TWEETS METADATA

May 7, 2015

Virginia Tech Department of Computer Science
CS 4624: Multimedia, Hypertext, and Information Access
Taught by Edward A. Fox
Client: Mohamed Magdy

Team Members:
Christopher Conley
Alex Druckenbrod
Karl Meyer
Sam Muggleworth

Table of Contents

Table of Figures

Executive Summary

The previous CTRnet and current IDEAL projects have involved collecting large numbers of tweets about different events. Others also collect about events, so it is important to be able to merge tweet collections. We need a metadata standard to describe tweet collections in order to perform this merge and to be able to store them logically in a database. We were tasked with creating software to carry out the merger, and to prepare summaries and reports about the respective collections, their union, and their overlap, etc. Preliminary work on this was carried out by Michael Shuffett (Ref. iv) and we were asked to develop upon his code test it, extend it where necessary, apply new technology, and further document it before publishing.

We met with our client, Mohamed Magdy, a PhD candidate working with QCRI and Dr. Fox and came up with the following project deliverables: a standard for tweet sharing and for tweet collection metadata, a method for merging such collections and developing a report, putting all of these things together into a web-application tool.

The expected impact of this project will be having increased collaboration between researchers and investigators all trying to use tweet metadata to find insights into everything from natural disasters to criminal activity and even stock market trends. A tool of this type will help ease the process of merging archives of tweets between researchers which will then lead to better analysis and less time spent trying to re-organize information that could be sifted through by this tool.

Our team was able to develop upon Michael Shuffett's code, improve it, and set up new and improved wireframes for the website. We were able to start framing out a tool that allows more than two types of files to be merged, which previously had to be in a single format. In the future, the required formats wouldn't be as strict, making it a lot easier to upload different types of files thus making it even easier on the user.

User's Manual

The following is a guide to using the TweetsMetadata web tool. This tool is meant to encourage the sharing and archiving of event-based tweet collections for research and data analytics purposes. The tool is not fully integrated and tested at this point, so this serves as an overview of how actions with the tool are meant to work.

