Term Project - Final Documentation
Group 2
May 20, 2015

Atmospheres
Final Project for CSC 668-868 Spring 2015

Team members:
Ryan Eshleman (reshlema@mail.sfsu.edu), Kumari Sweta, Sammy Patenotte, Sree Vidya Sastry, Vishal Ravi Shankar, Harini Parthasarathi, and Eric Chu

Source Code:
https://github.com/eshlefest/atmospheres

Table of Contents

1. Contributions by team members

Sammy Patenotte:

- Webapp layout
- About us page
- Classes: geo_new.html, geo.css

Eric Chu:

- Plotly.py graph data visualization
- geo.js

Kumari Sweta:

- Setup controller for RESTful web-service to provide the aggregated data (web_service.py).
- Used jinja2 template wherever it was required (web_service.py).
- Created model and resource classes (tweet.py & sentiment.py).
- Written aggregator class to aggregate the sentiments (sentiment_aggregator.py).
- Created setup.py for packaging the application and created different commands for the application.
- Exposed method for aggregating sentiment in datastore.py.
- Debugged various issues i.e templating issue, sentiment aggregator query etc.
- Code cleanup and restructuring
- Maintained README.md file which help team members to setup their development environment.

Vidya

- Worked on the aesthetics of the webpage to improve the user interface1
- geo.js

Ryan (Team Lead)

- Designed and implemented Sentiment Analysis algorithm
- Built the interface to live Twitter feed
- Implemented data analysis and ingestion pipeline including Read, Analyze and Store phases
- Carried out MapBox + Leaflet.js integration to overlay zipcode polygons on SF map.
- Wrote geographic coordinates-to-zipcode function
- Consulted on Data Visualization strategy and technologies.

- Deployment on Apache HTTP server using mod_wsgi gateway module
- Implemented current sentiment trend indicator arrows on map.
- Classes Developed: ingestion.IngestionEngine, ingestion.SentiClassifier, ingestion.TweetStreamReader, ingestion.Properties, ingestion.Utils,db.DataStore

Vishal

- Built front-end on framework AngularJS
- Created custom html tag using AngularJS directive
- Custom controller to make an AJAX call
- Created routing for page loads using AngularJS' in-built routing mechanisms
- geo.js

Harini

Data visualization using Plotly: Implemented script to capture data to be fed to Plotly and obtain:

- Graph representation of Sentiment flow w.r.t. Time
- Bar chart representation of Sentiment w.r.t Zip-code

2. Platform and Software used

This project was using python libraries, for storing the data MongoDB is used. Here is the list of software and libraries which are used for this project.

2.1 Python packages

- **Tweepy**: This library provides ability to listen to the tweets generated from San Francisco region.
- **Nltk**: This is a natural language processing library which helped to analyze the sentiment of the posted tweets.
- **Pymongo**: This library provides an python interfaces for CRUD operation on MongoDB. This helped us to store the Tweets and aggregate the sentiments.
- **Flask**: It is a microframework for Python based on Werkzeug and Jinja 2. It helped us to create RESTful web-service.
- **Plotly:** Python package for data visualization. Provides hosted web platform to view and manipulate data.
- **Shapely**: Geometric computation package for translating GeoJSON encoded zip code boundaries into polygons and solving the Point Within Polygon problem for tweets.

2.2 JavaScript Packages

- **jQuery:** A JavaScript library that allows for an easy document manipulation and event handling. It also helps implementing Ajax requests very easily and works across a multitude of browsers
- **Bootstrap**: An HTML and CSS framework designed primarily to help developers to easily develop a responsive website. It features many defaults styles to improve the webpage's design.
- **Angular.js**: A structural javascript framework for dynamic web apps. Uses HTML as a template language and extends the HTML's syntax to express your application components easily.
- **MapBox**: A custom online map provider service that is an alternative to Google Maps.
- **Leaflet.js:** Integrates with MapBox for map overlays and interactivity. Used to draw and interact with zip code representation.

2.3 DataBase

- **MongoDB**: MongoDB is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas, making the integration of data in certain types of applications easier and faster. It is free and open-source software.

2.4 Browser debugging tool

- **Google-Chrome:** It provides tools for developer which are really helpful in debugging front-end related issues.

2.5 Web Hosting

- **Ubuntu 14.04:** Operating system for hosting web server. Chosen for reliability and robust support for web server technologies.
- **Apache HTTP server**: Industry standard server software maintained by the Apache Software Foundation. Used for exposing web applications to the public internet.
- **Mod_wsgi**: Module for hosting Python applications on Apache HTTP server.
- **Tmux**: Terminal Multiplexer for managing multiple terminal sessions on server

2.6 Editors

- **Vim**
- **Sublime**

2.7 Source control

- **Git**: It is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It was used for collaborative coding.
- **Github**: It is a cloud based source control system. We used it to manage our source code.

2.8 Miscellaneous

- **Draw.io:** Various graphical tools for creating sequence, class, and conceptual diagrams
- **Google gChat**: Chat client for intra group communication
- **Google Drive:** Document Repository

3. Project Description

3.1 Overview

Atmospheres is a Data Aggregation, Analysis and Visualization application that allows the user to explore the spatial and temporal relationships present in twitter data. The application views a tweet as an expression of sentiment, either positive or negative, and extracts the sentiment from the text with a sentiment analysis algorithm. A collection of past tweets and their associated sentiments resides in a database on the backend and is continually updated in real time as tweets are sent into the network.

The web application provides intuitive data visualization renderings in the form of choropleths (similar to a heatmap), time-series plots and bar graphs. The user is initially presented with a choropleth map of San Francisco with regions corresponding to administrative zip code boundaries. The coloring of each region corresponds to the average sentiment expressed therein. This allows the user to quickly and visually analyze the current mood of each zipcode.

The user can view a more quantitative representation of each average sentiment by clicking on the bar chart button in the upper right corner. This will render a bar chart of the sentiments for each zipcode. This bar chart is interactive and the user can use the mouse to further explore the data.

The user can also dive deeper into the sentiment trends of the individual zip code by clicking on the region. This will render a time series plot of the fluctuations of sentiment over time. Doing so, the user can observe daily and weekly trends.

All charts that are presented to the user can be exported through the plotly.com interface by clicking through the “play with the data” button.

3.2 By Function

Data Ingestion and Analysis

- Listens to live twitter stream, filtered for tweets that fall within the city of San Francisco
- Upon arrival of tweet,
    - performs sentiment analysis of text
    - assigns sentiment label
    - assigns zipcode of tweet
    - stores data in database
- leverages NLTK library to implement Naive Bayes text classification algorithm
- Uses Shapely computational geometry library to solve 'Point Within a Polygon' problem to assign zipcode membership of tweets.
- Interfaces with MongoDB for data persistence

HTTP server and Database

- HTTP Server cater two purposes. It serves the webpage as well as RESTful request.
- Controller provides various methods which are mapped to url. These methods serve the RESTful request.
- Controller methods which are RESTful return the resource object as a response.
- All tweets are store in MongoDB using Tweet model.
- Datastore is a class which is interfaced with MongoDB to store the Tweet model. It exposes various CRUD methods to manage Tweet model objects.
- ViewAggregator uses datatstore to get the aggregated sentiment objects for different zipcodes.

Web Framework

- An AngularJS app that consists of one module
- Module consists of one directive and one controller
- Directive forms the custom html tag that renders the map on the page
- Controller drives behavior into the app. It was used to make an asynchronous call to Plotly that returned GeoJSON objects

Web Layout

- Simple one-page design
- Map-centered, since it's the main functionality
- Initially had a separate page for team information, but it was moved to the bottom of the page, hidden by default, for better navigation

Data Visualization

- Used Plotly python framework to generate and create graphs to visualize data.
    - *get_plotly_zip_sentiment_series_url(x,y,filename)*
- This function takes in two arrays; one contains the zip codes that are specific to San Francisco, the other contains the specific sentiment values based on the zip code. The filename being passed into the function is the specific title for the graph which indicates the date, time, and zipcode.

- The function aggregates the data being passed into it and assigns it to specific variables or values. All of the data is then bundled and passed into the plotly framework which generates the graph externally on the plotly website. It also generates a URL which contains the graph. This can be passed onto the javascript portion of the project and embedded into the front end.

4. Challenges faced in design/implementation

4.1 Backend

4.1.1 Data

This application relies on maintaining an up to date database of the tweets in order support the subsequent analysis and visualization. Because of this, it was clear that we would be required to divide the project into at least 2 modules that run independently. One module for maintaining the database and one for providing the user facing functionality. To this end we developed the data ingestion engine which is a long running process whose actions are triggered by the receipt of a tweet from the Twitter Streaming API.

We encountered several challenges in implementing the Sentiment Analysis algorithm. Data representation is rarely an easy task and using NLTK as our algorithm library was no exception. In order for NLTK to perform its task on a given text, the text had to be transformed into a set of boolean features. One feature for every word that the classifier encountered during the training phase. The consequence is that to classify a tweet with the text “Welcome To The Jungle,” it must be transformed into a data structure of the form:

contains(Welcome): true
contains(To): true
contains(The): true
contains(Jungle): true
contains(zoo): false
contains(santa):false
etc etc for all words not in the text.

This held true for the training phase as well. This meant that the data structures were rather large quickly consumed memory. In order to solve at least the memory problem, the transformations were carried out on the fly and only as needed by a lazy loading mechanism.

4.1.2 Server and Database

While developing the application, we realized that Flask doesn’t provide support for model/datastore out of the box, unlike Ruby On Rails which provides user friendly model class and it maps the model classes to database tables.

However, Flask does provide different extensions that provide similar functionalities. We chose to use Pymongo library to create the datastore and interface it with the MongoDB. We had to define datastore and model class and map them using our own defined interfaces.
Another challenge was figuring out the right way of aggregating the sentiments based on the zipcode for a given date time. After implementing the sentiment aggregator, we realized it is not working as intended. In fact it was not returning any result. Even MongoDB was not complaining about the query string. It perplexed us for a while. After close examination of the documents(records) stored in datastore, we realized that datetime is stored as string and hence MongoDB was not able to aggregate the result. After converting it python datetime object, pymongo was able to store it in the right format. And it helped the query to fetch the required result. If MongoDB had followed the schema strictly like other DB e.g MySQL, It would have helped us identify the problem right away. Coming from SQL background gave us a steep learning curve for learning MongoDB. It has completely different construct for making the query string. It took us some time to wrap around our head around NoSQL database.

4.2 Frontend

4.2.1 Web Framework

Developing the framework from existing Javascript functions was a steep challenge. Understanding LeafletJS and refactoring existing code into the Angular app was time consuming and laborious.

