Design Specification

GeoSkynet

Version 1.0

Prepared By Team BRX

Aaron Dodson (gonfunko@gmail.com)
Anthony Coccia (anthony.coccia@gmail.com)
Kenny Bier (kenbier@gmail.com)
Matthew Greenfield (mattgreenfield1@gmail.com)
Peter Huang (peterhuang913@gmail.com)

With Mentors, Aerospace

Nehal Desai (nehalmeister@gmail.com)
Setso Metodi (t.metodi@gmail.com)

Instructor

Chandra Krintz

T.A.

Stratos Dimopoulos

Course

CMPSC 189A

Date

February 26th, 2013

Table of Contents

1. Introduction

1.1 Product Overview

GeoSkynet is an application that collects live Twitter data, parses it for location (geospatial coordinates), time, and context information, then stores it for later use. Tweets are typically made by people and contain ambiguous natural language instead of explicit identifiable properties. Because of this, the problem of identifying useful information is a difficult and commonly researched problem. The system must distinguish between usable and unusable strings of text and filter through colloquialisms and abbreviations to generate accurate results while still being robust enough to not miss any important information.

GeoSkynet will initially focus on the domain of food trucks. A list of food truck Twitter accounts will be monitored for new tweets. When a new tweet is created, the system will grab the text, username, timestamp, and other metadata, process it, and store it in a database.

The data processed by GeoSkynet can be accessed via a web interface that allows querying of food trucks and particular locations. The data will be presented in a list and on a map that contains geospatial coordinates, dates, times, the context within the original tweet, and the original tweet itself.

1.2 Definitions, Acronyms and Abbreviations

- **Middleware** - a mediator between the database and the user interface
- **Natural Language** - the language in which humans speak: complex, ambiguous, and littered with idioms. Unlike mathematics, natural language is not exact and can be difficult for a computer to interpret.
- **NLTK** - Stands for “Natural Language Toolkit”; A library for Python that parses and stores natural language into data structures
- **Geostring**- The part of a tweet that contains the language that describes the location.

1.3 Technologies Used

1.4.01 Python: Python version 2.7 will be used as the base language for all code. http://www.python.org/

1.4.02 PyUnit: PyUnit version 1.4.1 will be used to unit test all Python code. http://pyunit.sourceforge.net/

1.4.03 JavaScript: JavaScript version 1.8.5 will be used to interface with the Google Maps API and for the front-end web user interface.

https://developer.mozilla.org/en-US/docs/JavaScript

1.4.04 RabbitMQ: RabbitMQ version 3.0.2 will be used as the Message Broker for communication between all components.
http://www.rabbitmq.com/

1.4.05 Pika: Pika version 0.9.9 will be used to interface with RabbitMQ in Python.
https://github.com/pika/pika/

1.4.06 Natural Language Toolkit (NLTK): NLTK version 2.0 will be used to help process natural language in tweets.
http://nltk.org/

1.4.07 MongoDB: MongoDB version 2.2.3 will be used to store all the tweets and their related data.
http://www.mongodb.org/

1.4.08 PyMongo: PyMongo version 2.4.2 will be used to interface with the MongoDB database in Python.
http://api.mongodb.org/python/current/

1.4.09 Django: Django version 1.5 will be used to handle requests from the user interface.
https://www.djangoproject.com/

1.4.10 Apache Tomcat: Apache Tomcat version 7.0.x will be used to set up the server that runs the front-end website.
http://tomcat.apache.org/

1.4.11 Google Geocoding API: Geocoding API version 3 will be used to find lat-long coordinates from a string describing a location.
https://developers.google.com/maps/documentation/geocoding/

1.4.12 Twitter API: Twitter API version 1.1 will be used to receive tweets from a list of accounts. The tweets will be sent to RabbitMQ so they can be processed and locations can be extracted.
https://dev.twitter.com/

1.4.13 Google Maps JavaScript API: Google Maps JavaScript API version 3 will be used to display the map on the website that the user will interact with. It will also be used to plot points of interest, like the user's location or locations referenced in tweets, on the map.

https://developers.google.com/maps/

2. Design Specifications

2.1 Product Modules

2.1.1 Twitter Sensors: The Sensors monitor a specific set of twitter handles in real time and pass the Tweets to the Message Broker as they arrive. Since we are focusing on food trucks, there could be a single sensor that handles all Twitter food truck traffic or multiple sensors that collect data for specific geographical regions, or even a sensor for each Twitter handle. The idea of sensors is that they are independent lightweight Python functions with the task of collecting very specific data.

2.1.2 MongoDB: The database is central repository for storing all data for offline and generally long-term analysis.

2.1.3 Message Broker: The Message Broker routes incoming sensor data to the appropriate sensor processor and is also in charge of arbitrating data exchange between the sensor processors. It is implemented using the Python-based RabbitMQ framework and Pika API.