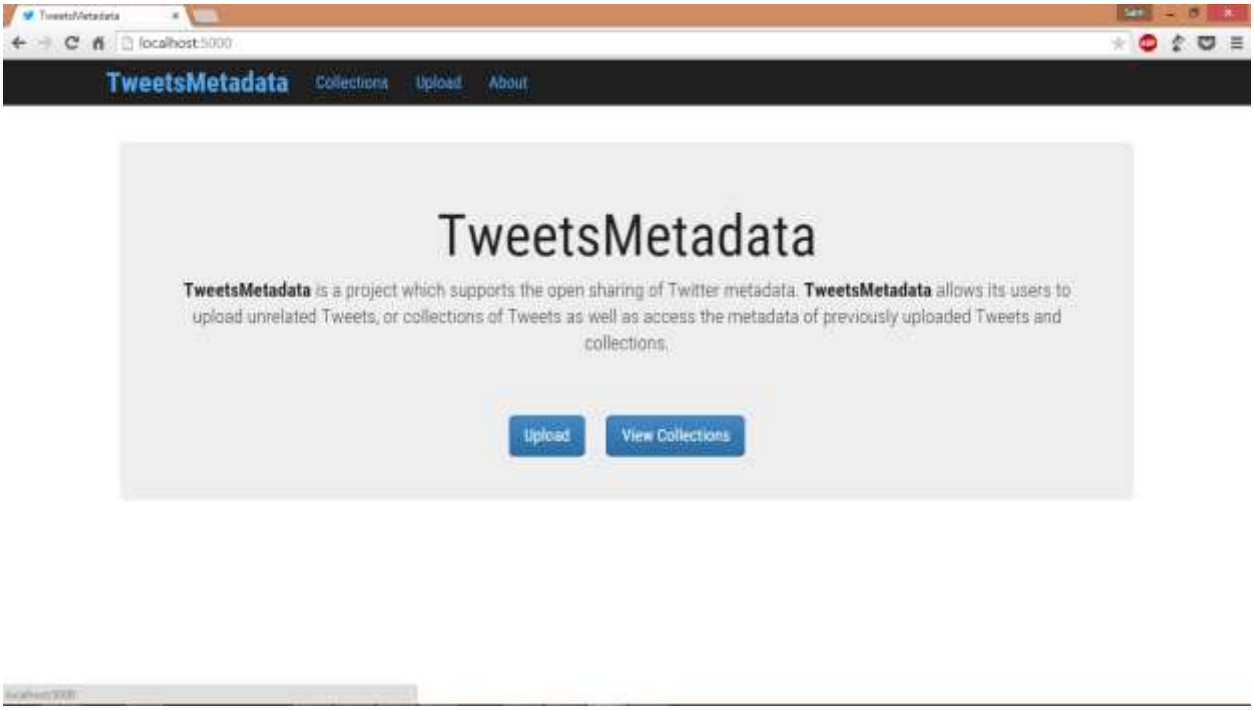

Figure 1: Home Page

Uploading a collection: To Upload a collection:

1) Click the upload button at the top of the page, in the navbar (Figure 1).
2) Fill in the required information and any additional information (Figure 2).
3) Select a file then click next.
4) If the file selected is properly formatted, it will be uploaded, otherwise an error will be thrown.
5) If successful, you will see an indicator under the navbar at the top of the page and they will be taken to our skeleton interface for mapping different collection schemas to ours (Figure 3).

