CS CAPSTONE TECHNOLOGY REVIEW

JANUARY 10, 2017

AEROSOL ANALYZER MOBILE WEB APPLICATION

PREPARED FOR

NASA JPL

KIM WHITEHALL, LEWIS MCGIBBNEY

PREPARED BY

GROUP 22

AEROLYZER

E. REILLY COLLINS

SOPHIA LIU

JESSE HANSON

Abstract

Aerolyzer is a mobile web application capable of processing visible images and inferring atmospheric phenomena to provide the general public with near-real time monitoring of aerosol conditions. This document outlines the software design descriptions for the Aerolyzer mobile web application and acts as a representation of a software design to be used for communicating design information to stakeholders. A written description of the Aerolyzer software product is provided.

CONTENTS

List of Tables

I Introduction

Aerolyzer is a mobile web application capable of processing visible images and inferring atmospheric phenomena to provide the general public with near-real time monitoring of aerosol conditions. In this document, we will be focused on determining which technologies will be used throughout the duration of this project. The functions for which we need technologies are as follows: Coding environment, image and metadata storage, extract EXIF metadata, web interface, getting meteorological data, testing, upload photos, getting geo-related data, documentation/development of API documents.

E. Reilly Collins: Technologies 1, 2, and 3
Sophia Liu: Technologies 4, 5, and 6
Jesse Hanson: Technologies 7, 8, and 9

II Coding Environment

II.1 Available Technologies

Since our team will be using Python to implement the Aerolyzer web application, we need a style guide that is Python-specific. Three such existing technologies that would accomplish this aim are Pylint, PyChecker, and Pyflakes. These tools search for errors in Python code in order to enforce a coding standard. [1] They define a default coding style and provide refactoring suggestions while avoiding false positives and attempting to detect ”dangerous” code blocks. After analyzing the Python code, the Python code checkers offer feedback for improving the code syntax and/or style.

II.2 Goals

With regards to our coding environment, we are mainly striving for readability: we want our code to be easily read and understood by every member of the development team. Another goal for the use of a code checker in our design is to be able to catch syntax errors in order to avoid bugs. Python uses indentation to denote blocks, and significant logic errors can arise when whitespace is accidentally misused. We want to avoid such errors and reduce time spent on bugs that have a quick fix. A final goal for technology used in our coding environment is ensuring maintainability. The Aerolyzer software should live beyond our senior capstone class; this means other programmers will potentially need to read and add to our code. The handover and upholding process will be much smoother for our client (and result in better-quality software) if all code follows a specific format that can be learned by future team members.

II.3 Criteria

The merits of using technology for our coding environment will be evaluated based on several factors. These include cost, availability, and thoroughness. In order to keep our costs for this project at zero, we need to take advantage of technology that is free - preferably something that is open source. Along the same lines, we want to use technology that will be available and maintained long-term. If our coding environment software becomes deprecated or defunct, our own Aerolyzer software will have to spend time searching for and incorporating a different technology (which is something we definitely want to avoid). Finally, we will need to utilize a code checker that is thorough enough for our needs and enables us to produce code that is both maintainable and able to be modified for our team’s style.
Table 1 compares the three technologies based on the described criteria. [2][3][4]

TABLE 1
Python code checker technologies compared

	Cost	Availability	Thoroughness
Pylint	Free	Currently maintained, last updated 7/2016 (v. 3.5)	Uses Python's own style guide and additional commands/plugins can be added, highly configurable
PyChecker	Free	Last updated 8/2008 (v. 0.8.19)	Website shows a list of bugs
Pyflakes	Free	Currently maintained, last updated 12/2016 (v. 1.4)	Fast but limited in what it checks (does not check style)

II.4 Selection

We have presently begun moving forward with a Pylint configuration based on the above criteria, as well as our client's setup and previous experience. Justification for this decision is based on the many benefits of using Pylint discovered through dialogue with our client and research into our specific needs. These include its ability to catch bugs ranging from small to serious, more easily provide our team with a unified coding style, and offer refactoring features. Furthermore, Pylint is freely available, currently maintained with no evidence of becoming obsolete [5], has plenty of easily understood documentation [6], and is easily customizable to allow for the thoroughness we require.