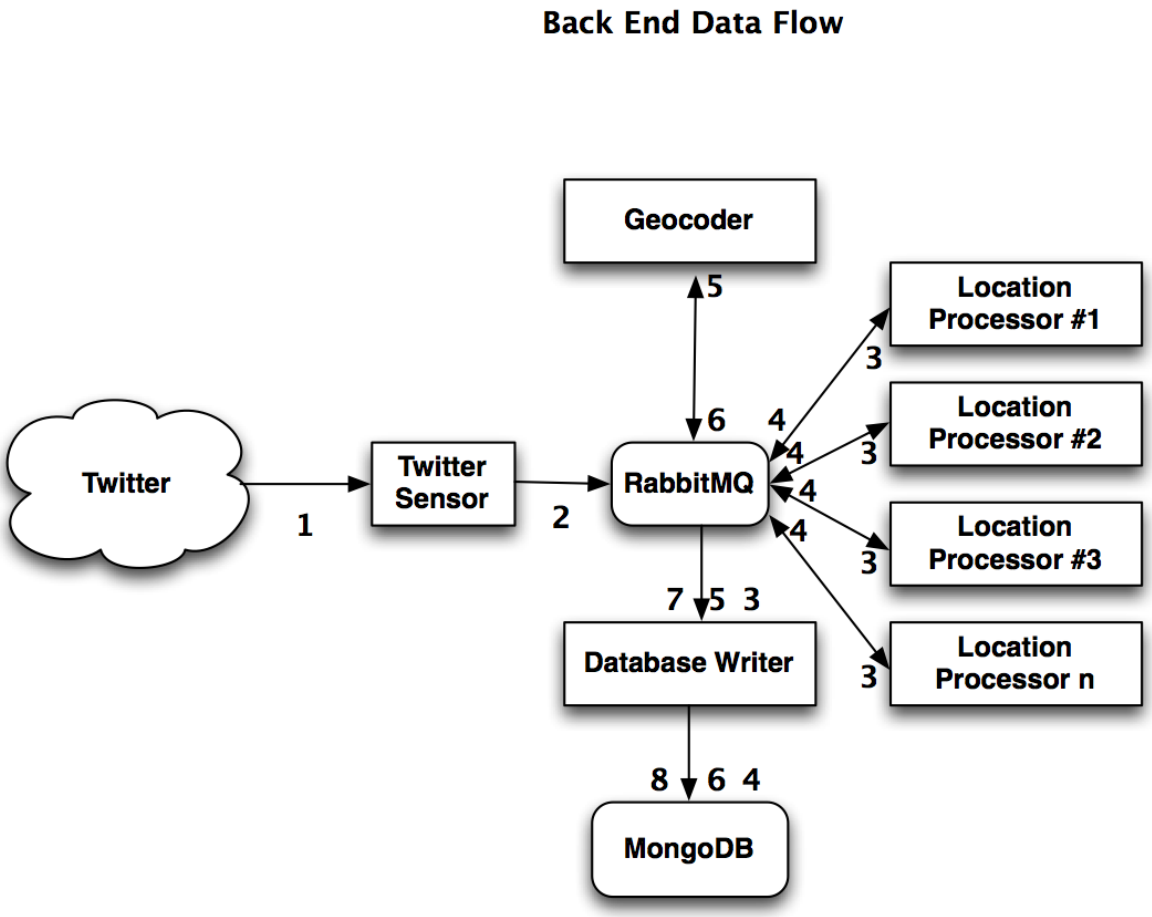
2.1.4 Sensor Processors: The Sensor Processors transform the raw sensor data to its final form(s) and pass the data through the Message Broker to other sensor processors. The sensor processors comprise a set of methods, written in Python, that Implement the data processing algorithms for extracting the geo-locations, dates, and times contained in each tweet. There is a separate sensor processor specifically tasked to route processed data back into MongoDB.

2.1.5 Middleware: The middleware implements a web framework that serves the data to the user through the front end. When the user display makes a request, the middleware interfaces directly with MongoDB to gather the requested data and feed it back to the user display.

2.1.6 User Display: The user display will allow users to specify food truck, location, date, and time information that they are interested in, then query the middleware to display the data.

2.2 Module Communication and Data Flow

2.2.1 Data Flow: Backend



The backend data flow begins at the Twitter Sensor module. As relevant tweets are posted, the Twitter Sensor receives a continuous stream of input tweets. (1) Each tweet is in the form of a JSON object, represented as a string, with the fields outlined at https://dev.twitter.com/docs/platform-objects/tweets. The Twitter Sensor sends the complete Tweet object to the RabbitMQ message broker, (2) which dispatches it in parallel to the Database Writer module and all Location Processor modules. (3) The Database Writer module writes the "raw" tweet to MongoDB. (4) It is important to note that although much of the system operates in parallel, race conditions are not a problem – a new document will be created regardless of whether a write from a Location Processor or a write from a Geocoder gets to the Database Writer first. Since none of these sources write the same fields, whichever one arrives first will create the document, and the others will append the data in the fields they are responsible for to the document.

At (approximately) the same time, the raw tweet is also dispatched to the Location

Processor modules. (3) Each one runs its heuristic, identifies what portion of the tweet (if any) it thinks is a location, and sends that data--appended to the Tweet object it received--back to RabbitMQ. (4) The data is tagged such that it is routed to both the Database Writer module and the Geocoder module. (5)

As before, when the Database Writer module receives the Tweet object--now with identified location substrings--it writes it to MongoDB (6).

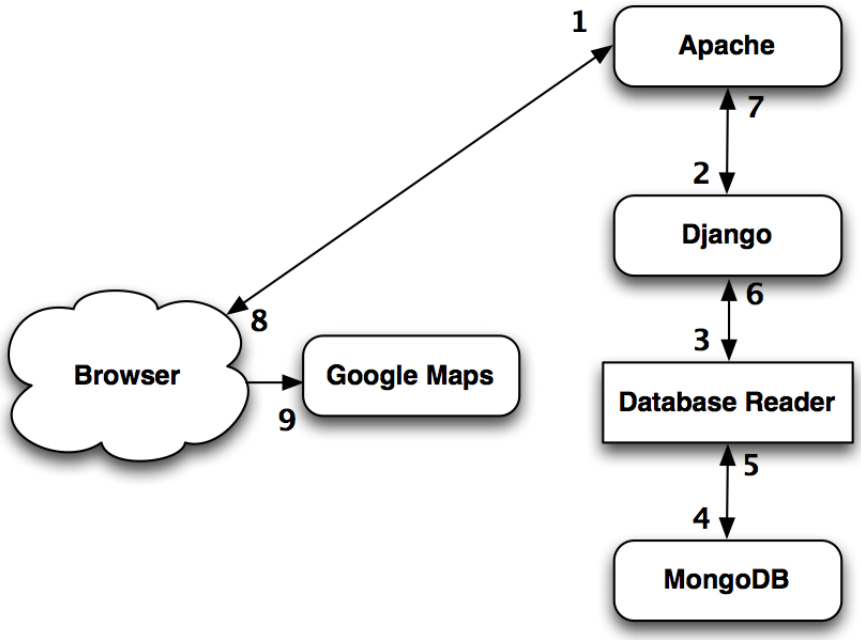
When the Geocoder receives the Tweet object with identified location substrings, it sends a request to the Google Geocoding API. Assuming coordinates are returned, it appends them to the Tweet object it received and dispatches it to RabbitMQ (6).

RabbitMQ routes the Tweet received from the Geocoder to the Database Writer (7), which writes it to MongoDB (8) as usual. At this point, the tweet is fully processed and stored in the database, along with sections identifying a location and the coordinates corresponding to that location.

Although the system is highly asynchronous, there are a few hard dependencies. Specifically, every part of the system is unable to proceed until the Twitter Sensor has received a tweet and dispatched it to RabbitMQ. Additionally, the Geocoder is dependent on at least one Location Processor having completed its task.

