Final Report

Wes Lanning, Tony Flemings   
COMP 495 V1WW

Professor Moreland  
December 14, 2013

**Project Requirements**

Our project ended up not 100% completing trending hashtags, total complete searching for things other than other users, and replying/retweeting/favoriting posts. Much of the code to do all that was there. However, we did not have enough time to put into projection. Although a project like this can have an infinite number of requirements, at its basic level of functionality, it should have something like the following:

1. Front page for the site describing the site. Users that already have an account and are signed in will see their dashboard instead of this page when they go to the site index page (described below). Those without an account can view/search messages of other users and view those users’ dashboards, but are not able to post any content without signing up.
2. Account creation for users (signup page). In order to post on the website, every user must create an account that will collect their email, user name, optional real name and password.
   1. Usernames should be alphanumeric only (i.e. [A-Za-z0-9]+ as a regular expression) to simplify searching and parsing.
3. After account creation, user is redirected to their dashboard/homepage to begin creating messages or to find users to subscribe to. The dashboard will contain the following:
   1. All messages created by the user ordered by most recent
   2. All messages created by posters that the user subscribes to ordered by most recent Messages are mixed in with the user’s own messages with the ability for a user to see only their own on a separate page. Each message will be like a separate container/card/box in a list with controls (described below)
   3. A search box to find other users/content (described below)
   4. A list of trends (described below)
4. Each message card/box/container should contain the following (with controls adjusted if viewed by the author or another user):
   1. A message of 140 characters or less
   2. The poster’s name/real name that links to the poster’s dashboard
   3. Time the message was posted (in days:minutes:seconds) with a link wrapped around the time to share the message with others when they load the link in their browser
   4. Button for a user to add a poster’s message to their own dashboard
   5. Button for the user to bookmark/favorite a message by a poster (or “unfavorite” it)
   6. Button for the user to subscribe to a poster’s messages (which adds all new messages by that poster to the user’s dashboard) or unsubscribe (which removes all previous messages by that poster from the user’s dashboard and stops adding new ones).
   7. The original creator of a message has the ability to delete any message they create with a button next to the message as well. Deleting the message will also delete it from all other user dashboards and any shared links to the message will show a “404” (not found) error message.
   8. A button and textbox to reply to a user’s post (described below). To optimize space, the textbox should not be shown until a user clicks the “reply” button. Once clicked, a textbox will appear below the message along with a “send” button. The message will then be posted to the intended recipient’s dashboard for them to see. It will also be shown on the creator’s dashboard. Although not required, it would be optimal to make all replies show in a threaded/tree format (that show when clicking a view replies link/button) so that they do not clutter up the dashboards of users.
5. Users can send other users messages by including the username of the person(s) the message is intended for prepended with an @ sign in front of the username (i.e. @happyuser23 or @soccerfan99). Usage example:

@redsfan76, it looks like the #reds might make the playoffs if they can only beat the #pirates.

1. Users will have a page they can display all messages they bookmarked. Format is the same as all other pages with messages and sorted by time the message was created.
2. Ability to find other users to subscribe to. Ideally, this will be done with a search box. If time permitting, the search box will give suggestions as the user types to help facilitate easier searching. The following content can be search for in the search box:
   1. Username
   2. Real Name
   3. Email
   4. Hashtag (subjects)
3. Hashtags will also serve as trends. Trends are a finite number of popular hashtags currently being used. Trends will show to a user in the form of a list of links they can click on. When a particular trend is selected from the list, the user is given a filtered list of recent messages that contain the trend/hashtag ordered by the most recent. A user will then be able to click on any message and be taken to the dashboard of the user that posted it or optionally subscribe to that person without going to the page via the message itself. Users will also be able to add a poster’s message they like to their own dashboard so that those that subscribe to the user will see the message of the original poster. Users will also be able to bookmark/favorite messages to save for later without adding to their dashboard.
4. Users will also have a settings page where they can change their password and email.
5. Log out button and log out page
6. Log in button and log in page that allows a user to log in via either their email or username. No two users ever have the same email or username.