No tool is without flaws: a common complaint with the Pylint software is that noisy output can be both unhelpful and cause slow analysis. [7] However, Pylint is configurable, so our team can spend some time at the beginning of our development to narrow down a coding style for the Aerolyzer project. This discussion will allow us to end up with a Pylint configuration that reduces unnecessary output and increases analysis speed. Additionally, the time we would spend completing a somewhat slow code analysis is definitely not equal to the time our team would otherwise spend manually finding syntax and logic errors. This time tradeoff will be especially relevant for our student team members - we do not have as much experience coding in Python.

III Image and Metadata Storage

III.1 Available Technologies

The Aerolyzer application will need a centralized storing location for images, associated metadata, and output from the color-analyzing core algorithm. Although we could make use of something simple, such as files in a shared directory, a database would be best suited for our needs: being able to maintain a repository of collected images and corresponding aerosol information will allow us to build up data for use in research, compare users' results against real aerosol content, and query the data set for specific information that would be useful for our team and the research community. Three database options we decided to research are Apache Solr, MongoDB, and MySQL. Apache Solr is a standalone search server with a REST-like API [8]; MongoDB is a NoSQL database [9]; and MySQL is a SQL database management system[10]. All of these technologies are open source and support Python and concurrency. [11]

III.2 Goals

The benefits of using a database management system include data retrieval through a query language, secure data storage, concurrent access by multiple users, and the ability to have data backed-up or recovered. We require a maintained storage of uploaded landscape images and their associated metadata, meteorological data, and geo-related

data. Access to this database is necessary for a range of users, including members of the Aerolyzer development team (for improving the core algorithm) and any researchers who would like to make use of our collected data. This use case outlines two goals for our project: being able to query our data set while allowing for multiple users to add to or pull from our database concurrently. Additionally, using a DBMS will keep our data stored more securely than it would be on a file system - this aim is significant because we want to be able to promise our users that their images will be safely contained. Along the lines of security, a final goal involves being able to back up our dataset. Allowing for recovery in the event of data loss or some other issue will ensure that our collected information can be retained for the life of the application.

III.3 Criteria

The criteria that will be used to evaluate and compare the three DBMS technologies are cost, availability, and security. We want to keep our costs for Aerolyzer at zero, so making use of free software is extremely important. We also need technology that is available long-term and maintained; this is especially significant for our database technology, as we could experience a complete loss of our data if the system becomes defunct. Finally, technologies will be judged on whether data will be stored securely; as mentioned previously, our team wants to promise users that their personal images will be safe and only accessed by authorized users.

Table 2 compares the three technologies based on the described criteria. [11] [12]

TABLE 2
Image and metadata storage technologies compared

	Cost	Availability	Security	Scalability
Solr	Free	Currently maintained (initial release 2004), support for making data persistent	3 vulnerabilities reported in 2016	Highly scalable, supports searches
MySQL	Free	Currently maintained (initial release 1995), support for data persistence	Susceptible to SQL injection-type attacks, 0 vulnerabilities reported in 2016	Scalability requires significant, custom engineering work
MongoDB	Free	Currently maintained (initial release 2004), support for data persistence	JSON documents can be altered maliciously, 1 vulnerability reported in 2016	Highly scalable, not designed for searching

III.4 Selection

Based on research revealed in the above table, we will be using the Solr system. Though the three technologies are very different, they each have similar features based on specific criteria needed for Aerolyzer's data storage. The deciding factor for our purposes is scalability. Comparisons using this criteria reveal that Solr is not only highly scalable, but is the best option for performing searches and even implements realtime search. [8] Scalability will be very important for our app moving forward - we need to be able to store several pieces of data for each processed image and require searching for research purposes. Combining the scalability advantage with the fact that Solr is completely open source, stands behind the Apache brand, and supports data persistence makes it the ideal choice.