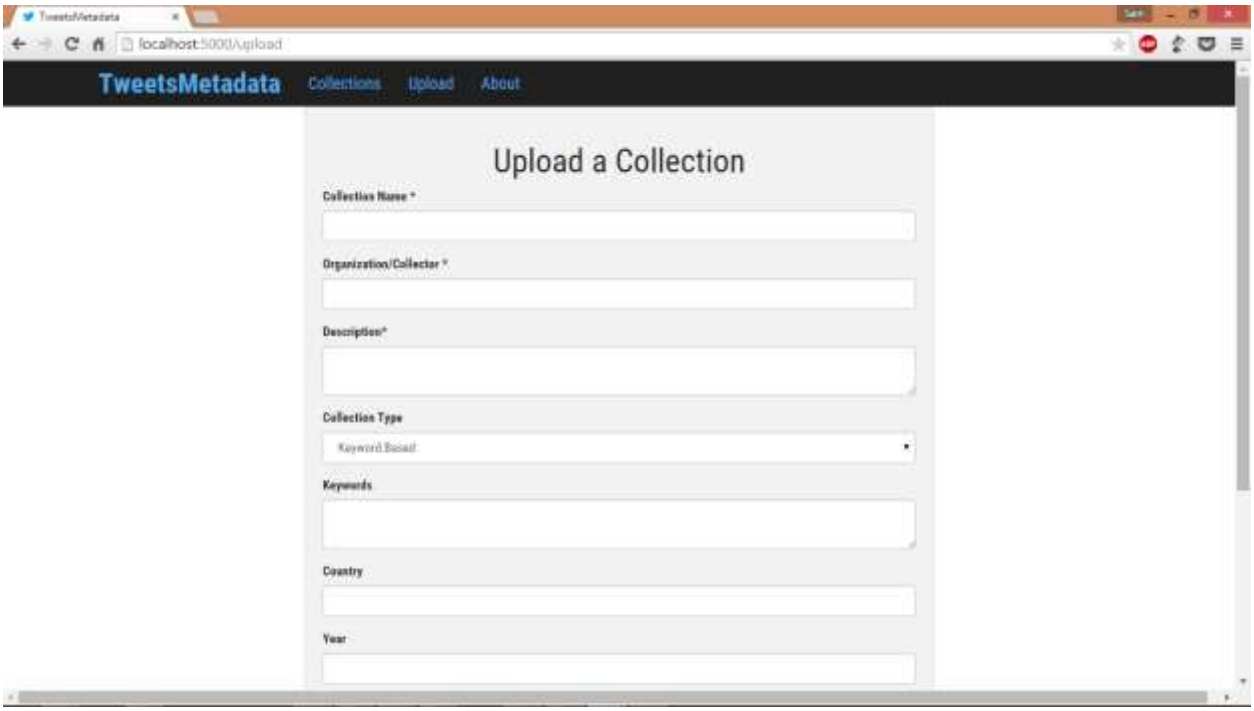

Figure 2: Upload Form

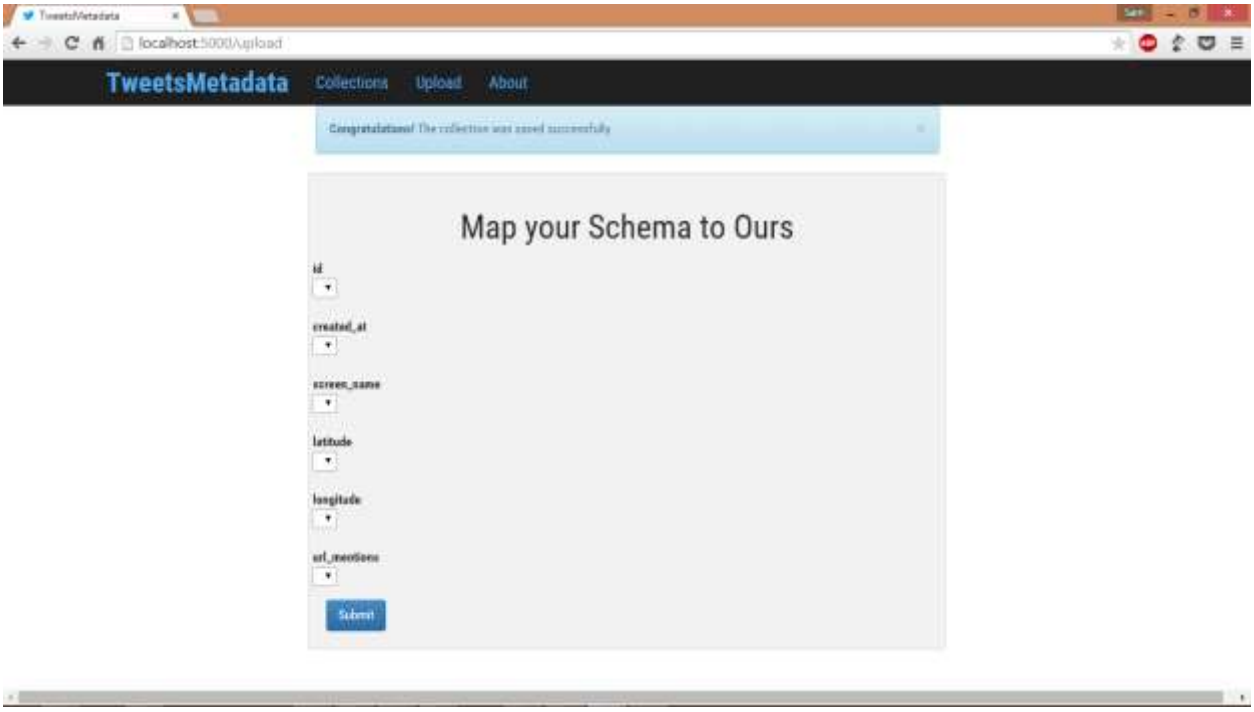

Figure 3: Successful Upload and Schema mapping skeleton

Viewing list of Collections: To view the list of Collections,

1) Click the Collections link at the top of the page
2) The page for the list of collections would come up as in Figure 4.
    a. Collections are currently sorted by year

