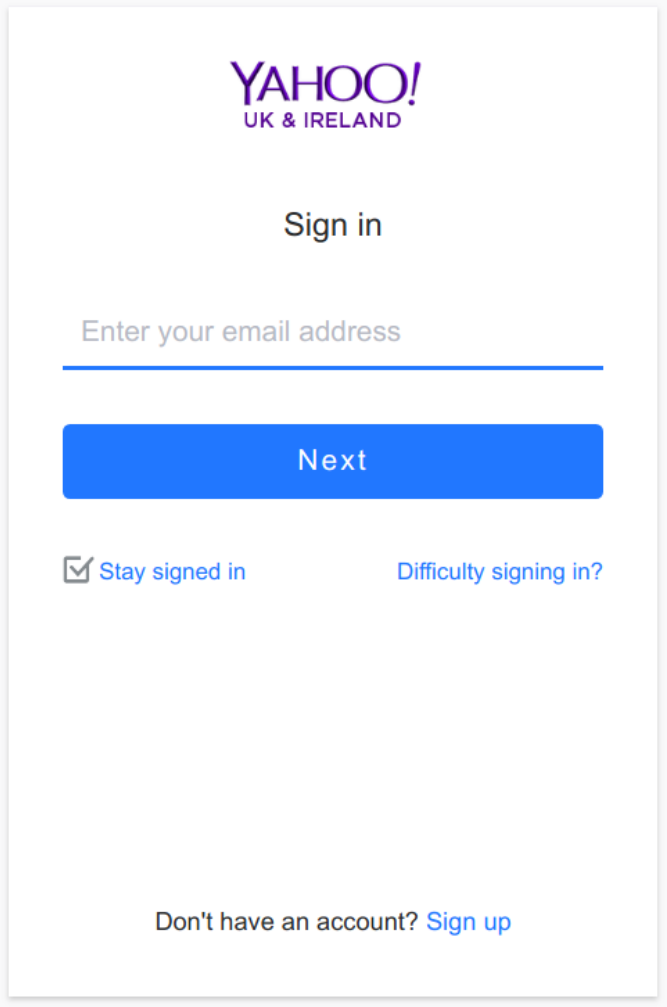

Figure 10: **Oauth flow** - Flickr/yahoo login[4]

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. Although the initial idea/functionality is all covered, a few more elements would greatly improve this application.

3.1 Tiered user system

As is, all users are the same to the application, but during development, I had the idea to create a tiered user system, where different usage tiers would have different privileges. For example, there is no limit to the amount of photos a user can upload, but to make this application feasible and free to use, there would have to be restrictions on the amount of images a user can post so the disk usage on the server doesn't grow endlessly. One of the user tiers, could have the user pay for unlimited or a higher cap on storage. This would mean the paid users would be paying for the server/hosting, while free users could still enjoy some of the benefits of the system.

Introducing a cap on storage would mean that the application would have to keep track of how much disk space a user is currently using, which it doesn't at the time of writing.

3.2 Plugable storage back-ends

An alternative (or possibly something that could run alongside) the paid tiered user model is to have multiple storage backends that could be connected using SDKs and APIs. It might also bring piece of mind for more security aware users who don't want to trust their images to some random server on the internet. Services like Dropbox [0], Amazon S3 [0] and Google Drive [0] offer APIs to store and retrieve files. If the application offered an alternative to disk storage, the running costs could greatly be reduced, meaning the service could stay free of charge, case the user decided to store their photos on their own storage. Then Flickr Uploadr acts merely as a vehicle between a user's personal storage and their Flickr account.

3.3 Sneaky Flickr Uploadr tag

Another idea to raise funds to run the application was to introduce a `Flickr Uploadr` tag on every photo uploaded through the application. Users would then be given an

[4] At the time of writing, flickr was in the process of transitioning from being owned by Yahoo and using Yahoo's login system, which is why the flickr login screen is a yahoo login screen.