**Project Design Overview**

For convenience, documentation explaining how things work on a technical level can be found in this project under the /docs directory by opening the index page there in one’s browser. A UML diagram with notes about the core library classes can also be found in the root directory under uml-diagram.png. After discussing possible implementations for the site with various tools, we have decided on the following:

1. The project will implement the client-server model with a web browser (client) and a remote, centralized webserver.
2. The client should be tested for use with the latest versions of Chrome and Firefox, but if time permitting, also test against Internet Explorer 9 and 10.
3. JavaScript and jQuery will be used on the client side to facilitate real-time updates via [AJAX](http://en.wikipedia.org/wiki/Ajax_(programming)) and [websockets/comet](http://en.wikipedia.org/wiki/Websockets). [Comet](http://en.wikipedia.org/wiki/Comet_(programming)) (also known as reverse AJAX) refers to data that is sent from a server originated event to browsers without a user refreshing the page. This is the reverse of AJAX, where a user’s browser is the originator of an event and the server cannot perform any actions until it receives such an event from the user’s browser (like clicking a form’s button).

Websockets are used for the same reason as comet, but are built into newer browsers, while comet is implemented by developers using specialized JavaScript code to manually send/receive new distributed events. Use cases for websockets/comet are web apps that connect multiple users simultaneously and give the ability for each user to see what the other user is doing without refreshing the webpage. Examples of webapps that use websockets/comet are:

1. Website Chat Applications (Facebook, Google Talk, Google Hangouts)
2. Whiteboards/collaborative editing (Google Docs, Office 365)
3. Twitter
4. The site mockup and formal design will be done with HTML/CSS and Twitter Bootstrap. Bootstrap is just helper CSS and HTML to help organize a site layout better and gives basic building blocks (design patterns) that are common to any site. There’s also lots of premade bootstrap templates we can use as a base to make the design easier since time is limited.
5. The web server will be [Tornado](http://en.wikipedia.org/wiki/Tornado_(web_server)), an asynchronous, event based web server made for the type of project we are doing. It has built in libraries to connect with websockets/comet on client side and send data back and forth.
6. We discussed various languages that could be used for the server side programming and decided on Python because of various tools and libraries made for it to do asynchronous event based programming and the amount of documentation and examples out there.
7. We will be using the [Flask micro-framework](http://en.wikipedia.org/wiki/Flask_(programming)) for building a website on the server side. Flask is a minimal model view controller (MVC) web framework and one of the two main web frameworks for Python. It can also handle project related problems such as user authorization and form validation.
8. To keep data storage simple, we are going to use a key-value storage system (a [database called Redis](http://en.wikipedia.org/wiki/Redis)) set up as a [graph-like database](https://github.com/Doist/redis_graph). This way, a user’s subscriptions and other assets can be modeled as a large [graph data structure](http://en.wikipedia.org/wiki/Graph_(abstract_data_type)) that is much more simplistic and natural for the type of data than a traditional relational database.

**Code and Project Operation**

See the zip file included with the project for all source code. Also, the /doc directory has documentation related to the source code and libraries.png in the root directory show a UML diagram of the core libraries as well as notes about what each does.

**User Manual**

A manual about how to use Twic is located in UserManual.doc.

**Test Plans and Test Results**

We did a number of unit tests against our code base (unit tests are included in the source code). Python has its own built in unit test framework, simply named “unitest.py” and that was what we used. Combined with our IDE, Intellij IDEA, one can click a button and have it run through the tests automatically and tell us instantly if anything fails after it is done. We wrote out unit tests to ensure all parts of the libraries that communicated with the database were working as intended as well as tests to ensure the parser for messages was working correctly. This allowed us to avoid most bugs that could slow down a project. However, it also took a while to write out the tests and fix any issues from tests that failed. Aside from unit tests, we also ensured functionality was working correctly by doing some functional tests on the user interfaces. The attached document, functional\_tests.doc, walks the reader through the various tests we did to ensure functionality was working as intended.