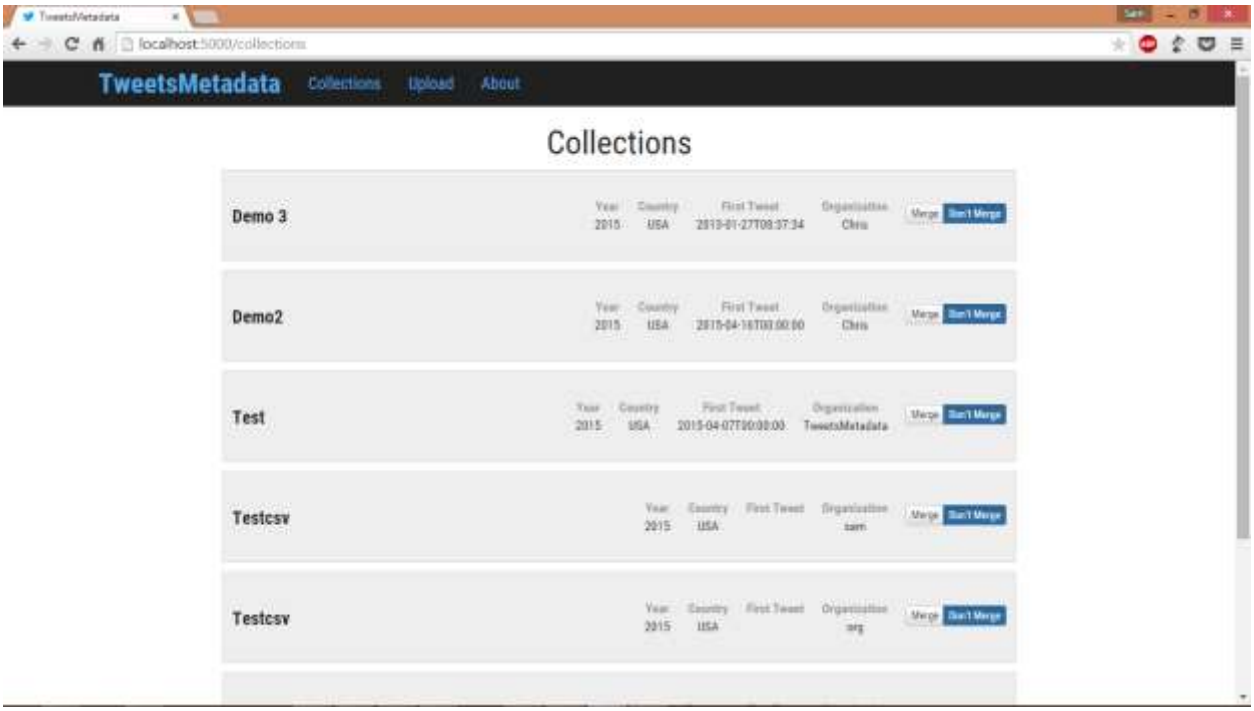

Figure 4: The Collections page

Preparing/Executing a Merge: To request a merge,

1) User selects a collection by toggling the merge button on the right of a block (Figure 4).
    a. The user can select up to two collections to merge at a time.
2) At the bottom of the page is a button that says Merge
    a. If is active, the merge can occur
    b. If not, too few or too many collections are selected
3) The request is sent to the server which processes the query and returns the resulting merge.
4) When received, the merge summary and report page is brought up, displaying the results of the merge.

Learn More About the Project:

Go to the about page. Click the About link in the navbar and the page will come up (Figure 5).