4.2.2 Layout

No particular challenges faced, just some CSS and JavaScript knowledge missing, quickly fixed after going through a few online tutorials. The biggest challenge was figuring out the floating “About us” button, and making work as desired, using JavaScript and CSS.

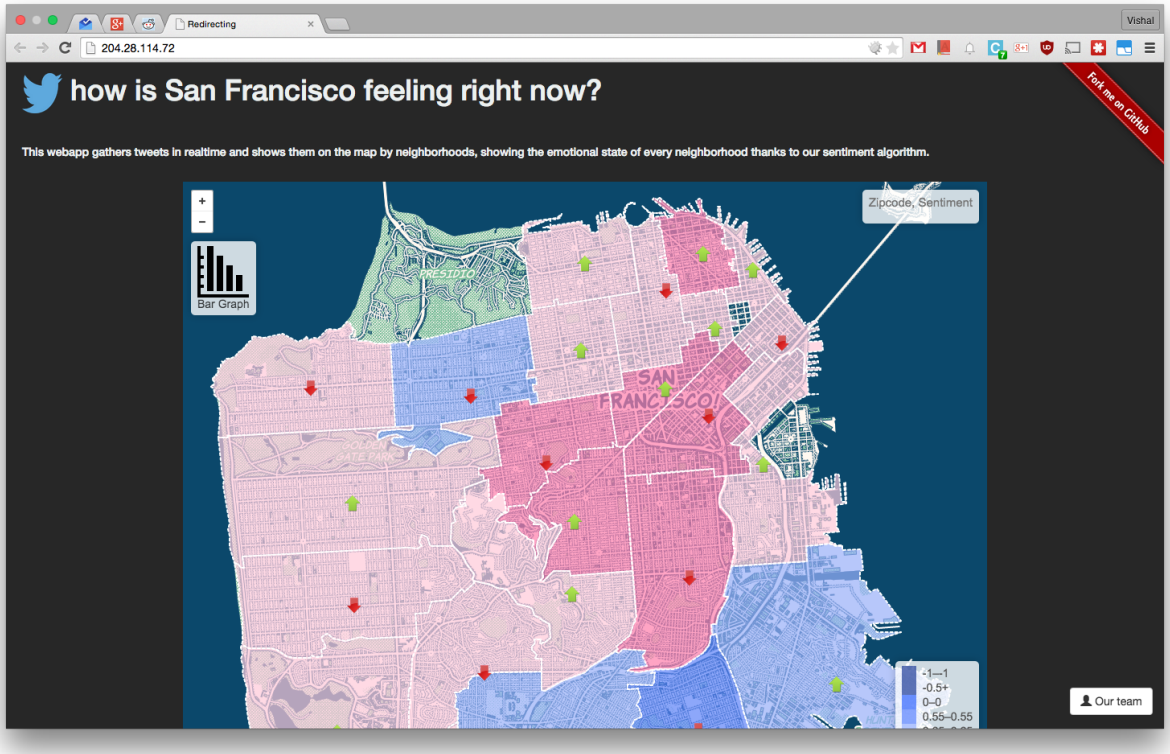
4.2.3 Data Visualization

Eric: Some challenges faced in the design and implementation portion of the project were backend knowledge and integration. Originally, Harini and I were tasked with using d3.js to generate an interactive map that contained all the data visualization. I found out how hard it was to learn and implement simple d3 functions. There wasn’t enough time to learn how to do everything, so we had to throw it out for more intuitive and easy to use solutions such as leaflet.js and plotly.py.

Harini: Script (capturing and storing parameters such as value associated with Sentiment, Time at which the text was posted and the location (neighbourhood) from which it was posted is captured and sent to Plotly which in turn returns a URL where we can view the graph. It gives us easy to use and serializable data structures, namely lists and dictionaries. Plotly gives us total control of the graph, allowing us to control styling and formatting of the GUI.

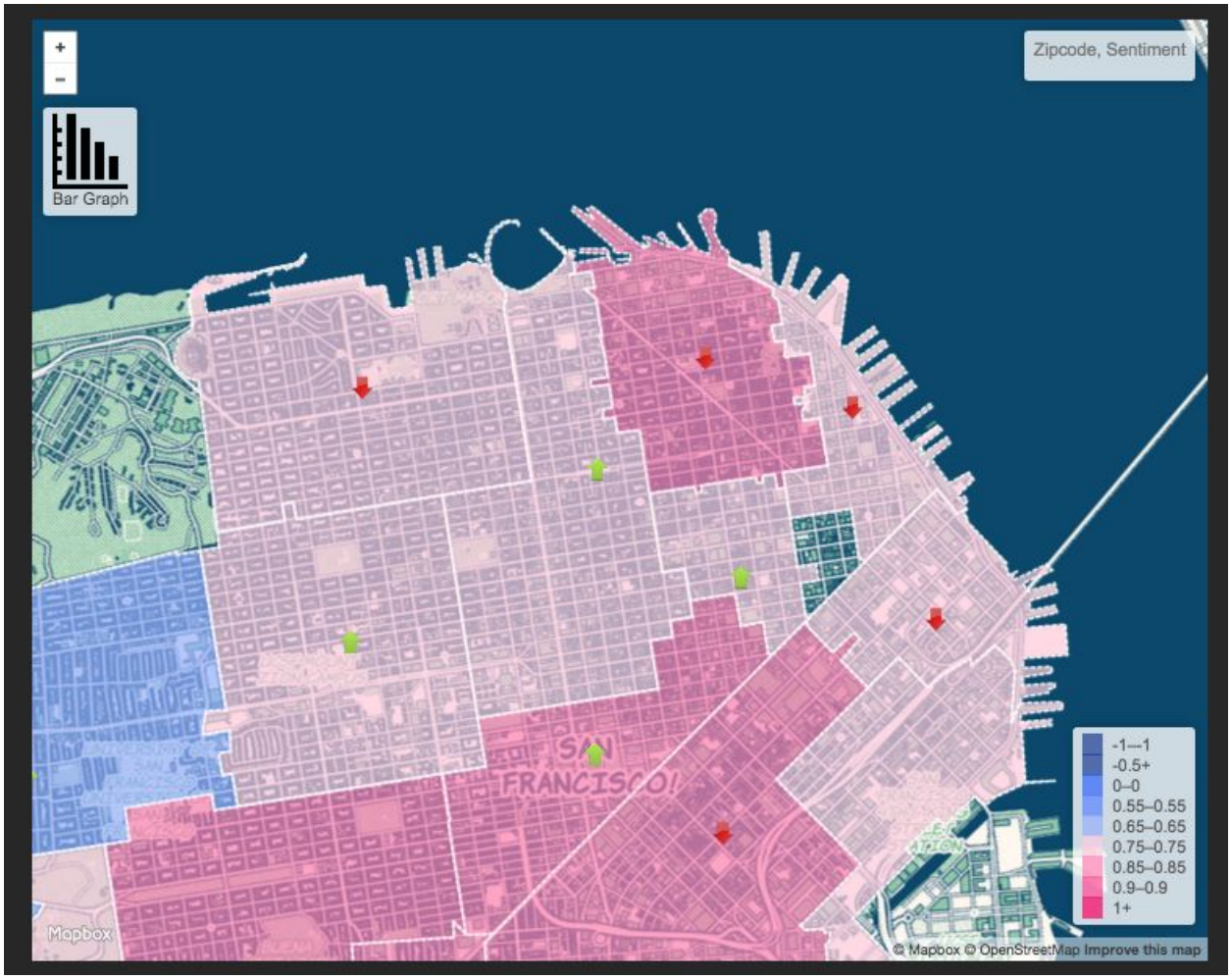
5. User Guide

Instructions are tagged with the respective screenshot showing interaction with the application:

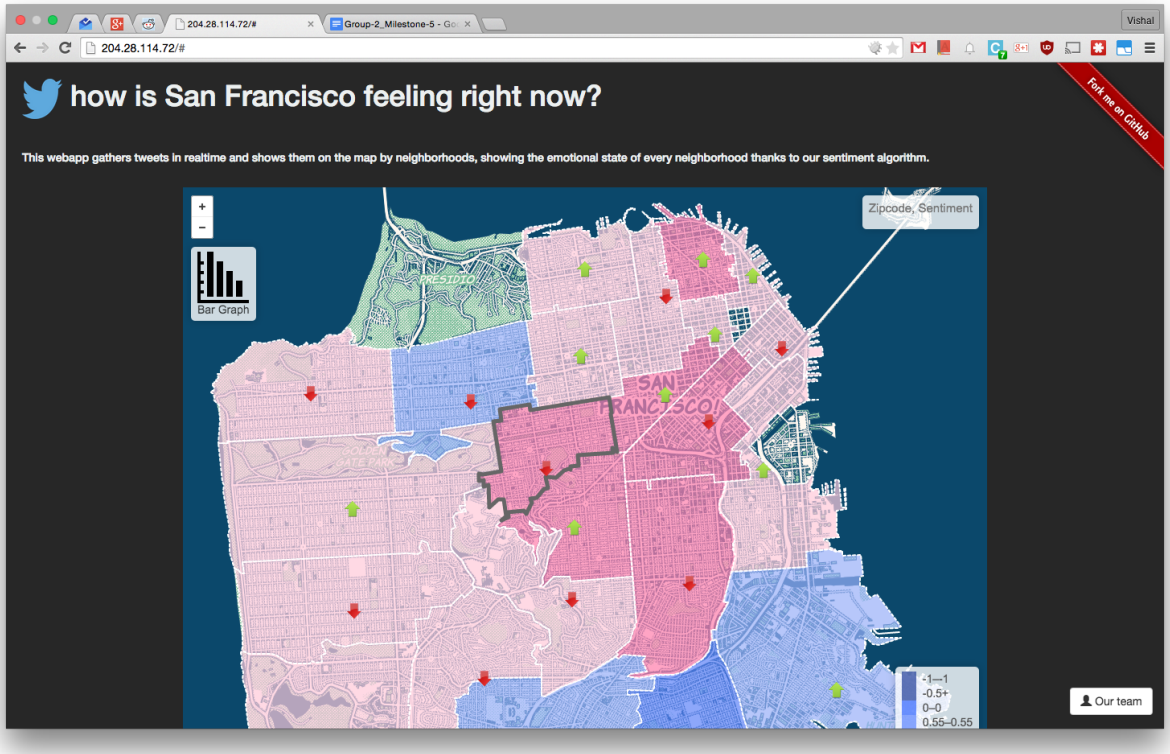


Greeted with the Main page

The user is greeted with this main page, which shows the sentiment of each zipcode based on the sentiment analysis of the tweets generated.