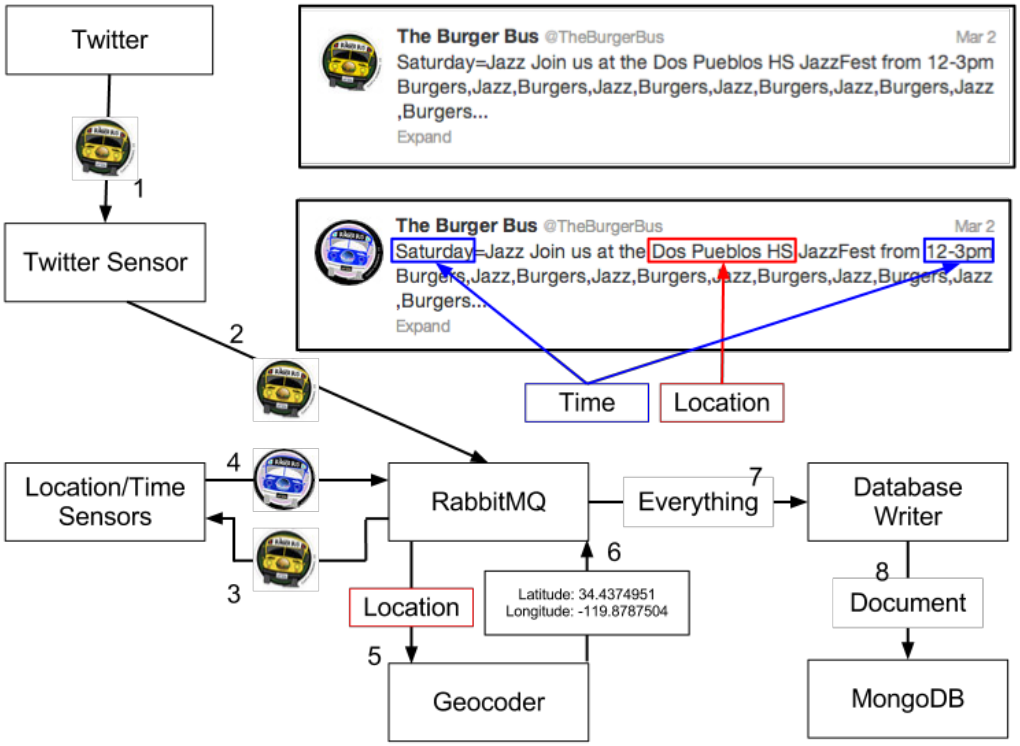
In general, the “expected” order of processing, taking into account processing time and network latency, is that a tweet is received by the Twitter Sensor, dispatched to RabbitMQ, routed to the Database Writer and written to MongoDB, sent to the Location Processors, sent to RabbitMQ, routed to the Database Writer and written to MongoDB, then sent to the Geocoder, sent to RabbitMQ and routed to the Database Writer and MongoDB.

2.2.2 Data Flow: Frontend



In contrast to the backend, the frontend of the system is synchronous. When a user visits the GeoSkynet homepage, their location will be determined using the HTML 5 location API via Javascript, which is executed automatically in response to an onload event. The Javascript will then send an XMLHttpRequest to http://www.aerotestbed.com/location/00-00-00-00-00-00, with the 00's replaced by the hours/minutes/seconds of the latitude and longitude, respectively. The Apache web server will handle the request and dispatch it to Django, which extracts the location parameters from the URL. Django will then pass these as integers in a tuple to the Database Reader module, which will calculate a 20 mile radius and query MongoDB for tweets with location coordinates within that region. MongoDB will return matching tweets to the Database Reader as JSON, which will in turn return it to Django, then to Apache, and back to the site's Javascript. The Javascript will iterate through the tweets, and plot each coordinate on the site's embedded Google Map.

2.2.3 Data Flow: Example Tweet



First the tweet is brought into the system from Twitter via the Twitter Sensor. The Twitter Sensor sends it to RabbitMQ which ships it off to the Sensor Processors dedicated to extracting the geostring.The geostring from the processed tweet is sent from RabbitMQ to the Google Geocoding API to find geographic coordinates for the location. Everything is send to the Database Writer which prepares the JSON string and send it to MongoDB.

2.3 Detailed Module Design

2.3.1 Twitter Sensor Operations:

2.3.1.1 Run: Sequentially calls initializing and starter functions.

Function Name: run()

Pre Conditions:

- None

Post Conditions:

- Either:
  - All required functions have been called
  - Some functions have been called and appropriate error messages have been logged

Details:

- Calls authorize()
- If no errors, calls get_handles()
- If the list of generated handles is not empty, calls stream()

2.3.1.2 Authorize Twitter Account: Connect to Twitter using a designated developer Twitter account that has API read permissions

Function Name: authorize()

Pre Conditions:

- None

Post Conditions:

- Sensor is authorized and connected to Twitter

Details:

- Using the python-twitter API, logs in with developer account via oAuth
    - Retries three times if unsuccessful
    - If fails, logs the error

2.3.1.3 Get Twitter Handles: Get a list of food truck Twitter handles that we'll be reading from.

Function Name: get_handles()

Pre Conditions:

- None

Post Conditions:

- A list of Twitter handles to stream from is generated

Details:

- Sensor attempts to read handle configuration file
- If configuration file exists, reads in handles separated by newline characters
- If configuration file does not exist, generates a smaller default list of handles to stream
- Stores the generated list to be streamed from that is
    - If the generated list is empty, log an error

2.3.1.4 Get Stream of Tweets: A single Twitter sensor is constantly connected to a food truck's twitter account, listening for new tweets to be made.

Function Name: stream()

Pre Conditions:

- Sensor is connected to the network
- Twitter is up
- Logged in
- Authorized
- List of Twitter handles to stream from exists

Post Conditions:

- Sensor is streaming to designated handles

Details:

- Iterates through the list of Twitter handles to stream from
- Starts streaming from as many Twitter handles as the API will allow
    - If fails, logs the error

2.3.1.5 Receive New Tweets: When a new tweet is made by a particular food truck, it is automatically sent to twitter sensors listening to that food truck.

Function Name: recv_tweet()

Pre Conditions:

- Already listening for tweets
- A new tweet is available

Post Conditions:

- New tweet is passed to MongoDB

Details:

- Handles the event of a new Tweet coming in
- Takes the JSON string from the Twitter API and splits it up into only the pieces that we are interested in (as seen in the MongoDB specifications: Username, timestamp, text, etc.)
    - If tweet does not have required pieces, log the error
- Calls the send_tweet() function

2.3.1.6 Send raw tweet data to MongoDB: After receiving a raw tweet from a twitter handle, the sensor then sends the data to MongoDB

Function Name: send_tweet(tweet: String)

Pre conditions:

- Has received a new tweet
- Mongo service is running
- There is enough Mongo memory to hold new tweet

Post conditions:

- Mongo actually received raw data tweet

Details:

- Opens a connection to MongoDB
    - Retries three times if unsuccessful
    - If fails, logs the error
- Sends the JSON string to Mongo
    - Retries three times if unsuccessful
    - If fails, logs a send_tweet

2.3.2 MongoDB Structure:

The MongoDB database will store a document for each tweet in one collection. The document will contain the following fields for each tweet:

- username
- id
- timestamp
- coord_lat
- coord_long
- tag
- location_substring
- time_substring
- expected_time
- tokenized_text

The tweets in this collection will be indexed both by id and by a (username, coordinates, time) tuple.