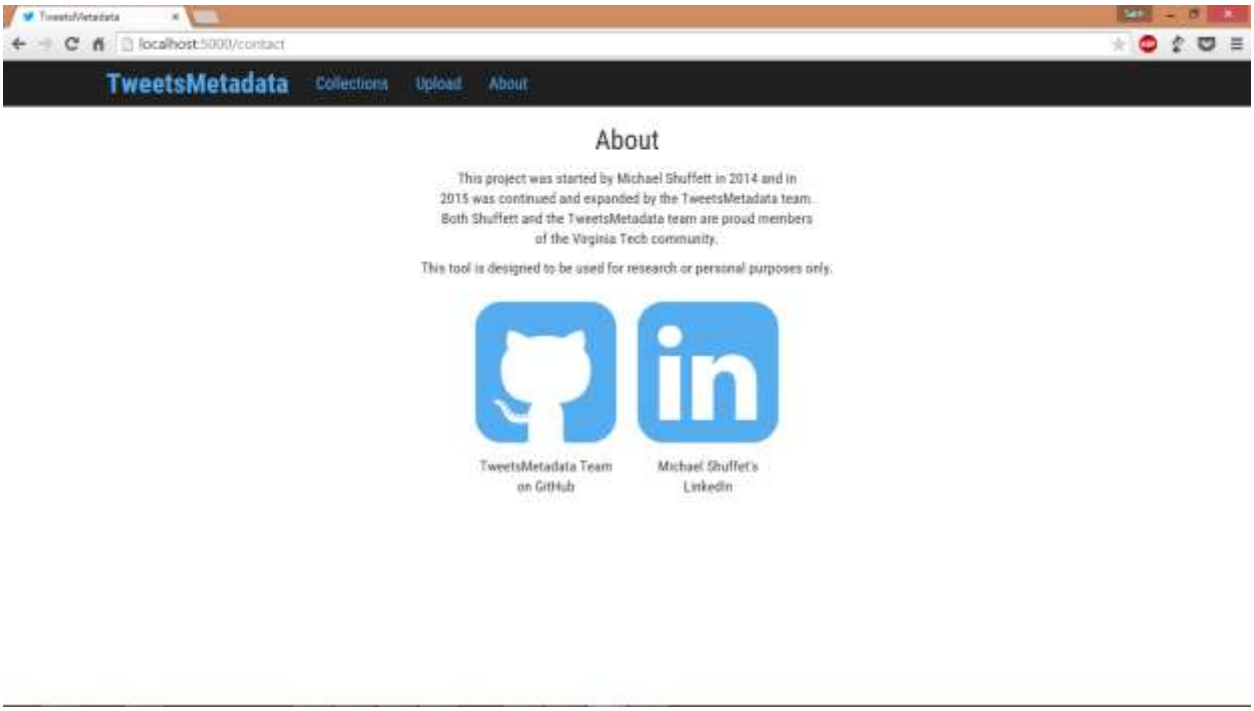

Figure 5: About/Contact page.

Developer's Manual

Here is how we expanded the tool and our thoughts going in, and coming out of the project. Please find below everything necessary to develop upon our work or to understand its more intimate details.

Project Status

Due to reasons briefly touched on in the Lessons Learned section (pg. 13), the project does not now work as intended. It still only accepts specifically formatted csv or tsv files. After a discussion with Dr. Edward Fox and Mohamed Magdy, we resolved to put together and deliver the parts of the missing step, schema mapping. We hope that those continuing development on this tool will find our comments here beneficial, whether starting the project from scratch or continuing where we left off.

Technologies and Software

The software and technologies used for this project were largely decided by Michael Shuffett as we were expanding his project from May, 2014 (Ref. iv).

On the front end, Shuffett had been using jinja 2 (templates), WTForm/Flask-WTForm (forms/uploading info), Bootstrap/Flask-Bootstrap (style/CSS), and jQuery/Flask-Script (client-side scripting). These tools make it easy to create form and list based tools such as this one. **Figure 12** shows an example of how easy it is to build a dynamic section of a page.

```
{% for collection/tweet in collections/tweets %}
      <div>
            <ul>
                  <li>collection/tweet.name</li>
                  <li>collection/tweet.created</li>
                  ...
            </ul>
      </div>
{% endfor %}
```

Figure 6: Jinja2 Code example

As demonstrated above, jinja2 makes it easy to recycle templates and create dynamic pages that can render large sets of data with minimal programmer effort. With the other technologies, Flask specific versions of each have been included allowing for guaranteed cooperation and compatibility with the server side as it is using Flask for its server software. The backend of the web application is written in Python. We are specifically using the Flask library for easy server integration. All code has been uploaded to a team GitHub page.