The map can be zoomed in to view the desired area. It can be zoomed out to display the whole map of San Francisco as desired.



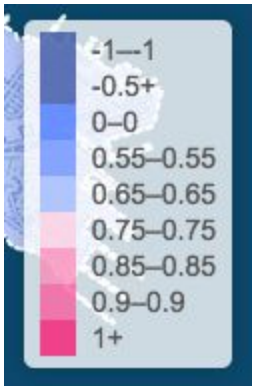
Hovering over the zipcodes highlights them

Hovering on the particular zipcode highlights them

Red arrow over a zipcode indicates that the sentiment is falling, green indicates that it is rising

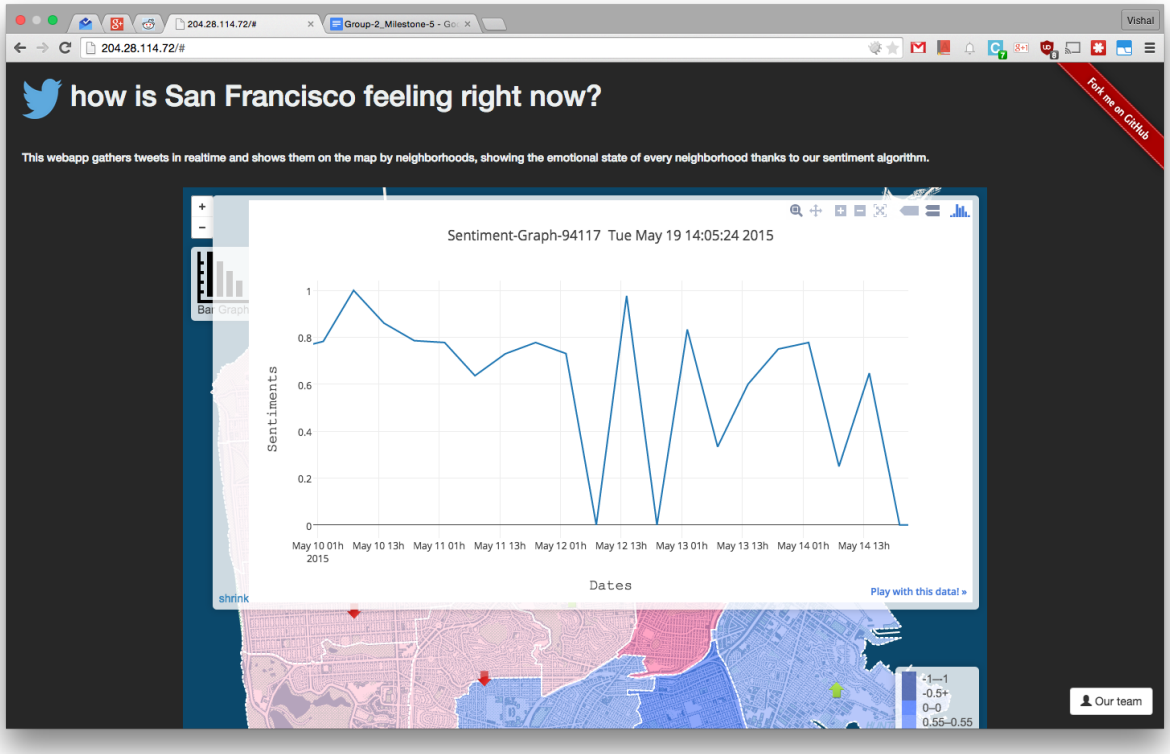
The arrows on the map indicate the change in the sentiment of each district. A red arrow indicates that the sentiment graph of that district is on a negative slope, which means that the

tweets getting generated from that district has a less happier sentiment. On the other hand, a green arrow indicates that the tweets getting generated have a have a happier sentiment.



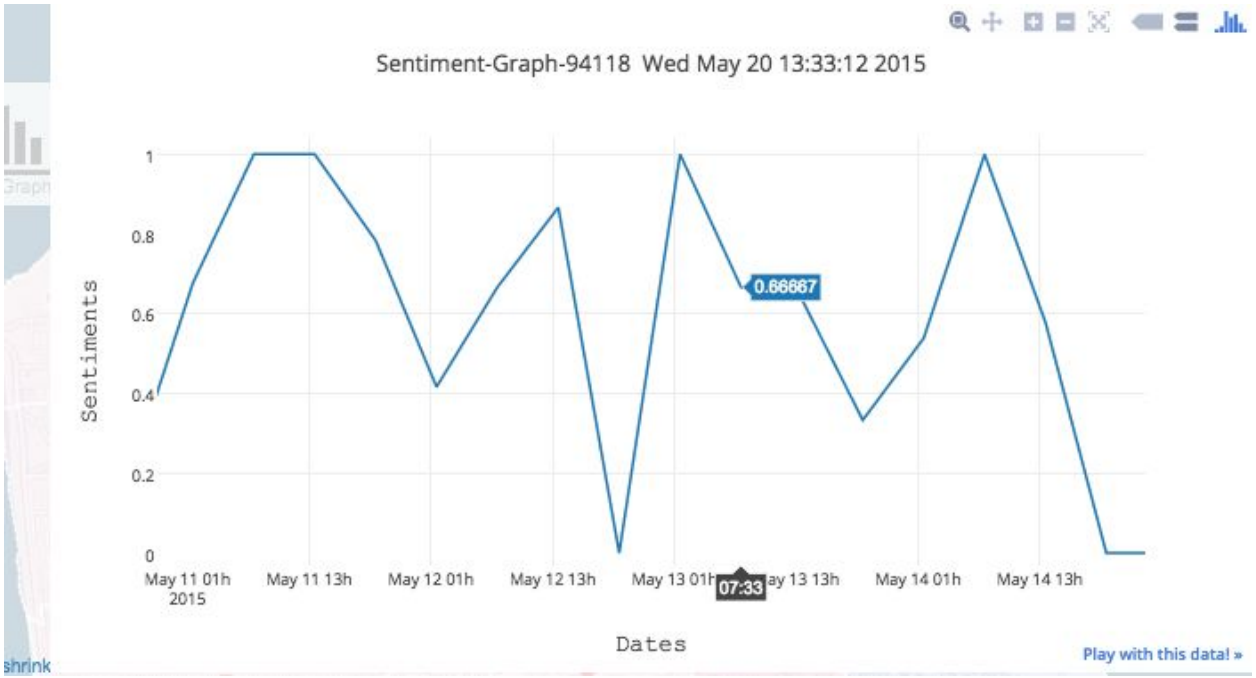
Bottom right: Sentiment range and corresponding color

The legend displays the color associated with every sentiment. The color pink represents a positive or happier sentiment and blue represents a negative or a less happier sentiment.



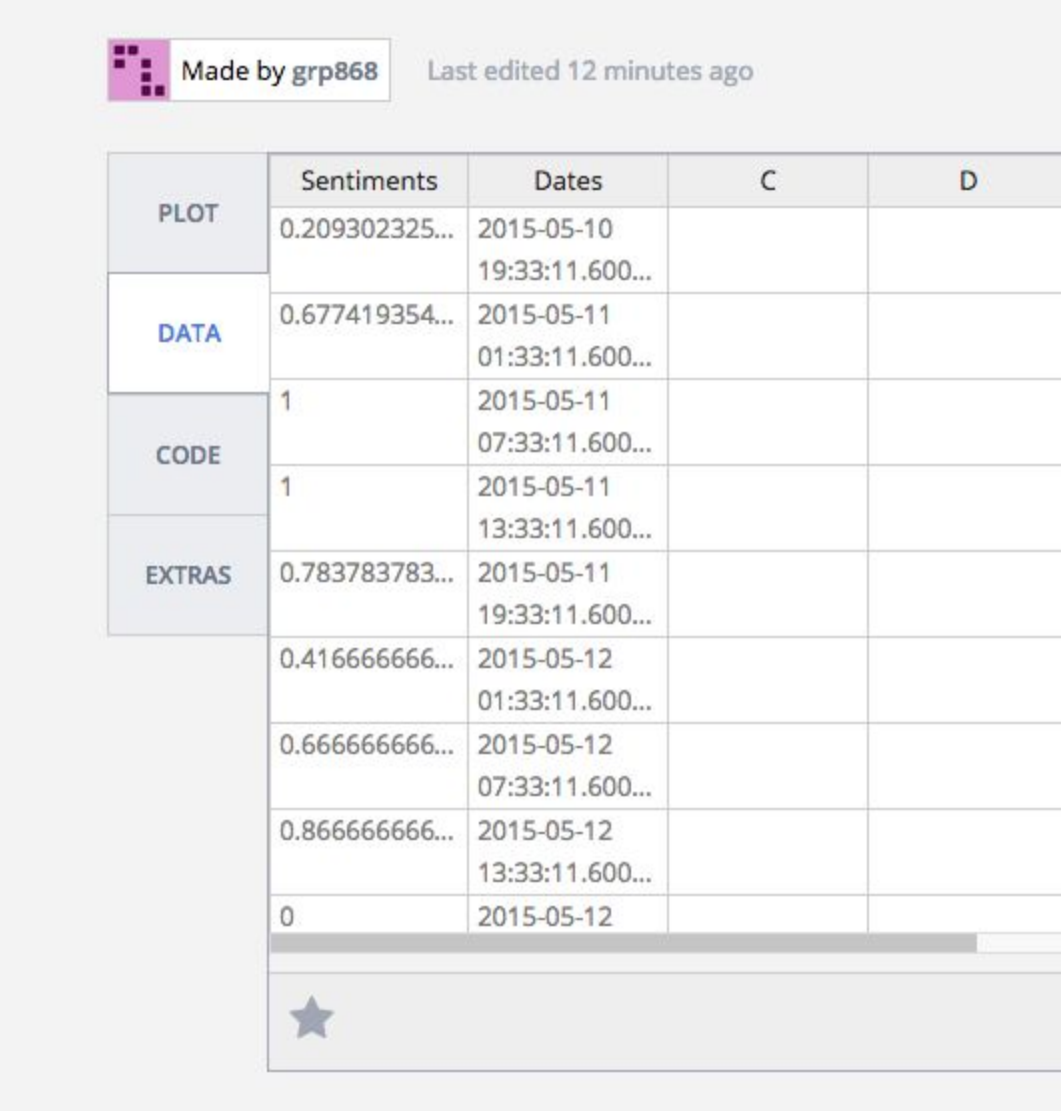
Clicking a zipcode on the map loads a graph showing sentiment trend over a course of time. Graph generated by Plot.ly. To shrink the graph, click the 'Shrink' link on the bottom left of the graph.