IV EXIF Metadata Extraction

IV.1 Available Technologies

In order to ascertain aerosol content, the core color-analyzing algorithm needs an image's metadata as input. We are interested specifically in metadata that will provide the algorithm with needed information, such as the type of smartphone the image was taken on and the location of the photo. Having a wide range of information about uploaded images for the algorithm's input will in turn give output that is as accurate as possible. The three technologies that will be compared for this functionality are the Cloudinary Admin API, ExifRead package, and Pillow ExifTags module. All three APIs can be used in Python. [13] [14] [15]

IV.2 Goals

Our main goal regarding EXIF metadata extraction is to pull as much data from the image as possible. Ideally, we would use a technology that extracts specific data that we need for the core algorithm's input; however, at this stage in the project, we do not have concrete metadata specifications. For example, after the client completes research for and begins designing the core algorithm, it may be discovered that the use of flash when taking a picture is significant; in that case, we would need to pull flash metadata from the image. As a result, we require a technology that extracts all EXIF metadata to aid in our core algorithm implementation.

IV.3 Criteria

Criteria used to analyze the merits of these three technologies will be cost, availability, and what EXIF is returned. As stated throughout this document, we are attempting to create Aerolyzer at no cost; therefore, the selected API needs to be freely available. Along with free availability, a well-maintained API that will be available long-term is a significant factor in our technology selection. The chosen API should be available for as many calls as our system needs. Next, in alignment with our goals, we require technology that returns all associated EXIF metadata; furthermore, any additional photo metadata that the API returns could potentially improve our algorithm's accuracy. The availability of documentation outlining the API's capabilities will provide this information.

Table 3 compares the three technologies based on the described criteria. [13] [14] [15]

TABLE 3
EXIF metadata extraction technologies compared

	Cost	Availability	Metadata returned
Cloudinary	Free for up to 500 requests per hour	Currently maintained, limited to uploaded photos	EXIF, facial recognition, dominant color values and 32 leading colors
ExifRead	Free Python package	Currently maintained, recently updated for Python 3.4 with backwards compatibility (v. 2.6, 2.7, 3.2, 3.3)	EXIF, GPS, and manufacturer data
Pillow	Free Python module	Currently maintained	Well-known EXIF data

IV.4 Selection

Comparing the technologies in the above table reveals various strengths for each API. The Cloudinary Admin API provides not only EXIF data, but facial recognition coordinates, dominant color values, and the 32 leading colors in the

image. [13] This information would be extremely useful for our color-analyzing core algorithm. On the other hand, the ExifRead package supports various versions of Python and returns GPS and manufacturer data in addition to EXIF metadata. [14] GPS information would be provide the core algorithm with additional meteorological data based on the image's location, and manufacturer data would be helpful for narrowing down the camera specs. Some of the tradeoffs of these APIs, however, include Cloudinary's requirement that images be uploaded to their cloud server in order for the API to return metadata. This collides with our application's use of its own database for storing images, and we do not want to store images twice. Additionally, the Pillow ExifTags module's documentation states that it return "well-known EXIF data" and is not clear on whether this includes all data or a specific subset. [15] There are no other outstanding benefits to using the Pillow implementation. For these reasons, we will select the ExifRead package to implement our EXIF metadata extraction. It will be easy to use within our code, as it is a Python package, and we will not have to worry about storing our images in multiple locations. Furthermore, we will be able to have GPS and manufacturer data for our algorithm as well as EXIF metadata.

V Web interface

V.1 Available Technologies

Since the Aerolyzer application is going to be a web application, having a solid web framework is key to a successful application. Our team is using Python to implement the Aerolyzer web application which makes Django the first option. Django is a high-level Python Web framework that "encourages rapid development and clean, pragmatic design" [16]. Our second option is Flask, flask is a microframework for Python that is based off of Werkzeug, and Jinja 2[17]. Our third option is web2py, web2py is a open source full-stack web framework that allows the fast, scalable, secure, and portable database-driven web-based applications[18].