2.3.3 Message Broker Operations:

2.3.3.1 Get raw_tweet from Mongo

Function Name: get_tweet()

Pre conditions:

- There are raw_tweets to process

Post conditions:

- Have received raw_tweet

Details:

- Opens a connection to MongoDB
  - Retries three times if unsuccessful
  - If fails, logs the error
- Gets a new Tweet from the database
  - Retries three times if unsuccessful
  - If fails, logs a send_tweet
- Calls send_tweet(tweet: String), passing in the Tweet gathered from the database

2.3.3.2 Send raw_tweet data to sensor processor

Function Name: send_tweet(tweet: String)

Pre conditions:

- At least one sensor processor is available

Post conditions:

- raw_tweet has been sent to sensor processor

Details:

- For all sensor processors subscribed, the tweet consuming

queue, send the JSON string passed into the function to that processor
    - If any of message passing fails, logs the error

2.3.3.3 Listen for partially processed tweet from sensor processor

Function Name: listen(tweet: String)

Pre conditions:

- Listening job is working

Post conditions:

- None

Details:

- Listens for messages added onto the tweet listening queue from various sensor processors
- When a new message is added into the queue, calls update(tweet: String), passing in the tweet received

2.3.3.4 Send partially processed tweet to sensor processor

Function Name: update(tweet: String)

Pre conditions:

- Have partially processed tweet to send
- Sensor processor is available

Post conditions:

- Partially processed tweet has been sent to sensor processor

Details:

- Creates a new JSON string that holds the only the primary key of the document in the database, and the new information to be appended to the document.
- Sends the request to MongoDB to update the document
    - Retries three times if unsuccessful
    - If fails, logs the error

2.3.4 Sensor Processor Operations:

2.3.4.1 Initialize: Initializes required connections and variables

Function Name: init()

Pre conditions:

- None

Post Conditions:

- Subscription to RabbitMQ's processed Tweet queue is established
    - Retries three times if unsuccessful
    - If fails, logs the error

Details:

- Attempts to subscribe to the processed Tweet queue
- Generates any data required for processing

2.3.4.2 Receive raw_tweet from RabbitMQ: When RabbitMQ has a new tweet that needs to be processed, it will send it to one or many Sensor Processors to be processed.

Function Name: recv_tweet(tweet: String)

Pre conditions:

- RabbitMQ has a new raw_tweet that needs to be processed

Post conditions:

- The Sensor Processor receives the raw_tweet

Details:

- Gets a new tweet from the RabbitMQ queue
- Calls the send_tweet(tweet: String) function with the new tweet received

2.3.4.2 Send processed tweet to RabbitMQ: When the Sensor Processor has finished processing a raw_tweet, it will send the geostring and the tweet id to RabbitMQ to be forwarded to the Data Writer.

Function Name: send_tweet(tweet: String)

Pre conditions:

- The Sensor Processor has identified a possible geostring.

Post conditions:

- RabbitMQ receives the geostring and id to pass to the Data Writer.

Details:

- Sends a JSON string of the processed tweet to RabbitMQ's processed tweet queue
    - Retries three times if unsuccessful
    - If fails, logs the error

2.3.4.3 Perform Operation: When a Sensor Processor has a raw_tweet to process, it will use its perform_operation method to extract a possible geostring, timestring, or other useful information.

Function Name: perform_operation(input: String)

Pre conditions:

- The Sensor Processor has received a raw_tweet from RabbitMQ.

Post conditions:

- The Sensor Processor has identified a geostring for the raw_tweet.

Details:

- Each sensor processor will be responsible for executing certain heuristics on a tweet. Some of these include:

        - Parsing address strings
        - Parsing area code strings
        - Parsing city/state strings
        - Parsing prepositional phrases
        - Looking through Twitter profiles for additional information
        - Inferring information based on #hashtags
        - Parsing keywords and inferring locations
    - After the sensor processor is done processing the street, it calls send_tweet(tweet: String) to return any additional information it gathered during processing.

2.3.5 Middleware:

2.3.5.1 Get food truck request: Get a request from the user display to look up location and time information for a particular food truck.

Function Name: get_foodtruck(truck: String)

Pre conditions

- User display has sent request to receive food truck information

Post conditions

- Request is sent to MongoDB

Details:

- Creates a MongoDB query that will return specific food truck data
- Calls request(query: String) with the generated query
- Calls send(result: String) with the result returned from the request

2.3.5.2 Get specific location request: Get a request from the user display to look up food truck location and time information on all food trucks in a specified location.

Function Name: get_location(location: String)

Pre conditions

- User display has sent request to receive specified location information

Post conditions

- Request is sent to MongoDB

Details:

- Creates a MongoDB query that will return specific location food truck data
- Calls request(query: String) with the generated query
- Calls send(result: String) with the result returned from the request

2.3.5.3 Get current location request: Get a request from the user display to look up food truck location and time information on all food trucks at the user's

current location.

Function Name: get_current_location(location: String)

Pre conditions:

- User display has sent request to receive current location information

Post conditions:

- Request is sent to MongoDB

Details:

- Creates a MongoDB query that will return specific location food truck data
- Calls request(query: String) with the generated query
- Calls send(result: String) with the result returned from the request

2.3.5.4 Request data from MongoDB: Using the PyMongo API, the middleware queries the MongoDB database for information based on the criteria provided from the user display

Function Name: request(query: String)

Pre conditions

- A request from the user display has been made
- MongoDB service is running

Post conditions

- A result is ready to be sent back to the user display (potentially empty)

Details:

- Opens a connection to MongoDB
  - Retries three times if unsuccessful
  - If fails, logs the error
- Sends query to MongoDB
  - Retries three times if unsuccessful
  - If fails, logs a send_tweet
- Calls send(result: String[]), with the resulting JSON string array

2.3.5.5 Send information to user display: Send results of MongoDB query back to the user display

Function Name: send(result: String[])

Pre conditions:

- MongoDB sent back a query result (potentially empty)
- The server is running

Post conditions:

- The result is sent back to the user display

Details:

- Sends an array of query result JSON strings to the user display
    - Retries three times if unsuccessful
    - If fails, logs the error

2.3.6 User Display Operations:

2.3.6.1 Send food truck request: Send a request to receive location and time information based on a particular food truck.

Function Name: get_foodtruck(truck: String)

Pre conditons:

- A food truck is selected

Post conditions:

- If available, location and time information is displayed
- Otherwise, an error message is displayed that no results were found

Details:

- Creates a JSON string request with the designated food truck
- Submits the request to the middleware
    - Retries three times if unsuccessful
    - If fails, logs the error and displays the error “Could not get current location.”
- Passes the results to process_results(results: String[])

2.3.6.2 Send specific location request: User sends a request to receive food truck location and time information on all food trucks in a specified location.