Instructions for installing all software for developing, running, and managing the tool can be found in Michael Shuffett's report (Ref. iv).

Functionality break down

There are mainly three ways where the front and back ends interact:

1. When a user requests a page (i.e. collections list page),
2. When a user uploads a collection to the tool, or
3. When a user requests a merge of collections.

In the first case, on page load a RESTful call will be made to populate the page with whatever it needs (list of collections or contributors, associated information, etc.). This call (GET) is a simple query for a list or additional information returned in json format. It is then broken down and displayed using jinja2 templates.

In the second case, the user has clicked the upload button, fills out the form ([Flask-]WTForm), and attaches the file to upload. Then the form is compiled into a packet and, if it is properly filled out (validated), POSTed to the server (again, RESTful).

In the last case, the user has decided to merge two collections. The collections chosen, the user clicks merge and a query is sent to the server to merge (POST). At the server an algorithm comparing all collection fields is used to create a temporary list of similarities. At this point, merge records are not persisted. When completed, the merged collection is sent back to the user along with a summary of the report. The merge success page shows the summary of the resulting collection (eg. what was merged, duplicate tweets).

Database Components

The database is organized into three parts. The first is the collection of tweets. The tweet schema is made up of the tweet id, date created, screen name of the user, latitude, longitude, and url_mentions (links in the tweet). The tweet id is a tweets principal key.

The second part is the collection data. Each collection stored has an id, name, organization (who uploaded it), description, type, keywords, country, tags, first_tweet_date, and last_tweet_date. The latter two are computed in extract_dates.py. In Shuffett's implementation, the collection name was the principal key. We have changed this to the collection id, which means for larger databases, duplicate collection names can exist and be completely separate.

The last part is a set of associations for collection id to tweet ids. In this manner, duplicate tweets won't exist in the database, but every collection still has its correct set of associated tweets.

Schema Mapping Model

Schema mapping is not fully implemented at this point. The parts necessary for it are in place but they are not linked together yet. The following is a look at those pieces and how they could fit together in future development.

```
# dargs is the tuple that contains which user column matches up to
# are schema fields
# this static example just sets them according to Michael's code
# the full product should set the dargs equal to column indeces
# that the user inputs
# make another function for this
dargs = {}
dargs['id'] = 0
dargs['created_at'] = 2
dargs['screen_name'] = 3
dargs['latitude'] = 4
dargs['longitude'] = 5
dargs['url_mentions'] = 6

# kwargs is the data that gets written to the db
# dargs['id'] gets the index in dargs that 'id' is at
kwargs = {}
kwargs['id'] = row[dargs['id']]
kwargs['created_at'] = row[dargs['created_at']]
kwargs['screen_name'] = row[dargs['screen_name']]
kwargs['latitude'] = row[dargs['latitude']]
kwargs['longitude'] = row[dargs['longitude']]
kwargs['url_mentions'] = row[dargs['url_mentions']]
```

This snippet gives a glimpse of how the manual mapping should work. The dargs[FIELD] = INDEX pattern will work perfectly if the user is prompted to map which index their columns align to. We would subtly ask the user for the index of the field. A new page would then be displayed with our schema's fields and drop down boxes populated with the user's fields.

```
# we get the headers from the file here so we can populate
# the the user's options for manual mapping
# form.check.data is the true/false value from the checkbox
# that asks if the user's file has headers
# if there are headers we populate the options with them
# if not we simply ask them which column index matches up with our
# schema fields
headers = {}
for i, row in enumerate(temp_file):
    # we do len(row) - 2 since the last 2 fields are null values
    headers_length = len(row) -2
    if(form.check.data):
        for k in range(headers_length):
            headers[k] = row[k]
    else:
        for k in range(headers_length):
            headers[k] = k
    break

# this is where we would make another call that generates
# the manual mapping form
# this function would return dargs which would then pass into
# the tsv.process_tsv() call
```

This is where we generate the header information. If the user had headers, we read them off and then populate the drop down boxes with them. We know what index each header is at since we

know the order the headers are in. If the user's do not have headers, we just populate the drop down with numbers 1 to number of headers and simply ask the user what index each header is at. Most TSV and CSV files should have headers since that is the only way a person/program can know exactly what each column represents. We left functionality that allowed for no headers since Michael's code initially specified that files do not have headers.