V.2 Goals

The purpose of a framework is to allow the application to be well structured, and allows for other developers to easily maintain or upgrade the application. The goal should allow for the developers to not have to focus on low-level details. The framework should also support numerous activities such as getting form parameters, handling cookies, producing responses, and storing data persistently[19]. We also want the framework to be compatible with Python since that is what our application will be using.

V.3 Criteria

The criteria that will be used to evaluate and compare the three frameworks are cost, availability, and available tools. Since our project is open-source, we want to make sure that there will be no costs to users or developers that want to use our application. We want to make sure that everyone has access to our project, and that the framework has consistent updates and is maintained in the long-term so our project does not get stuck on a dead framework. Finally we want the framework we use to have the tools we need to develop or web application.

Table 4 compares the three technologies based on the described criteria. [19]

TABLE 4
Web interface technologies compared

	Cost	Availability	Tools
Django	Free	Latest update date is 2016-09-01	FullStack, has security features, scalable to large projects, caching framework, design your own templates.
Flask	Free	Latest update date is 2016-05-29	Microframework, built-in development server and debugger, integrated unit testing support, supports cookies, RESTful request dispatching
web2py	Free	Latest update date is 2016-05-10	Fullstack, MVC model, works with MYSQL, capable of uploading/downloading large files, emphasis on backward compatibility

V.4 Selection

After reviewing our goals, the pros and cons of each framework, and the table above, we will be using Django. A fullstack framework would be more useful for our application so that rules out Flask. Django and web2py were very similar, but Django has more little updates so it is being maintained, but less big updates so the project would not have huge updates occurring too often. Django also support large companies that include pinterest and instagram that have a lot of image loading, so it will be able to support our needs of uploading and displaying images.

VI Getting meteorological data

VI.1 Wunderground

A large aspect of Aerolyzer is analyzing the Aerosol content in the pictures, and a crucial part of that is getting meteorological data. When users submit their pictures, they will also be submitting the location and time that will be extracted from the photo information. In order to get accurate aerosol information, the weather information is needed. The Weather Underground (Wunderground) API is the best tool to do this.

VI.2 Goals

Our goal is to gather meteorological data efficiently, and accurately. We want our processing time once the user uploads a picture to be short, but we also want it to be accurate. These two goals are crucial for the success of our application because in this day and age, users have high expectations for applications to run fast, and without accurate meteorological data the results returned to the user will be false.

VI.3 Criteria

The criteria of our technology forgetting meteorological data will be evaluated based on 3 factors. These include cost, availability, and speed. Since our project is open-source the tool we use must be free. It must also be available and maintained in the long-term, we do not want to use a tool that will not be available in a few years. The tool must also be fast. The speed of the application depends on the speed that the tool can send us the meteorological data, and if that is slow, then our entire application will be slow.

VI.4 Selection

In discussions prior to this Technology Review, our team and clients had already made the decision to make use of Weather Underground. Wunderground is the only API that is free, fast and efficient. While there are some other similar API's, Wunderground can process the most calls for free. However it still does have the limiting factor that it can only process 500 photos a day, and 10 calls per minutes [20]. Our client uses this API in her daily work and highly recommended it, so to our team that is a great justification to use this tool. We also did our own research and concluded that Wunderground API is the best tool for our application.

VII TESTING

VII.1 Travis CI and Coveralls

An important part of developing an application is testing. Making sure that all of your code is being covered by tests is really important to avoid bugs in the future. Since we are using Github, the best tools to do this are Travis CI and Coveralls. Travis CI is an open-source integration system that is used to build and test projects in GitHub [21]. Coveralls is an open-source tool that shows what parts of the code isn't covered by the test suite [22].