Function Name: get_location(location: String)

Pre conditons

- A location is specified

Post conditions

- If available, location and time information is displayed
- Otherwise, an error message is displayed that no results were found

Details:

- Creates a JSON string request with the specified location
- Submits the request to the middleware
    - Retries three times if unsuccessful
    - If fails, logs the error and displays the error “Could not get current location.”
- Passes the results to process_results(results: String[])

2.3.6.3 Send current location request: User sends a request to receive food truck location and time information on all food trucks at the user’s current location

Function Name: get_current_location()

Pre conditions

- The browser is able to get the user's current location

Post conditions

- If available, location and time information is displayed
- Otherwise, an error message is displayed that no results were found

Details:

- Attempts to get the users current location
    - Retries three times if unsuccessful
    - If fails, logs the error and displays the error "Could not get current location."
- Creates a JSON string request with the determined location
- Submits the request to the middleware
    - Retries three times if unsuccessful
    - If fails, logs the error and displays the error "Could not connect to server."
- Passes the results to process_results(results: String[])
- Upon receiving result from the middleware, parses the results into pin objects, bubble objects, and left panel objects.
- Displays the food trucks on the left panel
- Displays pins on the map
- The bubbles remain hidden until the associated pins are clicked

2.3.6.4 Process Results: After receiving a request result from the middleware, map and other display objects are created and shown on the page.

Function Name: process_results(results: String[])

Pre conditions:

- A result was returned from the middleware

Post conditions:

- A collection of user interfaces objects is generated

Details:

- Parses and groups the results by food trucks and locations.
- For a unique food truck and geospatial location, a single pin and hidden bubble object is created.
- Food truck, location, and tweet information is added to the bubble.
- For every time the food truck plans to visit that location, another datetime is added to the associated bubble.
- For each unique food truck (not split by individual locations), an item is added to the left panel.
- For each location and time the food truck plans to make an appearance, an initially hidden description line is added to the item

on the left panel.
- Calls display_objects(objects: Object[])

2.3.6.5 Display Objects: Adds pin, bubble, and, panel objects to the user display.

Function Name: display_objects(objects: Object[])

Pre conditions:
- Pin, bubble, and panel objects were generated.

Post conditions:
- User interface objects are displayed on the screen and ready to interact with the user

Details:
- For each pin/bubble pair, a simple call to the Google Maps API to add a pin and bubble is called.
- For each panel object, a new object is added to the panel on the left side of the page.

2.3.6.6 Pop-Up Bubble: When a pin on the map is clicked, the information bubble associated with that pin is displayed. This operation is completely handled by the Google Maps API.

2.3.6.7 Expand Food Truck Info: When a food truck item in the panel on the left side of the screen is clicked, the item expands in an accordion-like fashion to give more details about the food truck.

Function Name: expand()

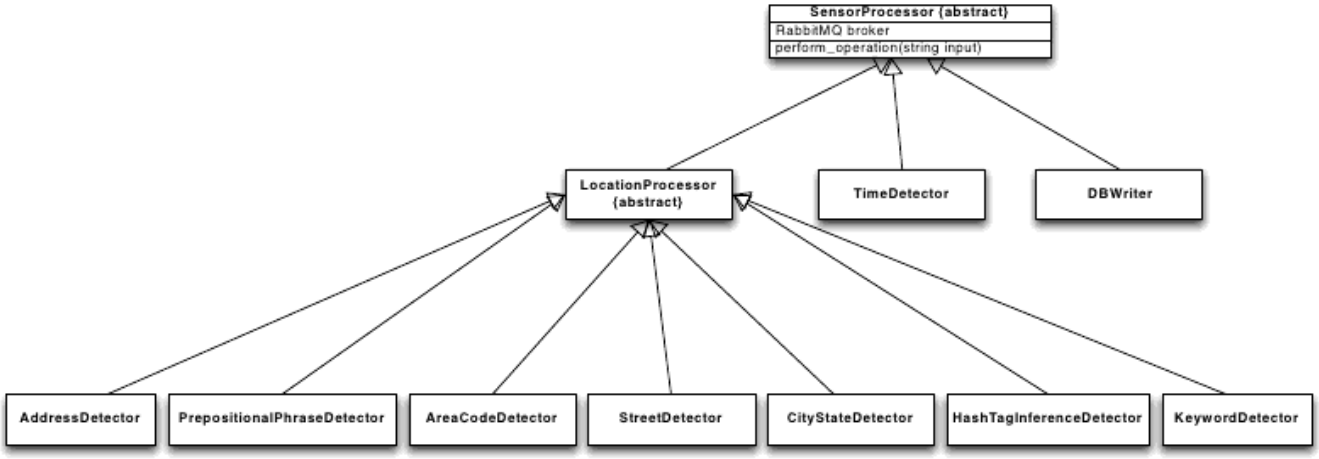
Pre conditions:
- None

Post conditions:
- The food truck item expands
- The user interface is not cropped or distorted

Details:
- Uses basic Javascript and CSS accordion methods of expanding list elements to display the additional information original hidden within the panel object.