Program Files

The following is a Tree of directories and files for the project generated from the Windows command prompt using the command: tree /A /F

```
\---TweetMetaData
|   collection_id
|   extract_dates.py
|   manage.py
|   requirements.txt
|   tweetid.db
|
+---static
|       bootstrap-toggle.js
|       favicon.ico
|       loading.GIF
|       merge.js
|       purl.js
|       styles.css
|
+---templates
|       about.html
|       base.html
|       collections.html
|       collection_details.html
|       completed_merge.html
|       contact.html
|       index.html
|       mapping.html
|       merge.html
|       pagination.html
|       sample.html
|       single_tweet.html
|       upload.html
|
+---TestData
|       data_1.csv
|       data_1.tsv
|       data_text.tsv
|       ebola.csv
|
+---tweetid
|       app.py
|       config.py
|       load_json.py
|       models.py
|       tsv.py
|       utils.py
|       __init__.py
```

Figure 7: Project Folder and Files Tree

In the root folder, extract dates is used for extracting dates from tweet metadata to calculate the first and last tweet found in a collection. The manage.py code is how the tool is initialized, cleared, and run. Type run 'python manage.py help' to see the options. Collection_id and tweetid.db are a simple text file containing the next id to be assigned to an uploaded collection and the database for the tool, respectively. Requirements.txt contains the tool names and versions required to run the tool. These can be installed using the pip command (instructions and more information in Michael Shuffett's report, Ref. iv).

The static folder contains javascript, css, and icon files. Bootstrap-toggle.js is where the activation, deactivation, and job of the merge button (on the collections page) is managed. Facivon.ico and loading.GIF are just icons in the tool. Styles.css is any custom styling on top of the bootstrap library. Merge and purl contain additional helper code.

The templates folder contains HTML templates, using jinja2, for the various pages. Single_tweet and sample are not currently used, but do contain good examples of jinja templating.

The TestData folder contains the short files we used to test uploading various schemas when exploring how strict Shuffetts design was and what needed to be done to improve it.

Last, the tweetid folder contains python scripts used in actually running the application, from routing to adding collections to the database.

Lessons Learned

Many lessons were learned throughout the course of this project. Some of them lie on the side of project management and some on the side of development.

On the project management side, we learned the importance of fully understanding and communicating understanding of the requirements for a project. When we first started there was miscommunication on the direction of the project and thus we had shaky roots. From having separate conversations with Dr. Edward Fox and Mohamed Magdy we became confused in the direction the project should be going and the breadth of the project we signed on to. Thus, we learned from experience that it is important to have a clear grasp of the big picture of a project before attempting to understand the working pieces of it.

Another lesson learned, through a conversation later in the semester, was that sometimes it is better to start a project from scratch than to start from a pre-existing code base. In our case, we did not start from scratch because we had come under the impression that we were expected to use the setup that Michael Shuffett had already started. This led to slower and later incomplete development as we were not familiar with the tools, particularly with using python, flask, and jinja as web development tools. Had we gone with something more familiar, such as an HTML/CSS/JS only implementation using jQuery or AngularJS, then this project may have been in a very different spot now.

Along that same train of thought, we learned that if you are using a code base written by someone else, it is important to first understand the technologies that they chose to use and then go in and see how they used them. We jumped head first into a pool of technologies that we didn't have a good grasp on and lost ourselves in Shuffett's project, unable to get a firm grasp on what it was actually doing for a long time.