VII.2 Goals

With goals for our testing, we want a tool that will run all of our tests. This is important because as we add code to our code base, we want to make sure that our tests are also changing to cover the code. To help with that we want to see what tests cover what part of the code, and what code is uncovered. That way we know what part of the test suite to alter to get to as close as 100 percent of the covered code as we can. This helps reduce the amount of bugs we will have to later fix. It will also help reduce the amount of unexpected errors we have later in the project when our code base is big.

VII.3 Criteria

The criteria for the tools used for testing are cost, availability, and speed. Our project is open source so we want to make sure that the tools we use are either open-source or free. We also want to make sure that these tools will be available for long-term use. It is important to look and see how often the tools are getting updates, and how long they have been around for. One of the most important criterias is speed. When the code base gets large, the tool has to be able to keep up with the amount of tests and code. The tests have to be run fast and efficient, or else it could take a really long time to test things once the projects gets large.

VII.4 Selection

Since we are using Github, TravisCI and Coveralls are the best option for testing. They are easily integrated into Github which makes them very easy to use to our project. They both have fairly new versions, and are being well-maintained. Another great thing about these two tools is that they are both completely open-source which works great for our project. Our client recommended these tools to us prior to this tech document, and after much research we agree that these are the best tools to integrate into our project on Github.

VIII Uploading Photos

VIII.1 Available Technologies

In order for our application to properly analyze images provided by a user, we will need to make use of an API that allows users to upload images to our web application. This is a crucial part of our application because without it, we would have no way of retrieving images and would therefore be unable to properly analyze a user's image. To fulfill this requirement, we examined three different image upload APIs that provide a multitude of different advantages. These APIs include DropzoneJS, Ospry, and Cloudinary. DropzoneJS is an open source library which provides its users with drag and drop image uploads as well as image previews[23]. Ospry on the other hand is a very simple way for developers to upload images to the cloud while providing additional tools which makes manipulation of the image very easy[24]. Lastly, Cloudinary provides image management in the cloud, and provides end-to-end solutions for web and mobile developers[25].

VIII.2 Goals

The upload image functionality of our application should be simple and easy to use. However, it should simultaneously allow users to upload high quality images in order for us to adequately analyze the content of the image being uploaded. We need an API for image uploading that is compatible with our code, and allows users to upload images from both the web and via a mobile device.

Table 5 compares the three technologies based on the described criteria. [23] [25] [24] [26]

TABLE 5
Upload photos technologies compared

	Cost	Availability	Tools/Advantages
DropzoneJS	Free	Currently maintained	Drag and drop, open source, highly customizable, simple usage, programmatic dropzones, compatible
Ospry	Not Free	Currently maintained	Reliable cloud storage, image manipulation
Cloudinary	Not Free	Currently maintained	Reliable cloud storage, security, image manipulation

VIII.3 Criteria

When searching for an API which we will use for image uploading, we must keep in mind certain criteria. First of all, we want to ensure that uploading the image is a simple process for the user, and can be performed from both the web and via a mobile device. Moreover, we need to make sure that the API being used allows for the upload of large images, and adequately maintains all of the image's data. This is a crucial aspect of this technology because we need to be able to accurately analyze the data provided by the image if we wish to determine the aerosol content from its features.

VIII.4 Selection

After extensive research on the aforementioned available technologies, our Aerolyzer application will make use of the DropzoneJS API for various reasons. First and foremost, DropzoneJS does not depend on any outside libraries and is highly customizable, which means we can manipulate it to fit our specific needs. Moreover, while DropzoneJS does not

provide the server side implementation of handling the images, its does provide a simplistic file upload and allows for file dropzones to be created programmatically. This API is also compatible with most main browsers including Chrome, IE, Firefox, and Safari[23]. While Ospry and Cloudinary also provided desirable features such as reliable cloud storage and security, we chose to go with DropzoneJS because we prefered the drag and drop functionality and favored the fact that it was an open source technology.