2.4 UML Diagrams

Sensor Processor Class Hierarchy



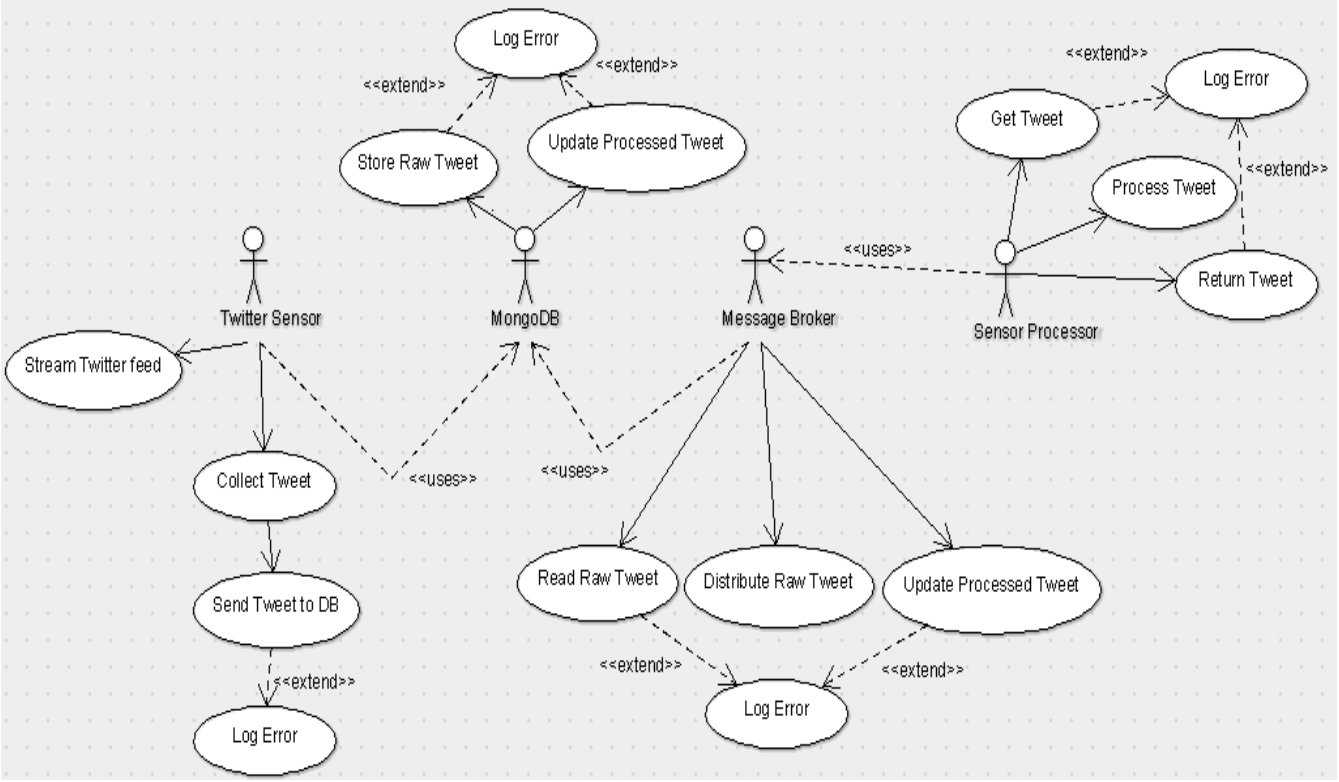
With the exception of the Twitter Sensor and Database Reader, all other backend modules are instances of the SensorProcessor class. SensorProcessor is an abstract class that exists to defines a common communication interface between all SensorProcessors, which communicate and interact through the RabbitMQ message broker. Every SensorProcessor has a reference to the message broker, and each SensorProcessor implements only one public method: perform_operation, which takes as input a JSON representation of a tweet as a string.

LocationProcessor is an abstract subclass of SensorProcessor. It is responsible for registering with RabbitMQ for tweets that need location processing, and for providing a common subclass so that the various concrete implementations can be referred to without specifying exactly which one - that is, it mostly exists for polymorphism. Each heuristic is a concrete subclass of LocationProcessor. These modules implement perform_operation to identify any substring(s) in the tweet their heuristics indicate may be locations, then append these to the tweet JSON string and add it to Rabbit's queue for further processing.

DBWriter is a concrete subclass of SensorProcessor. It communicates with MongoDB via PyMongo, and is responsible solely for extracting the values in the JSON string it receives in perform_operation and writing them into the database. Unlike the other sensor processors, it does not place anything into the RabbitMQ queue itself.

2.5 Use Cases

2.5.1 Back-End Use Cases



Actors:

Twitter Sensor

Uses Cases:

- Stream Twitter Feed
  - Listen for new tweets on a Twitter handle
- Collect Tweet
  - Process a newly created tweet and put it into a nice format
- Send Tweet to DB
  - Send tweet JSON to database
  - If could not connect to database → Log Error
- Log Error

Dependencies:

- MongoDB

MongoDB

Uses Cases:

- Store Raw Tweet
  - Store tweet sent from Twitter sensor into tweet collection
  - If could not store the document → Log Error
- Update Processed Tweet
  - Update the original tweet record with additional information gathered by sensor processors
  - If could not update the document → Log Error
- Log Error

Dependencies:

- None
  - MongoDB does not depend on any other module in the system. It simply handles insert, update, and select requests from other modules and returns results

Message Broker

Uses Cases:

- Read Raw Tweet
  - Makes a call to MongoDB to get any new tweet
  - If could not reach MongoDB → Log Error
- Distribute Raw Tweet
  - Puts the tweet into a queue for sensor processors to read
  - **Note:** The Message Broker does not actually **send out** the tweet. It simply adds to the pool of available tweets from which subscribed sensor processors can read.
  - If If could not reach MongoDB → Log Error
- Update Processed Tweet
  - Makes a call to MongoDB to update a tweet document with the updated information from the sensor processors
  - If could not reach MongoDB → Log Error
- Log Error

Dependencies:

- MongoDB
  - Database for reading and updating tweet records

Sensor Processor

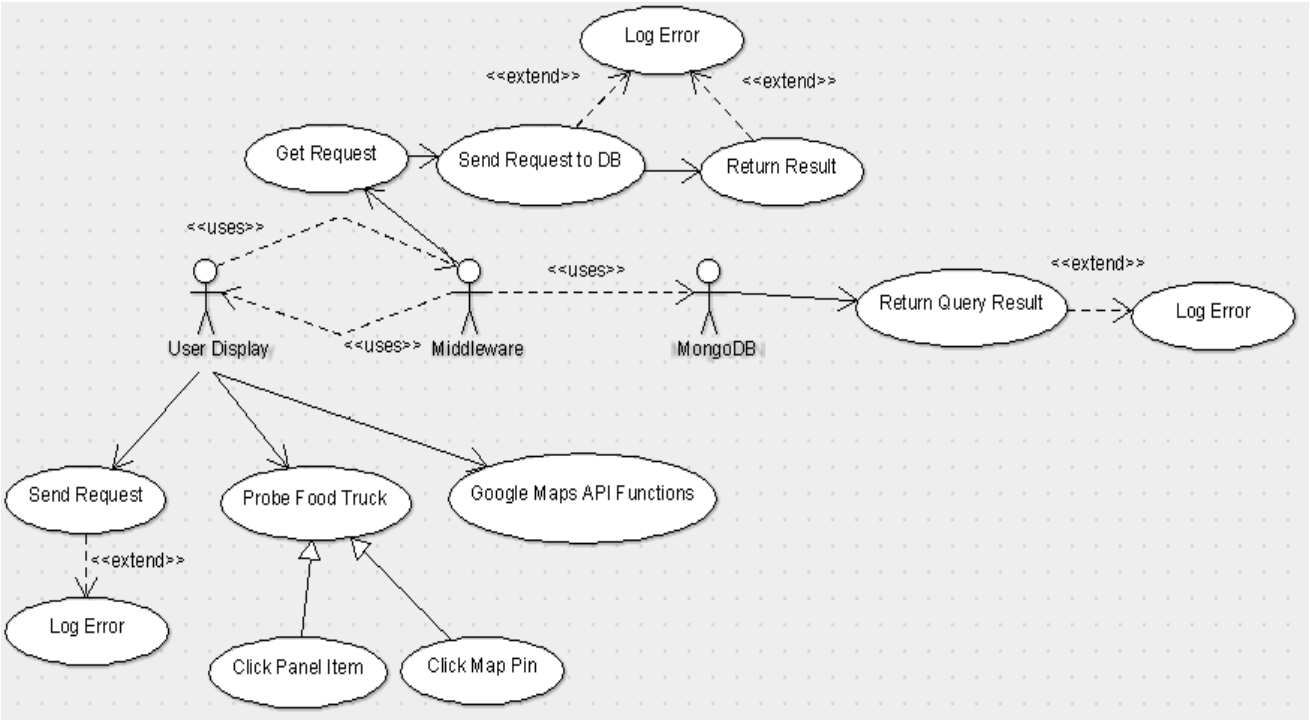
Uses Cases:

- Get Tweet
  - Read a new tweet from the Message Broker's pool of new tweets
  - If could not read from Message Broker → Log Error
- Process Tweet
  - Perform designated operations on tweet
- Return Tweet
  - Return the processed tweet to the Message Broker's queue
  - If could not reach Message Broker → Log Error
- Log Error

Dependencies:

- Message Broker
  - Queueing system to return and get tweets to and from the database

2.5.1 Front-End Use Cases



Actors:

User Display

Uses Cases:

- Send Request
  - Sends a food truck, specific location, or current location request to the Middleware
  - If cannot read server → Log Error
- Probe Food Truck
  - Find additional information about a food truck result
- Click Panel Item
  - Case of Probe Food Truck
  - Expands the list item on the left panel of the UI displaying more information about the food truck
- Click Map Pin
  - Case of Probe Food Truck
  - Shows the Google Maps API bubble displaying more information about the food truck
- Log Error

Dependencies:

- Middleware
  - The user display sends all data requests to the Middleware server, then handles all manipulation and analysis of the data on the client side

Middleware

Uses Cases:

- Get Request
  - Receive a request from the user display
  - Then uses Send Request to DB
- Send Request to DB
  - Sends the request to the database
  - If cannot reach MongoDB → Log Error
  - If successful, Returns Result
- Return Result
  - Returns the resulting data to the user display
  - If cannot return data → Log Error
- Log Error

Dependencies:

- MongoDB
  - Database holding all processed tweets
- User Display
  - While the user display relies much more heavily on the Middleware server, the server has nothing to do without a user display

MongoDB

Uses Cases:

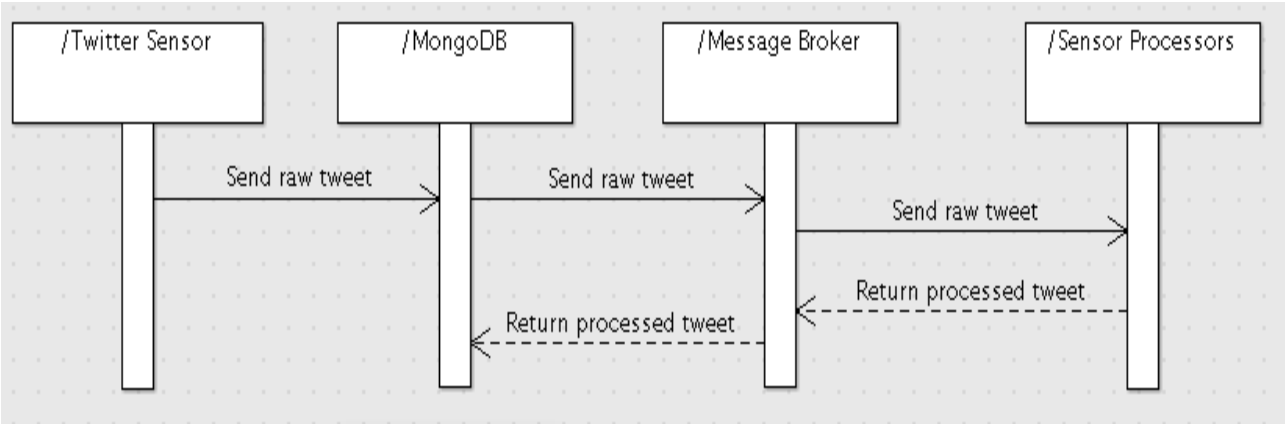
- Return Query Result
  - Returns the result of query
  - If could not run the query → Log Error
  - If could not return the result to the Middleware → Log Error
- Log Error

Dependencies:

- None
  - MongoDB does not depend on any other module in the system. It simply handles insert, update, and select requests from other modules and returns results

2.6 Timing Diagrams

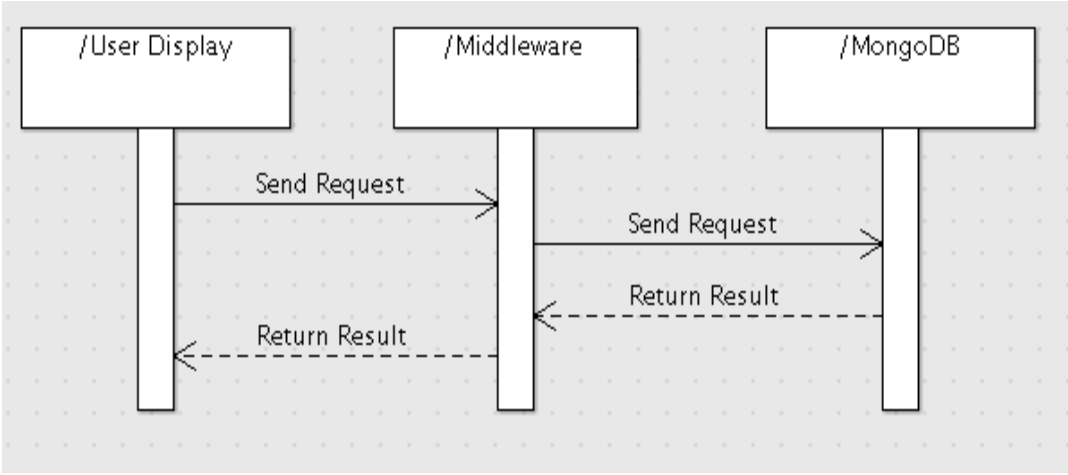
2.6.1 Back-end Timing Diagram



When a Twitter sensor receives a new tweet, it is sent to MongoDB. When a new tweet is available, it is the sent over to the message broker in the queue. The Twitter sensor sending data to MongoDB does not trigger for the tweet to passed down the chain. Instead, the message broker continuously looks for new tweets and puts them in the queue to be processed. This asynchronous separation allows for the system to adapt easily to any future changes.