On clicking the zipcode on the map, a graph is generated that shows the change in sentiment over a course of time

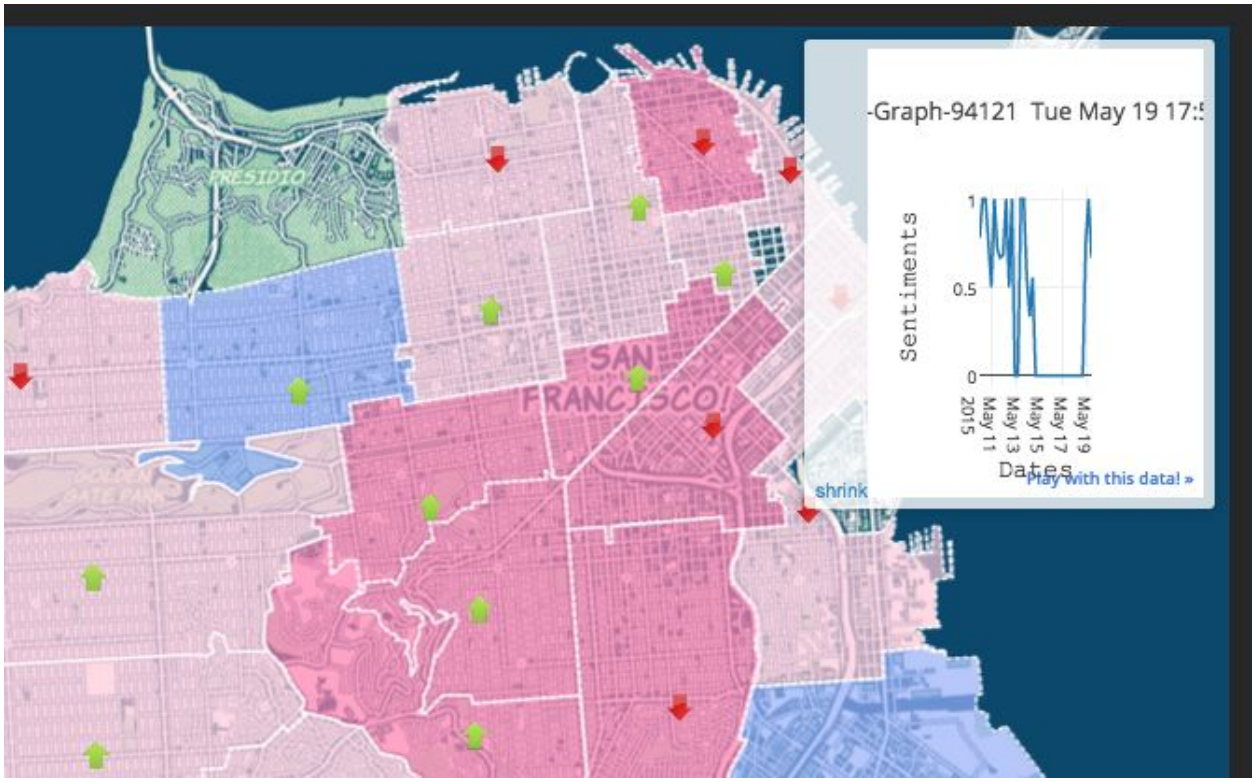


Hover mouse over chart to display details

The user can view specific data values by using the mouse to hover over the desired line segment.

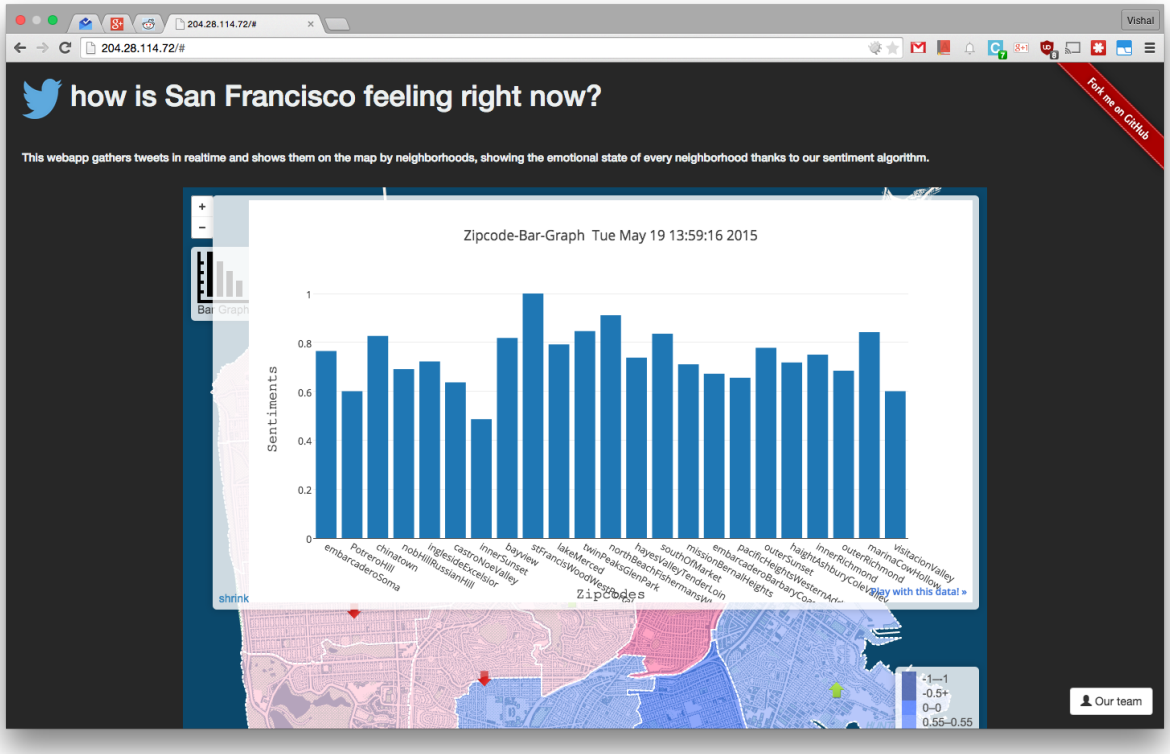


Click through the "play with this data" link to enter the plotly.com web interface and explore/download the raw data.



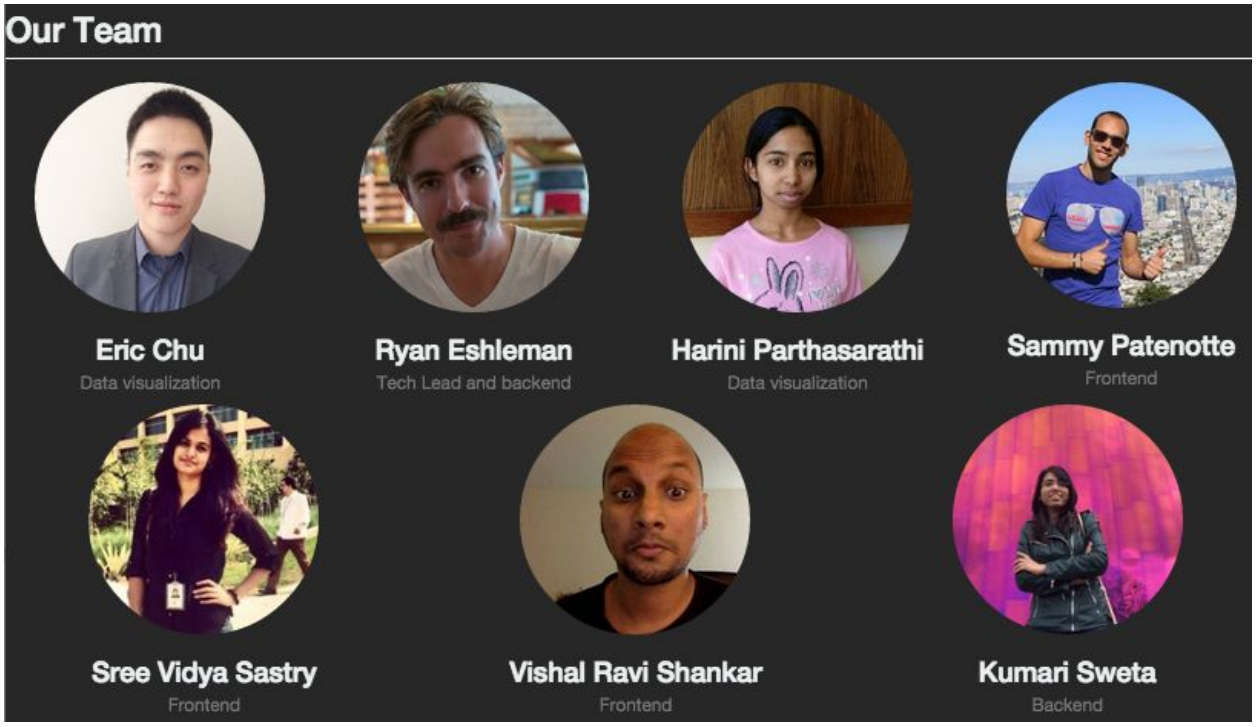
Shrink the graph to continue exploring the map

The graph can be shrunk to continue exploring the map and check the other zipcodes as desired.



Clicking  *(top-left) on the map loads the bar graph showing the sentiment value of all zipcodes*

On clicking the Bar chart icon on the left, a bar graph is generated that displays the sentiment value of all the zipcodes at a time. Hence the sentiment value can be compared for various zipcodes.