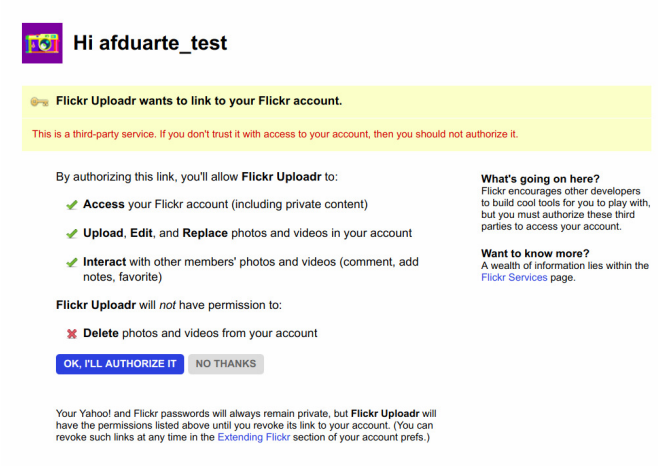

Figure 11: **Oauth flow** - Flickr consent page

option to donate towards the running costs of the server in exchange for not having this tag placed on their photos. This is obviously something that might slightly annoy users and a tag is easily removed by a user on Flickr, but the simple fact that a prompt shows up, might raise awareness that even though the service is free, it incurs in development, maintenance and hosting costs for the developer, and donating something towards the running costs could help the service stay up.

3.4 Microservices

While this application follows some rules of microservice-oriented architecture, it could be further broken down, which would mean it would be easier to scale. For example both the thumbnail generation and job uploading functions could be implemented using a message queue worker pattern [0], making the bulk of the application way more scalable and leaving only a simple static HTTP server to serve the frontend and a minimal Web Server to turn API calls into messages on the queue and retrieve completed jobs. This could greatly improve the performance of the application on scale, and would mean that more users could be using the application at the same time.

3.5 Smarter scheduling

Right now, there is a serious risk that everyone schedules photos to be posted at 9am on the 21st of December to mark the start of the winter solstice, and with enough users, this could just overload the server and severely delay or even error some photos being posted. With a smarter scheduling algorithm, some photo uploads could be delayed or anticipated to reduce the stress on the server and the network at that specific time. Again, this could be a feature of donating users, to schedule an exact time, and give free users a window of 30 minutes/1 hour when their photo would be posted

3.6 Deeper connection to flickr

This application could offer a much bigger connection to flickr. For example, as a user posts photos, the application could track which times are better to post photos, by measuring the number of views/likes that photos get when posted at different times. This could go even further and compare a users' average like/view gain/loss to other users and suggest better times to upload photos, based on the photo's reception.

4 Critical evaluation

This project is not exactly what I imagined. While certainly the functionality is all there, there are a lot of rough edges. The fact that I chose to build the frontend as a different codebase might have penalised my work in other parts of the project. Since I had to deal with 2 codebases, I feel that some of the work might feel rather unpolished.

4.1 The API

While functional for adding/editing/getting, I feel like the API exposed from this project could have been a bit more fleshed out. There are a few inconsistencies in field names, field types (e.g: uploading time is a string when GETing, but must be a Unix timestamp integer when PUTing). This was a problem I started having when trying to validate on the server side. Essentially it means that I should have planned for a lack of a canonical standard date format, hence Python and Javascript have to translate dates for each other. There are also missing features that are quite obvious, not hard to implement and I just ran out of time, like a `DELETE /job/<id>` endpoint.

4.2 The frontend

Although I'm happy with the code quality in the frontend codebase, I feel like the design feels a bit unpolished. It is not bad, but there are a few elements that seem odd and out of place. I had a few ideas, and asked for help from a proper designer, but like many other things in this project, in the end, just ran out of time. One idea that could turn the frontend upside down was to present uploading jobs in a timeline, rather than just listing them. This could be visually appealing and if done correctly, very functional, as this application hangs around 2 things: photography and time.