When the message broker has a raw tweet in its queue, any available sensor processor can process the raw tweet and return it back to the message broker, which in turn updates the document in the database. While these actions happen sequentially, the message broker does not wait for the sensor processor to finish before performing any actions. The message broker is able to send and receive additional messages while the sensor processors perform operations.

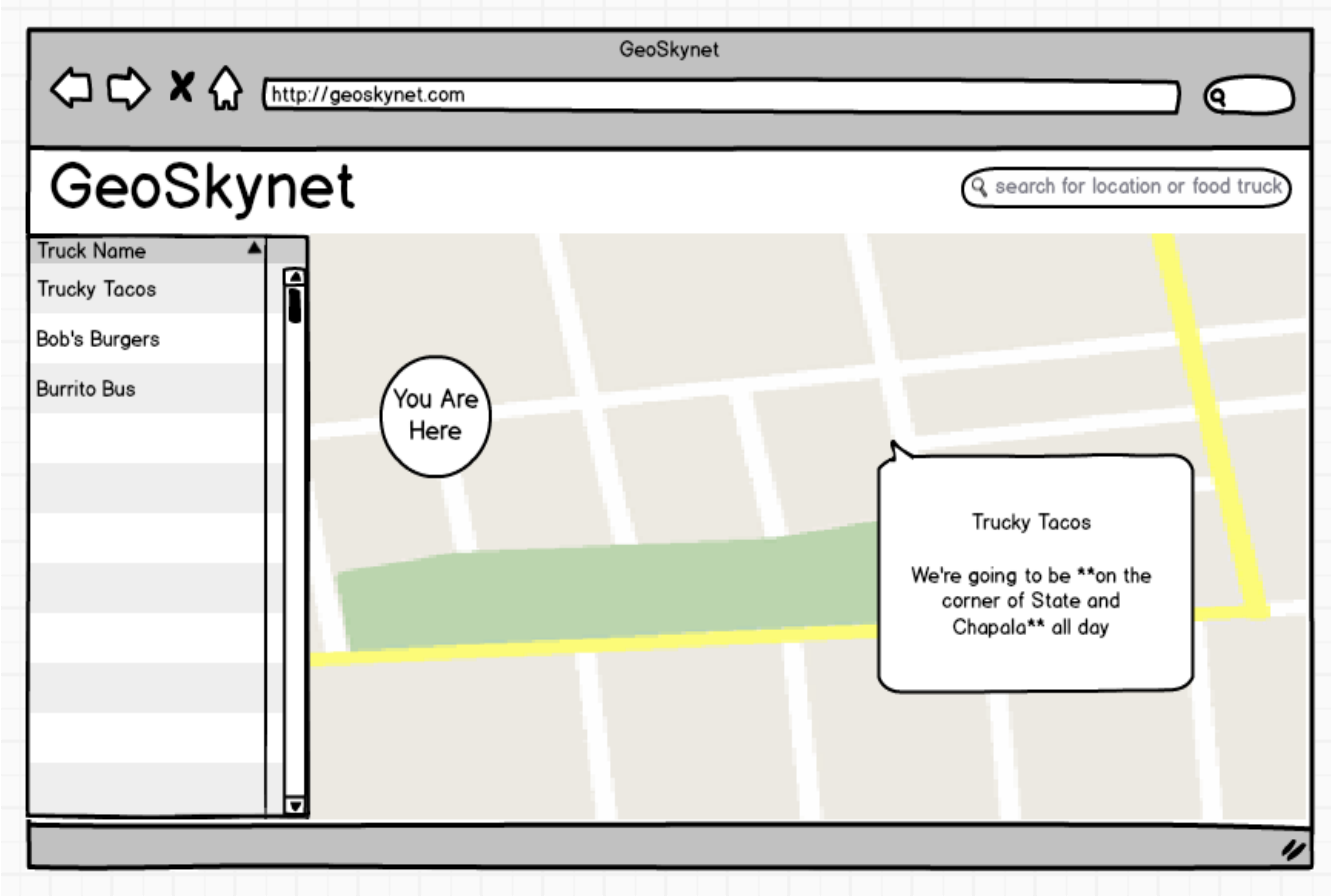
2.6.1 Front-end Timing Diagram



Unlike the back-end of the system, the components of the front-end must wait for a response before proceeding when making a request. A user is able to make a request to view food truck location information based on multiple different criteria. The request is sent to the middleware application residing on the web server. The middleware then transforms the request into a database query to return the desired data. The middleware may need to wait a short period of

time while MongoDB processes requests from other sources and generates query results. After the result is returned to the middleware, it is then passed back to be shown on the user display.

2.7 User Interface



2.7.1 First View

GeoSkynet will be available through modern web browser (including ones used on mobile devices). The site initially opens with the same view you would see on Google Maps. All of the inherent features of the Google Maps API will be available, including panning, zooming, street view, determining current location, etc.

2.7.2 Search Box

At the top right corner of the page, the user is able to type in a food truck, specific location, or “current location”, to query food truck locations.

2.7.3 Result Panel

On the left side of the page, a list of result food trucks is displayed. When clicking on an element in this pane, additional information will display: the location, time of the food truck visit, the

original tweets associated with the match, and other identifying information. The Google Maps bubble associated with the item will also activate, providing additional information on that food truck location.

2.7.4 Map Pins and Bubbles

For each food truck location, there will be a corresponding pin displayed on the map. When the user clicks on the pin, a bubble pops up that displays food truck information, more detailed location information, the different times the food truck will visit that location, and the original tweets from the food truck.

If the system is able to read the user's current location, a pin of a different color is also displayed on the map for that location. When clicking on that pin, a bubble pops up with more detailed location information.

3. Moving Forward

The initial phase of the system focuses entirely on food truck tweets on Twitter. This allows us the developers to keep focus on producing particular results from a specific data source under a specific domain. With a specific goal in mind to display food truck information that was gathered and processed from Twitter, there is a clear and comprehensive state that the application must reach before release.

However, the nature of the system allows for generalization in several areas.

- Several components are decoupled and receive messages in a queueing system, additional computing power can be applied to handle a much larger domain.
- The language and time processing algorithms are not food truck specific. The domain can be attributed to restaurants, to business, to virtually anything that could have location information.
- The methods for passing raw data to the back-end of the system are separated from the methods for collecting the data, allowing for other data sources to be used in the future (like Facebook, RSS feeds, blogs, etc.)

While the initial product has its own application, in the future (after the initial release), the product can expand to include much larger and more interesting problem sets.