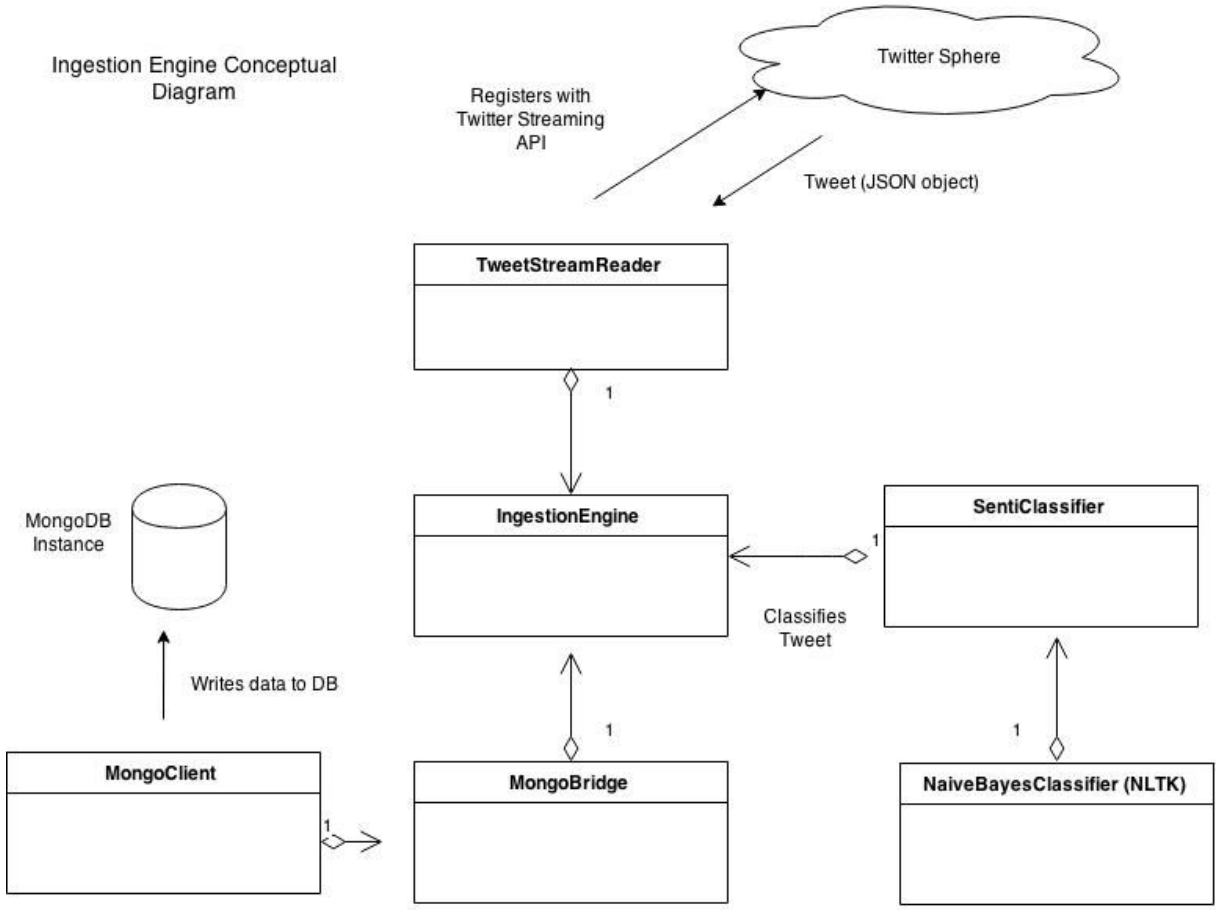


Get to know the team

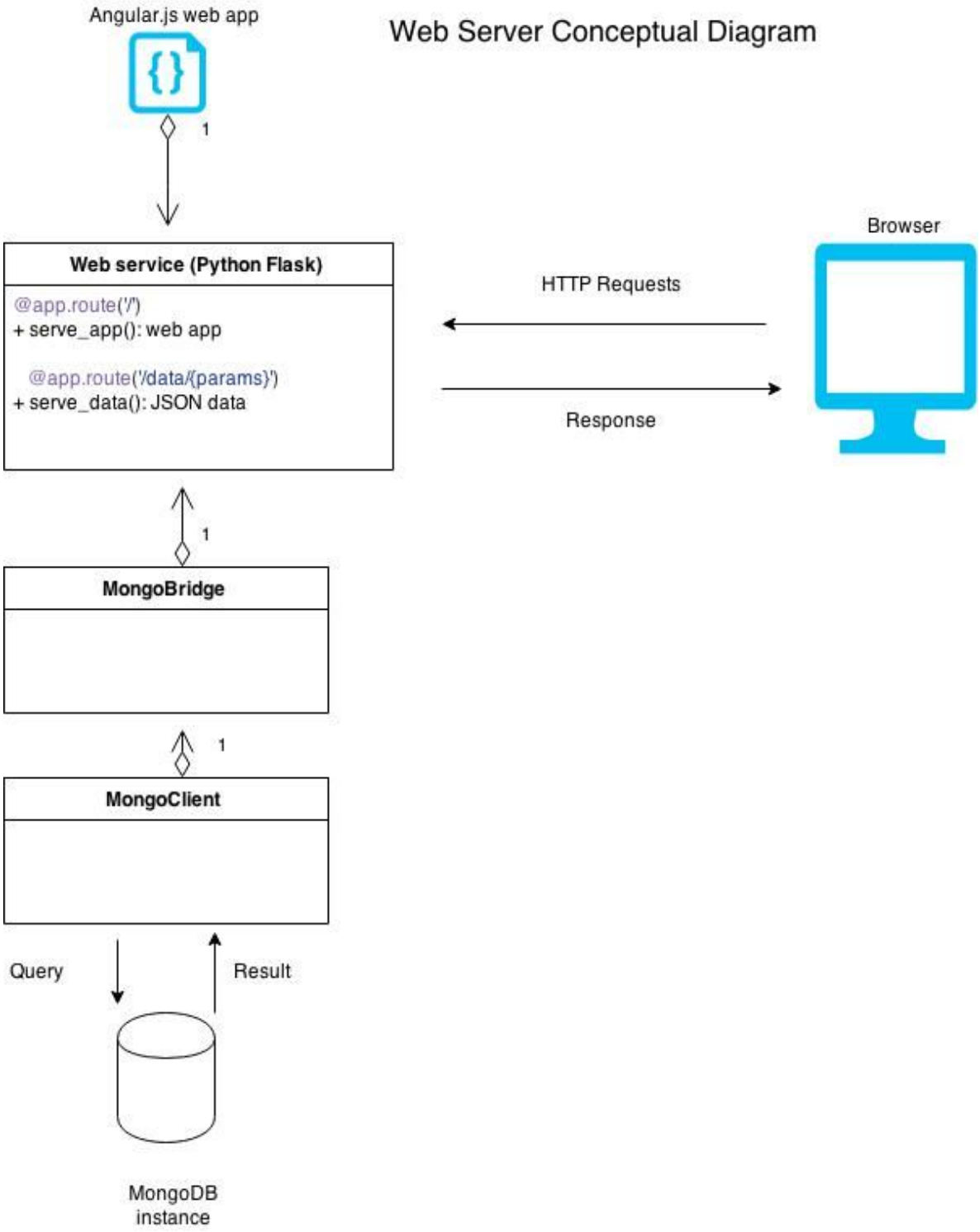
This web page displays all the team members along with their field of contribution to the project.

6. Conceptual Models

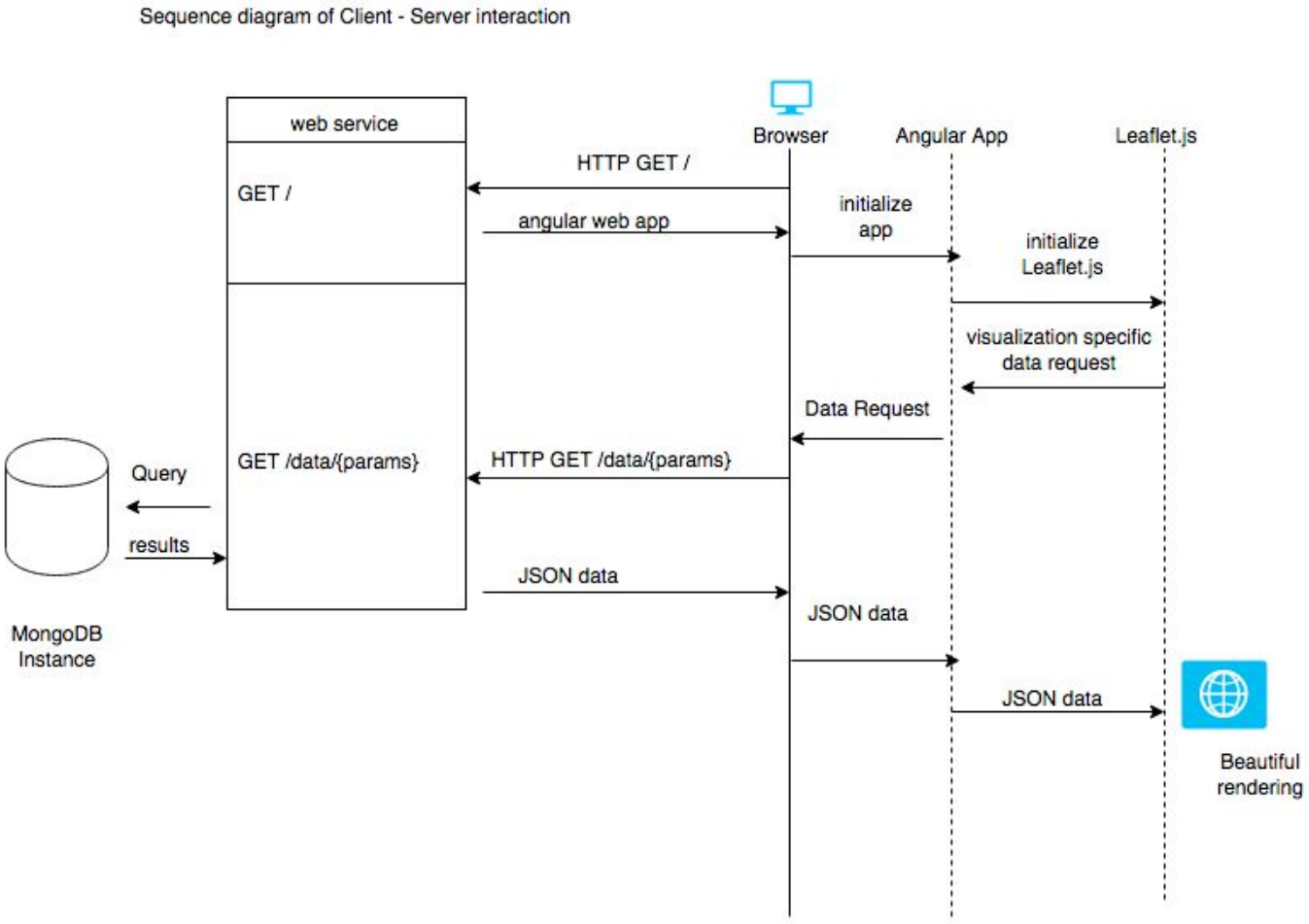
6.1 Ingestion Engine



6.2 Web Server Diagram



6.3 Client Server Interaction Diagram



6.4 Description of Components:

6.4.1 Web Application

The web application serves as the interface to the user. The application will be built on the Angular.js MVC framework. When the user makes the initial HTTP request to the server, the entire application is provided in response. As the user interacts with the application, data requests are sent to the server, the responses are used to update the view of the app.