Another part of why we had a difficult time breaking down Shuffett's code was because of lack of complete commenting throughout. There were some confusing sections that were messy in terms of variable names and methods used that had no comments, making it hard to be sure we understood what was happening. Therefore, another lesson we learned was the important of documenting code and giving more indications of what a process is doing, not just over-arcing functions. Projects are passed on and improved regularly, thus it is important that whoever goes in to the code next will be able to understand what is happening.

During development, we started implementation of mapping schemas, but it didn't get integrated because we ran into some major development problems, such as when we tried to switch to Python 3 and were unable to get any of the code working for weeks. As it turns out, parts of Python 3 were not fully compatible with the other technologies being used. Lesson: Always make sure that when changing any component of a project that it will still be fully compatible with the rest, whether the core language or just an additional library. Sometimes it will seem largely compatible but then a small part of it won't be.

Timeline/Schedule

Below is our implementation schedule as of the final day of work.

	March	March	March	April	April	April	April

Front-End	collection standard & mapping info	Style modi-fications	Mods cont'd	repackaging tweets	testing		
Back-End	Upload & parse CSV files	Upload & parse JSON files	Upload & parse XML files	testing			
Both	obtaining req'd & addt'l metadata			obtaining req'd & addt'l metadata	merge font with back	generate summary	testing

Future Work

For the future development of this project, we would like to recommend several advancements in the following key areas:

- The software should be able to upload any file with tweets in it. Right now there is only one format that is accepted: it needs to have 6 fields with tab-separated values.
- The software only takes TSV-formatted files, and it should take: JSON, CSV and TSV files in order to upload any type of reasonable collection
- Right now, it does not allow headers, but the new software should allow for this type of differentiation so as to allow for collections that have been manually altered to display their headers as well
- Future development should continue to only require the TweetID and UserID in the collected tweets metadata.[1]
- Possible: to make an autonomous tool, this could avoid the necessity of the user manually mapping archival information to make the merge possible
- Possible: to use an autonomous schema mapper tool like the one Dr. Fox used for archaeological dig information (Ref. iii)

[1]As before, due to changes to the Twitter ToS and Developer Policies (Refs. ii, v)

Acknowledgements

We would not have gotten far without help from the following people:

Dr. Edward Fox

Thank you for all of your time and consideration throughout the course of this Senior Capstone Project. Without your saintly patience and help, we would be in another place…thank you.

Mohamed Magdy

Thanks for meeting us all of those mornings and for listening to our group make ignorant mistakes. But most of all, for correcting our mistakes through advice and suggestions. Could not have don't it without you.

Michael Shuffett

It was a blessing to have some idea of what to work with when starting a project of this scale, and we appreciate the time and effort that you put into the beginning of this project.

NSF grant IIS - 1319578, III: Small: Integrated Digital Event Archiving and Library (IDEAL)

References

i. CS_4624_12303_201501: Wiki->ProjectsS15->TweetsMetadata <https://scholar.vt.edu>

ii. "Developer Policy." *Developer Policy*. Twitter, Inc., 22 Oct. 2014. Web. <https://dev.twitter.com/overview/terms/policy>.

iii. Fan, Weiguo; Fox, Edward A; Shen, Rao; Vemuri, Naga Srinivas; Vijayaraghavan, Vidhya. "EtanaViz: A Visual User Interface to Archaeological Digital Libraries." *EtanaViz: A Visual User Interface to Archaeological Digital Libraries*. Virginia Polytechnic Institute and State University, 1 Oct. 2005. Web. <https://vtechworks.lib.vt.edu/handle/10919/20193>.

iv. Shuffett, Michael. "Twitter Metadata." *Twitter Metadata.* Virginia Polytechnic Institute and State University, 10 May 2014. Web. 09 Feb. 2015. < https://vtechworks.lib.vt.edu/handle/10919/47949>.

v. "Twitter Terms of Service." Twitter, Inc., 8 Sept. 2014. Web. <https://twitter.com/tos?lang=en>.