IX Getting Geo-related Data

IX.1 MISR

As part of our application, our team requires the ability to obtain geo-related data in order to accurately analyze the aerosol content of the air. In order to do that, we will be using the Multi-Angle Imaging SpectroRadiometer (MISR). MISR is an API which views the sunlit Earth using nine widely spaced angles, and provides ongoing global coverage with high spatial detail. It is calibrated to provide accurate measurements of the brightness, contrast, and color of reflected sunlight[27]. The data and information provided by this API will allow us to adequately interpret the aerosol content of images sent through our algorithm by our users.

IX.2 Goals

The goal of using a geo-related tool is to obtain accurate information of the aerosol in the air, the cloud formation, and the light reflection. Through the use of such information, we will be able to accurately analyze the aerosol content in the picture using the data we obtain from our geo-related tool. Therefore, our goal is to use an API which provides accurate information and data in large enough amounts that our aerosol content algorithm will be capable of determining the aerosol content of any given accepted picture.

IX.3 Criteria

The criteria of our geo-related tool includes cost, availability, and accuracy. As an open source project, we aim to minimize costs while providing accurate results. As for availability, our application requires that the API supplying geo-related information and data be consistently running, otherwise our ability to determine aerosol content would be drastically reduced. Looking at our accuracy requirement, we need an API that will supply geo-related information not only on time, but accurately. If we receive information that is not accurate, our algorithm will have poor performance results and be incapable of adequately determining the aerosol content of user uploaded images.

IX.4 Selection

After discussing available technologies with our client, we decided that we will be using MISR for gathering geo-related data. It is the only satellite available for public use of the information, and more importantly it also provides highly accurate data. Our client works at NASA JPL, the company which currently commissions the satellite. Therefore, we will have someone on our team with prior knowledge on how to properly analyze and use all of the data. Moreover, this API satisfies all of our criteria mentioned above. The results from MISR are public information, so there is no cost associated with using MISR data. MISR also has no plans of being decommissioned, so the data will consistently be available assuming MISR is not decommissioned. Finally, MISR has 9 angles from which it analyzes Earth's atmosphere, so the results and data provided by MISR are highly accurate. Ultimately, it is clear that MISR is the ideal API for our geo-related data needs.

X Documentation/Development of API Documents

X.1 Sphinx

When it comes to the documentation and development of our API documents, our team requires the use of an API to create professional, informative, and concise documents. Sphinx is a documentation tool which allows its users to create both intelligent and aesthetically pleasing documents. This API's output formats include HTML, LaTeX, and several others. Moreover, it allows its users to perform semantic markups, as well as use automatic links for citations, glossary terms, etc. It also provides a hierarchical structure as well as code handling tools. Overall this API provides a multitude of functions that serve us well when completing our documentation phase.

X.2 Goals

The goal of using an API for documentation and the development of API documents is to produce documents that will be both professional and informative. As we develop our Aerolyzer application and once it has been completed, we will need to develop both coding and usage documentation for our users. Without documentation on our code and how to use our application, Aerolyzer will be minimally effective as users will be unable to adequately upload images and receive information on their aersol content. Therefore, it is imperative that we use a documentation API which provides the tools and functions necessary to develop clear, concise, and professional documents.

X.3 Criteria

As for the criteria of our application's documentation and development of API documents, there are several factors that we must take into consideration when choosing which API to use. Such features include price, availability, and tools provided. As an open source project, we need to ensure that the API being used for documentation is affordable. Moreover, we need to ensure that the API is consistently available throughout our documentation phases and produces outputs that are compatible for all users. Finally, we need to make sure that the specific tools and functions of the documentation API we choose provide us with the abilities necessary to create professional and informative documents that will satisfy our documentation requirements.

X.4 Selection

After conferring with our client, we determined that Sphinx would suffice as our documentation API for several reasons. First of all, the quality of the documents produced by Sphinx will exceed our expectations for code and usage documentation. Moreover, the tools provided for semantic markup, automatic links, and other various functions will aid in our documentation process greatly. Sphinx's ability to output documents in HTML, LaTeX, and various other formats meets our documentation compatibility needs, providing most users with access to our documentation. Overall, Sphinx will be a great addition to our set of technologies used for the Aerolyzer development process.