6.4.2 Web Server

The web server is responsible for serving the initial application to the requesting browser. It then becomes a ReST endpoint for data requests sent by the application. It is the interface between the application and the MongoDB instance. It only has read access to the database.

6.4.3 Ingestion Engine

The ingestion engine is run as a background process on the server that is responsible for keeping the database of tweets and sentiment up to date. It is the only component with write access to the database. On startup, the engine registers itself as an endpoint for the Twitter Streaming API with appropriate filters. When a tweet is received, it performs appropriate analysis of the text, strips unnecessary information from the tweet, then stores it in the MongoDB instance.

7. Use Cases

Use case #	1
Use case name	**View default visualization**
Summary	Main use case, the first visualization the user sees
Dependency	web server + ingestion engine running
Actor	User
Precondition	Script runs successfully
Description	The user opens the webpage and sees the default visualization - a map of San Francisco with different neighborhoods clearly outlined and of various colors which represent the happiness experienced by people of that region. A stack of colors with corresponding Sentiment values representing the scale is also present along with an icon to allow the user to indicate the choice of graph that he or she would like to view.
Alternative	
Postcondition	User can view the graphical representation of discrete data

Use case #	2
Use case name	**Line graph representation of Sentiment flow over Time**
Summary	User will be able to view the the range of Sentiments experienced or expressed by the people of a region over a certain period of Time
Dependency	The map has been loaded
Actor	User
Precondition	The user has to select a region or location on the SF map
Description	When the user loads the page, a colorful map with clearly defined outlines loads up. Each sub-plot of the Map is colored. The user will be able to view over the map an image of a line graph which represents the Sentiments or the value associated with it w.r.t a certain time frame
Alternative	
Postcondition	User can view or capture the Sentiment flow from a specific neighborhood over time

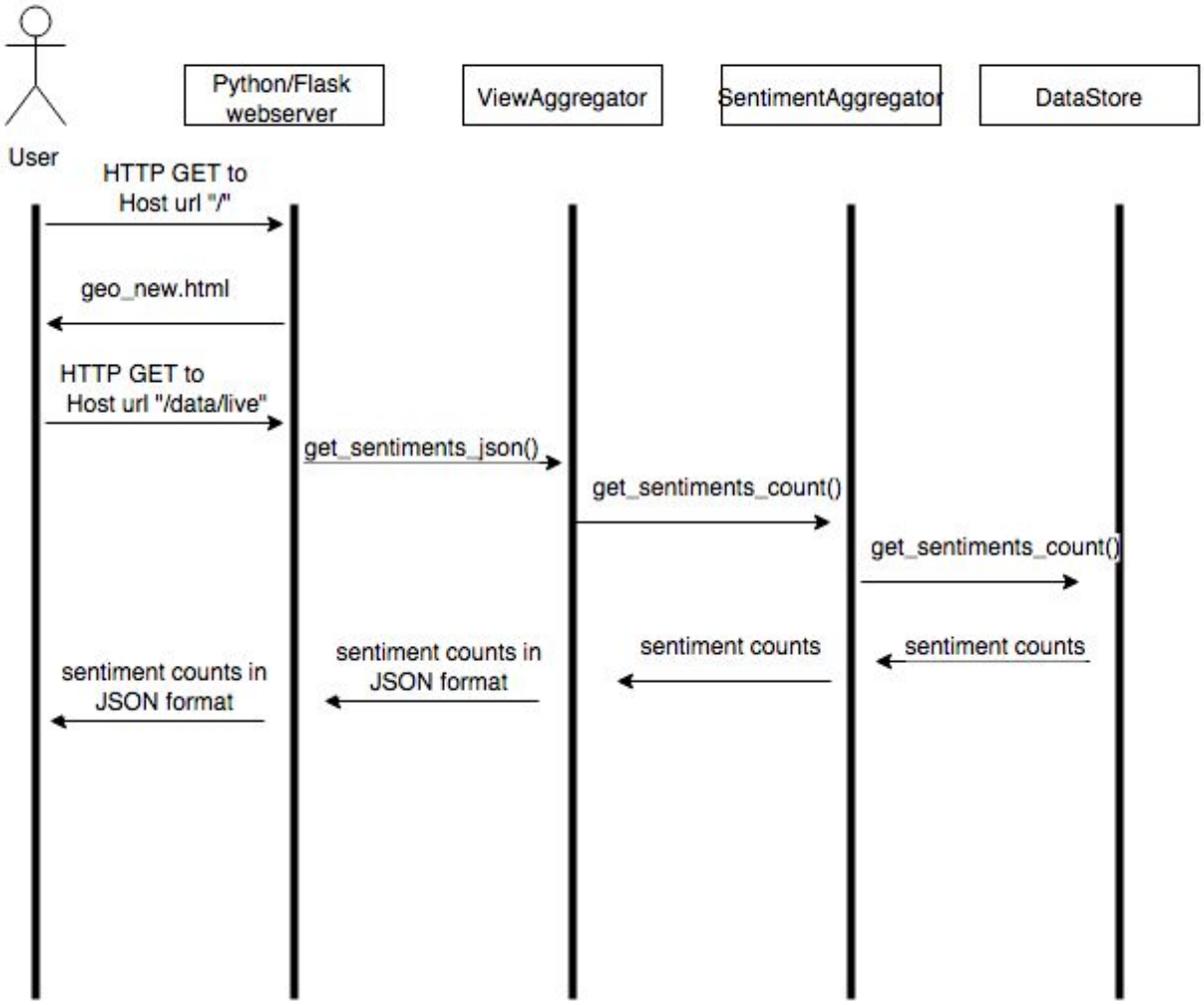
Use case #	3
Use case name	**Bar chart representation of Sentiment value w.r.t. region (neighborhood)**
Summary	User will be able to view the the type of Sentiments experienced or expressed by the people of the different neighborhoods
Dependency	The map has been loaded
Actor	User
Precondition	The user has to click on the map and then the bar chart button

Description	When the user loads the page, a colorful map with clearly defined outlines loads up. Each sub-plot of the Map is colored. The user will be able to view over the map an image of the bars indicating the level of happiness across the different neighborhoods
Alternative	
Postcondition	User can view and capture the level of happiness across the different neighborhoods

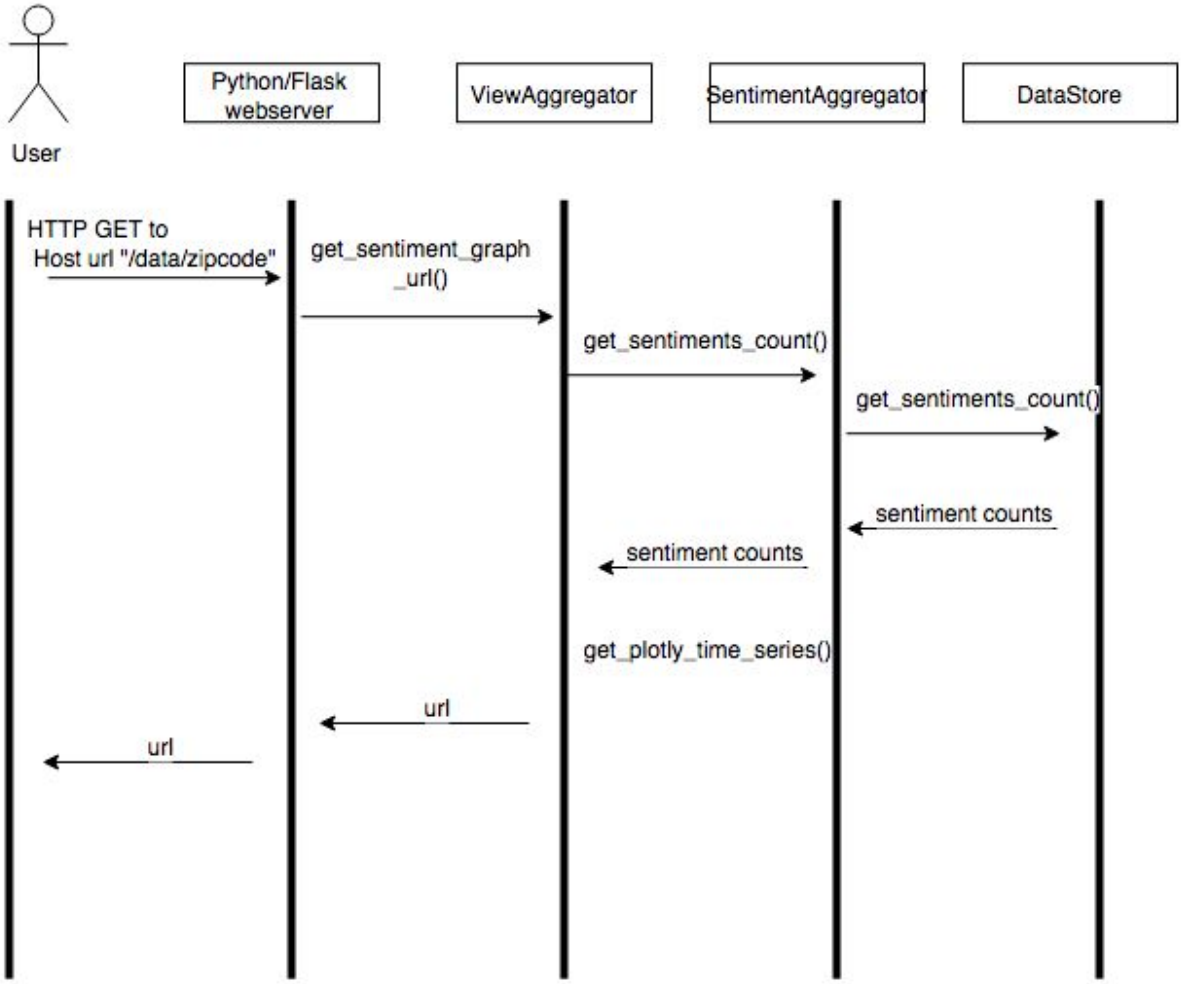
Use case #	4
Use case name	**Export chart**
Summary	The user can export the chart and save it on his computer
Dependency	The chart has been chosen
Actor	User
Precondition	Graph (Line or Bar) has been loaded and displayed
Description	The user click the “play with data” button in the graph window, and user is redirected to plotly website with graph manipulation options
Alternative	
Postcondition	User has a copy of chart