4.3 Code quality

I pride myself in striving for quality code. I always try to code for myself in 2 years, having looked at 2 year old code and cringing at how bad it used to be. I obviously don't know everything and expect to still cringe in 2 years time. The point being, there are things about this project that I am not happy about in terms of code quality. For example, when using the threading module and running the job thread, I think it should have been implemented in a way that the job thread knows when the main application thread is stopped, because when sending a SIGINT to the main program, the threading modules throws an Exception from the join function.

4.4 The good bits

Ultimately, I made it work. The application is a proof of concept, but it works. I will be using it, personally while

I rebuild it in a way I feel I could present it to the world. The good side of having decoupled the frontend is that I can completely rewrite the backend while using the same frontend, and vice-versa.

5 Personal Evaluation

As evidenced through a very long enhancements section (section 3), and a complete critical evaluation, there are a lot of things I wish I could have done with this project, so in a way it feels incomplete. But if I must be honest with myself, I should look at the complexities and intricacies of this project.
Apart from knowing the platform I was interfacing with really well (which admitedly I already did, being a user), I had to understand how the API works, I had to find the right libraries for the project, gain a deeper understanding of the Oauth flow, and then re-learn it since Flickr doesn't implement it exactly like the spec, only to then realise there is a library that does it for me, better than I was doing it at that point. Ups and downs, failures and successes, but such is the life of a developer.

5.1 The veredict

In the end I am happy, but not satisfied with the current status of the project and hope to be able to continue work on it. I plan to either rework the parts that I'm not very happy with, or actually re-design the whole application taking into consideration everything mentioned before in the enhancements section. One interesting idea is to try to implement the project as a serverless application in a FaaS (function as a service) platform. Ultimately this project shows a lot of promise, but because of some aspects, it doesn't completely deliver. I am commited to keep working on it and make it a full-fledged application.

References

[1] "Flickr." https://www.flickr.com/about.

[2] "Flickr app garden - rest api." https://www.flickr.com/services/api/.

[3] SitePen, "Microservices and spas." https://www.sitepen.com/blog/2017/02/20/microservices-and-spas/, Feb 2017.

[4] "Flask (a python microframework)." http://flask.pocoo.org/.

[5] "Sqlite." https://www.sqlite.org/index.html.

[6] "Vue.js." https://vuejs.org/.

[7] "React.js." https://reactjs.org/index.html.

[8] "Angular.js." https://angular.io/.

[9] Vue.js, "Single file components." https://vuejs.org/v2/guide/single-file-components.html.

[10] Vue.js, "Vuex (centralized state management for vue.js)." https://vuex.vuejs.org/.

[11] Vue.js, "Vue router (the official router for vue.js)." https://router.vuejs.org/.

[12] SmashingMagazine, "Replacing jquery with vue.js." https://www.smashingmagazine.com/2018/02/jquery-vue-javascript/, Feb 2018.

[13] "Peewee orm." http://docs.peewee-orm.com/en/latest/.

[14] agiledata.org, "Choosing a primary key: Natural or surrogate?." http://www.agiledata.org/essays/keys.html.

[15] "Models and fields — peewee 3.7.1 documentation." http://docs.peewee-orm.com/en/latest/peewee/models.html#id4.

[16] oauth.net, "Oauth community site." https://oauth.net/.

[17] "Flickr app garden - oauth." https://www.flickr.com/services/api/auth.oauth.html.

[18] SearchMicroservices, "What is a restful api?." https://searchmicroservices.techtarget.com/definition/RESTful-API.

[19] rowanwins, "vue-dropzone - a vue component for file uploads, powered by dropzone.js.." https://github.com/rowanwins/vue-dropzone.

[20] I. T. Union, "Universally unique identifiers (uuids)." https://www.itu.int/en/ITU-T/asn1/Pages/UUID/uuids.aspx.

[21] S. Brykman, "The benefits of ui/ux friction$_2$016.", Sep 2016.

"Dropbox." https://www.dropbox.com/.

I. Amazon Web Services, "Cloud object storage | store retrieve data anywhere | amazon simple storage service." =https://aws.amazon.com/s3/.

"Google drive." www.google.com/drive/.

"Message queues background processing and the end of the monolithic app." https://blog.heroku.com/end_monolithic_app.