XI Conclusion

In conclusion, several technologies will be used to fully implement the Aerolyzer application. Our chosen technologies for application-side functionality include Django for the web interface, MySQL for storage, ExifRead for EXIF data extraction, Dropzone JS for uploading images, and Weather Underground and MISR for weather-related data. For the development team, we will be making use of Travis CI and Coveralls for testing, PyLint for our enforcing a coding

style, and Sphinx for documentation. These selections were made based on research into different options and using criteria specific to our needs. A few of our technologies for pieces of the project have already been chosen and put into use; as a result, comparisons were not needed for these cases.[28]

[1] Pylint. Introduction. [Online]. Available: https://pylint.readthedocs.io/en/latest/intro.html
[2] P. S. Foundation. Python package index. [Online]. Available: https://pypi.python.org/pypi/pylint
[3] Neal. Pychecker. [Online]. Available: http://pychecker.sourceforge.net/
[4] P. S. Foundation. pyflakes. [Online]. Available: https://pypi.python.org/pypi/pyflakes
[5] J. Castro. (2016) Review of python static analysis tools. [Online]. Available: http://blog.codacy.com/2016/01/08/review-of-python-static-analysis-tools/
[6] Pylint. Tutorial. [Online]. Available: https://pylint.readthedocs.io/en/latest/tutorial.html
[7] PyDev. Pylint. [Online]. Available: http://www.pydev.org/manual_adv_pylint.html
[8] A. S. Foundation. (2016) Apache solr - features. [Online]. Available: http://lucene.apache.org/solr/features.html
[9] I. MongoDB. (2016) Mongodb for giant ideas. [Online]. Available: https://www.mongodb.com/
[10] Oracle. Mysql. [Online]. Available: https://www.oracle.com/mysql/index.html
[11] DB-Engines. Mongodb vs. mysql vs. solr comparison. [Online]. Available: http://db-engines.com/en/system/MySQL%3BSolr%3Bmongodb
[12] M. Corporation. (2016) Cve security vulnerability database. [Online]. Available: http://www.cvedetails.com/
[13] Cloudinary. Admin api. [Online]. Available: http://cloudinary.com/documentation/admin_api
[14] P. S. F. Python Package Index. (2015) Exifread 2.1.2. [Online]. Available: https://pypi.python.org/pypi/ExifRead
[15] P. P. Fork). Exiftags module. [Online]. Available: http://pillow.readthedocs.io/en/3.4.x/reference/ExifTags.html
[16] threespot. (2016) Django. [Online]. Available: https://www.djangoproject.com/
[17] A. Ronacher. (2016) Flask. [Online]. Available: flask.pocoo.org/
[18] M. D. Pierro. (2016) Web2py. [Online]. Available: http://www.web2py.com/
[19] J. Koskelainen. (2016) Web frameworks for python. [Online]. Available: https://wiki.python.org/moin/WebFrameworks
[20] T. W. Company. (2016) Weather underground. [Online]. Available: https://www.wunderground.com/weather/api/
[21] T. CI. (2016) Travis ci. [Online]. Available: https://travis-ci.com/
[22] L. H. Industries. (2016) Coveralls. [Online]. Available: https://coveralls.io/
[23] M. Meno. (2016) Dropzonejs. [Online]. Available: http://www.dropzonejs.com/#
[24] Ospry. (2016) Ospry. [Online]. Available: https://www.ospry.io/docs
[25] Cloudinary. (2016) Cloudinary. [Online]. Available: http://cloudinary.com/
[26] D. Hough. Dropzone and cloudinary. [Online]. Available: https://danhough.com/blog/dropzone-cloudinary/
[27] N. JPL. (2016) Misr. [Online]. Available: misr.jpl.nasa.gov/Mission/
[28] Sphinx. (2016) Sphinx. [Online]. Available: http://www.sphinx-doc.org/en/1.4.8/index.html