8. Sequence Diagrams

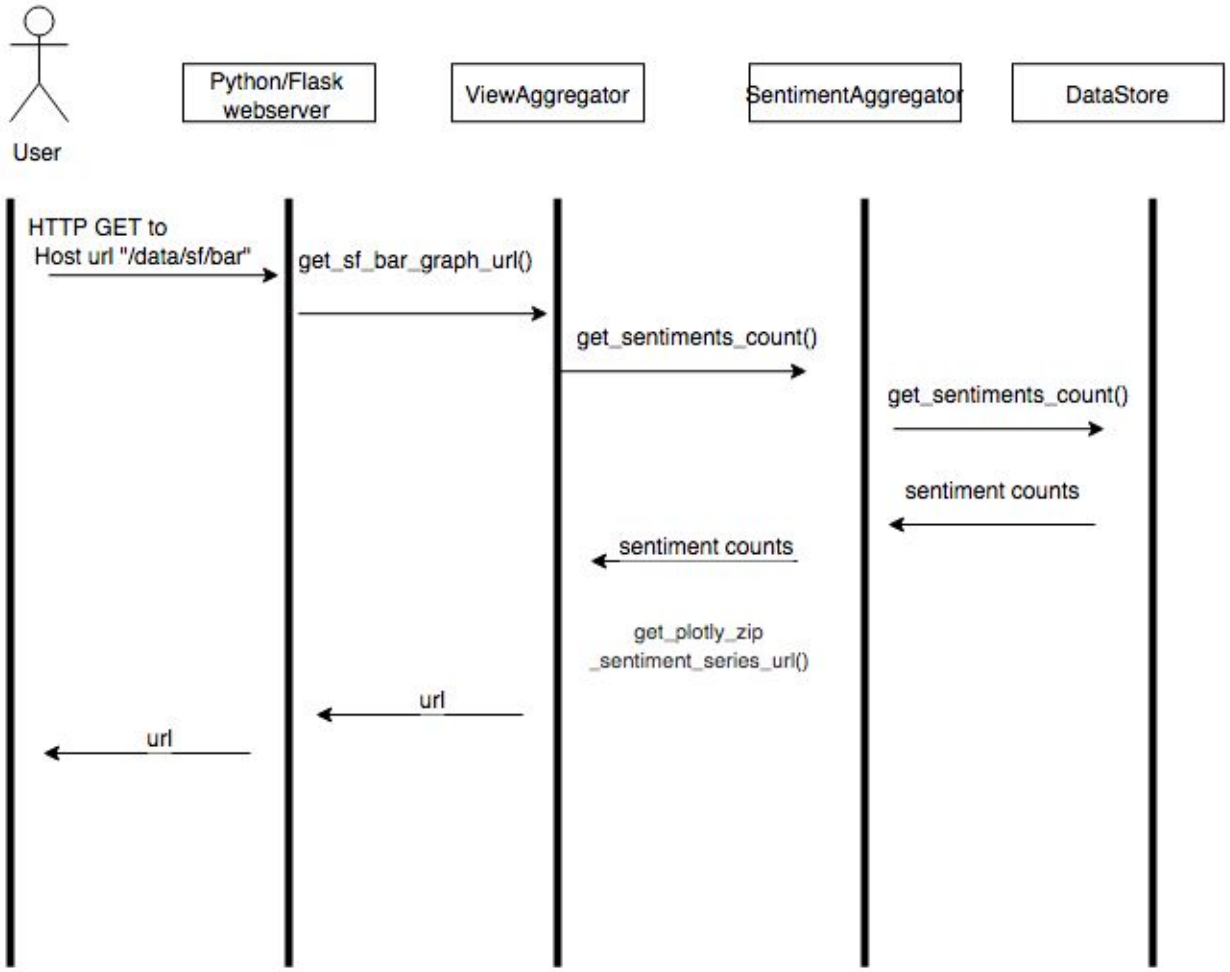
Use Case 1


User
Python/Flask webserver
ViewAggregator
SentimentAggregator
DataStore
HTTP GET to Host url "/"
geo_new.html
HTTP GET to Host url "/data/live"
get_sentiments_json()
get_sentiments_count()
get_sentiments_count()
sentiment counts
sentiment counts
sentiment counts in JSON format
sentiment counts in JSON format

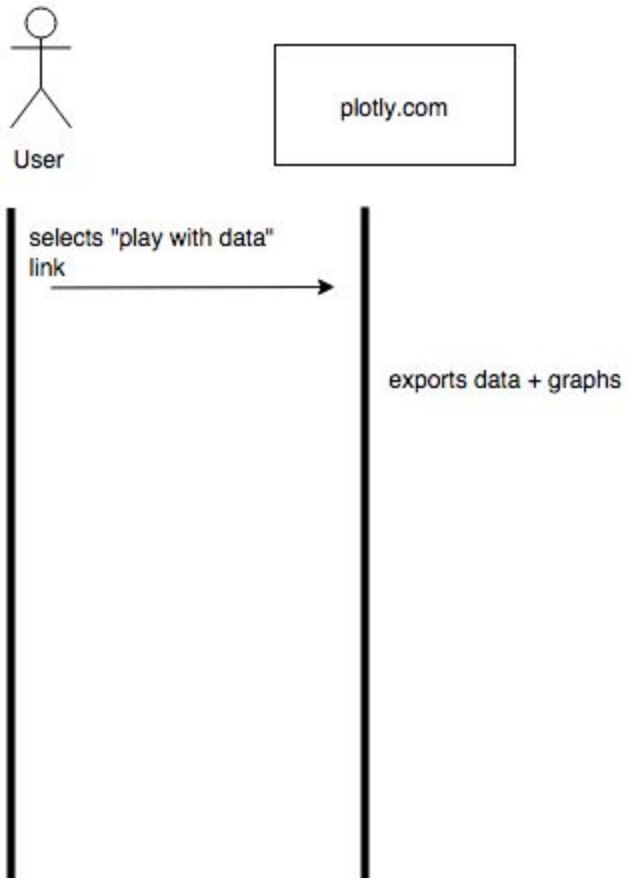
Use Case 2


User
Python/Flask webserver
ViewAggregator
SentimentAggregator
DataStore
HTTP GET to Host url "/data/zipcode"
get_sentiment_graph _url()
get_sentiments_count()
get_sentiments_count()
sentiment counts
sentiment counts
get_plotly_time_series()
url
url

Use Case 3


User
Python/Flask webserver
ViewAggregator
SentimentAggregator
DataStore
HTTP GET to Host url "/data/sf/bar"
get_sf_bar_graph_url()
get_sentiments_count()
get_sentiments_count()
sentiment counts
sentiment counts
get_plotly_zip _sentiment_series_url()
url
url

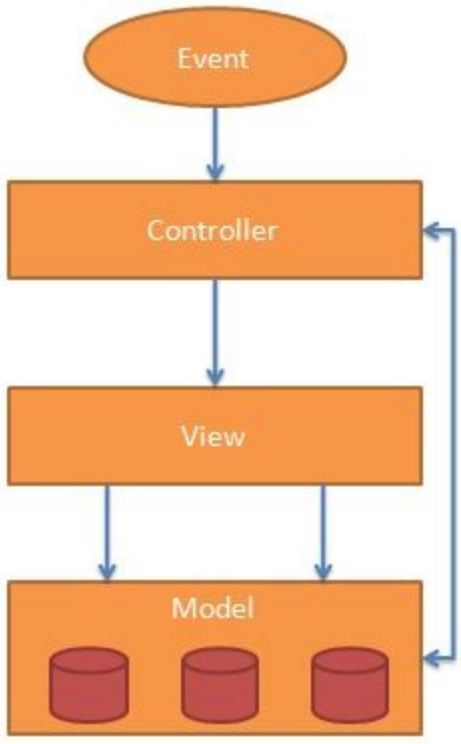
Use Case 4


User
plotly.com
selects "play with data" link
exports data + graphs

9. Design Overview

A) Design Patterns Used

Model-View-Controller



It is a design pattern embedded in Angular.js's nature. Using it is considered a good practice, and heavily recommended. It is generally very popular because it allows for a clear separation between the application logic and the user interface. The main three components of the MVC design pattern are:

- Model: it is the component responsible for the gathering and processing of the data
- View: it is the component responsible for displaying all of the elements on the screen
- Controller: the component responsible for all communication between Model and View

In our project, MVC is a clear choice, as the main object is a map, which can be displayed using the View component, while the Model component can take care of fetching the data from the server, with the Controller taking care of the communication between them.

Dependency Injection

Dependency Injection is a software design pattern that deals with how components get hold of their dependencies. The Angular injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

Dependency Injection in our application

In our application, the component can have the dependency passed to it whenever needed. For example, consider the Angular module MainLayout that displays the main page:

```
var mainLayout = angular.module('MainLayout', []);
```

The MainLayout module will load a function to render a map on to the page:

```
mainLayout.factory('renderMap', function($window) {
	//...
	return {
		$window.resolve();
	};
});
```

If we want to inject this function in another page, we create a new injector that can provide components defined in the MainLayout module and request our function renderMap from the injector.

```
var injector = angular.injector(['MainLayout'', 'newPage']);
var renderMap = injector.get('renderMap');
```

To hand the responsibility of creating components over to the injector, we use a declarative notation in our HTML templates. For example:

```
<div ng-controller="MyController">
  <button ng-click="sayHello()">Hello</button>
</div>
```

```
function MyController($scope, greeter) {
  $scope.sayHello = function() {
    greeter.greet('Hello World');
  };
}
```

Source: angularjs.org documentation

10. External documentation on Algorithms, as appropriate

10.1 Sentiment Analysis Algorithm

This system is built around a central algorithm for determining the polarity of sentiment expressed in a string of text. The algorithm is an implementation of a simple yet powerful machine learning classification technique known as Naive Bayes Classification. This technique is used in many domains outside of text processing. At a high level, when this technique is applied to sentiment analysis of text, the algorithm assigns probabilities to each word that it encounters in the training phase. Each word has a probability that it is found in text with a positive sentiment and a probability that it is found in text with a negative sentiment. These probabilities conform to stochastic constraints, thus the sum of all probabilities for each word is

equal to 1. (The algorithm in general is not limited to only two classes). When a new string of text is encountered, the algorithm uses Bayes Theorem to compute the posterior probability that the set of words came from a positive text and the posterior probability that the text came from a negative text. The sentiment with the highest probability is chosen as the classification for the text.

The algorithm can be generalized as solving this maximization function:

$$\hat{y} = \underset{k \in \{1,\dots,K\}}{\operatorname{argmax}} \; p(C_k) \prod_{i=1}^{n} p(x_i|C_k).$$

Where C_k is a class (in our case, k=2). x_i is a word in the new text, n is the number of words in the text. $p(x_i|C_k)$ is the probability that the word occurs in a text with class C_k.

To implement this algorithm we used the the python library called the Natural Language Toolkit (NLTK). Implementation is carried out in two steps. In the first step we need to determine the class probabilities associated with each word. To do so, we query a database for tweets that contain the emoticons ":)" and ":(". They are labeled accordingly.

```
        # read samples from DB
        posTweets = self.db.get_N_results("\\:\\)",num_samples)
        negTweets = self.db.get_N_results("\\:\\(",num_samples)

        # read query results into memory
        posTweets = [(t["text"].split(" "),'positive') for t in posTweets]
        negTweets = [(t["text"].split(" "),'negative') for t in negTweets]

        labeled_tweets = posTweets + negTweets

        random.shuffle(labeled_tweets)
```

This corpus of training data is split into two groups, train_set and test_set. train_set will be used to compute the probabilities and test_set used to verify the results. The train_set is then passed to the NLTK classifier to compute the probability.

```
        # apply_features is a lazy loader, so that features are
        # computed as necessary, instead of being loaded into memory
        # all at once
        train_set = apply_features(self.document_features,
                                   labeled_tweets[:len(labeled_tweets) - test_size])
        test_set = apply_features(self.document_features,
                                  labeled_tweets[len(labeled_tweets) - test_size:])

        self.classifier = nltk.NaiveBayesClassifier.train(train_set)
```

The ‘apply_features()’ function as well as the ‘self.document_features’ function are needed to transform the data set into the proper representation for NLTK. We will not dwell on those details here.

With the trained classifier, we can now pass a string to the ‘classify()’ function which will return the class with the highest probability computed from the text.

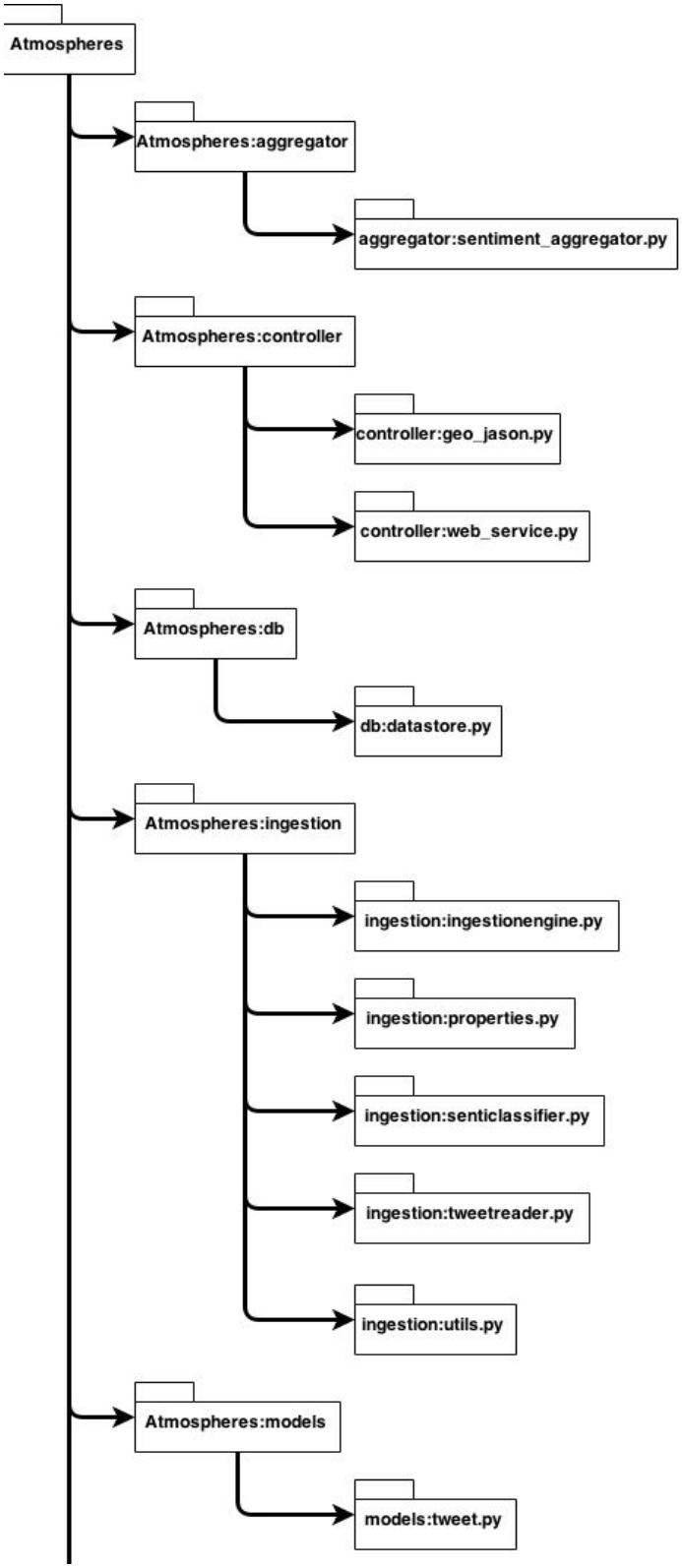
```
def classify(self,text):
        """runs the classifier on the input text """
        document = text.split(" ")
        features = self.document_features(document)
        return self.classifier.classify(features)
```

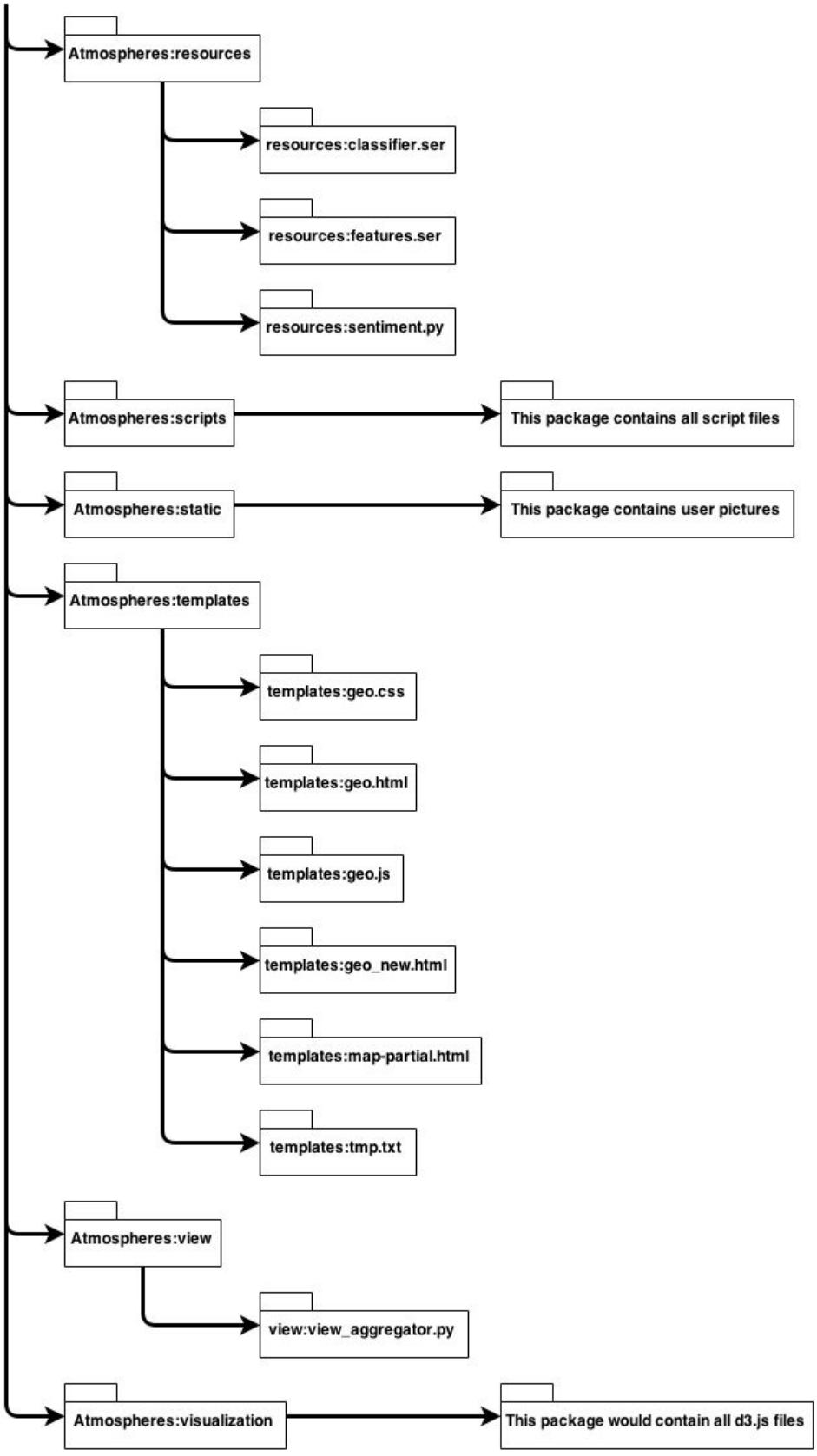
10.2 Zipcode Sentiment Computation

Tweet model stores sentiment. This sentiment is calculated by Natural Language library after analysing the content of the tweet. Tweet model also stores the date of creation. This helps our query to find the tweets which were posted during particular interval of time. Tweets are aggregated based on two parameters, types of sentiment (positive sentiments and negative sentiments) and zipcode. Then following algorithm is used to calculate the percentage of the sentiment for a particular zip code.

```
def get_sentiment_count(self, sentiment_type, zipcode, start_time, end_time):

   condition = {
                  'sentiment' : sentiment_type,
                  'created_at' : {'$gt': start_time, '$lt' : end_time},
            }

   if(zipcode):
       condition["zipcode"] = zipcode

   sentiment_count = self.collection.find(condition).count()

   return sentiment_count

# Algorithm to calculate the percentage of sentiment
positive_count =  aggregator.get_sentiment_count(SentimentType.positive,
                                                 zipcode,
                                                 datetimes[i],
                                                 datetimes[i+1])
negative_count =  aggregator.get_sentiment_count(SentimentType.negative,
                                                 zipcode,
                                                 datetimes[i],
                                                 datetimes[i+1])
if positive_count == 0 and negative_count == 0:
    sentiment = 0
else:
    sentiment = float(positive_count - negative_count) / float(positive_count +
negative_count)
```

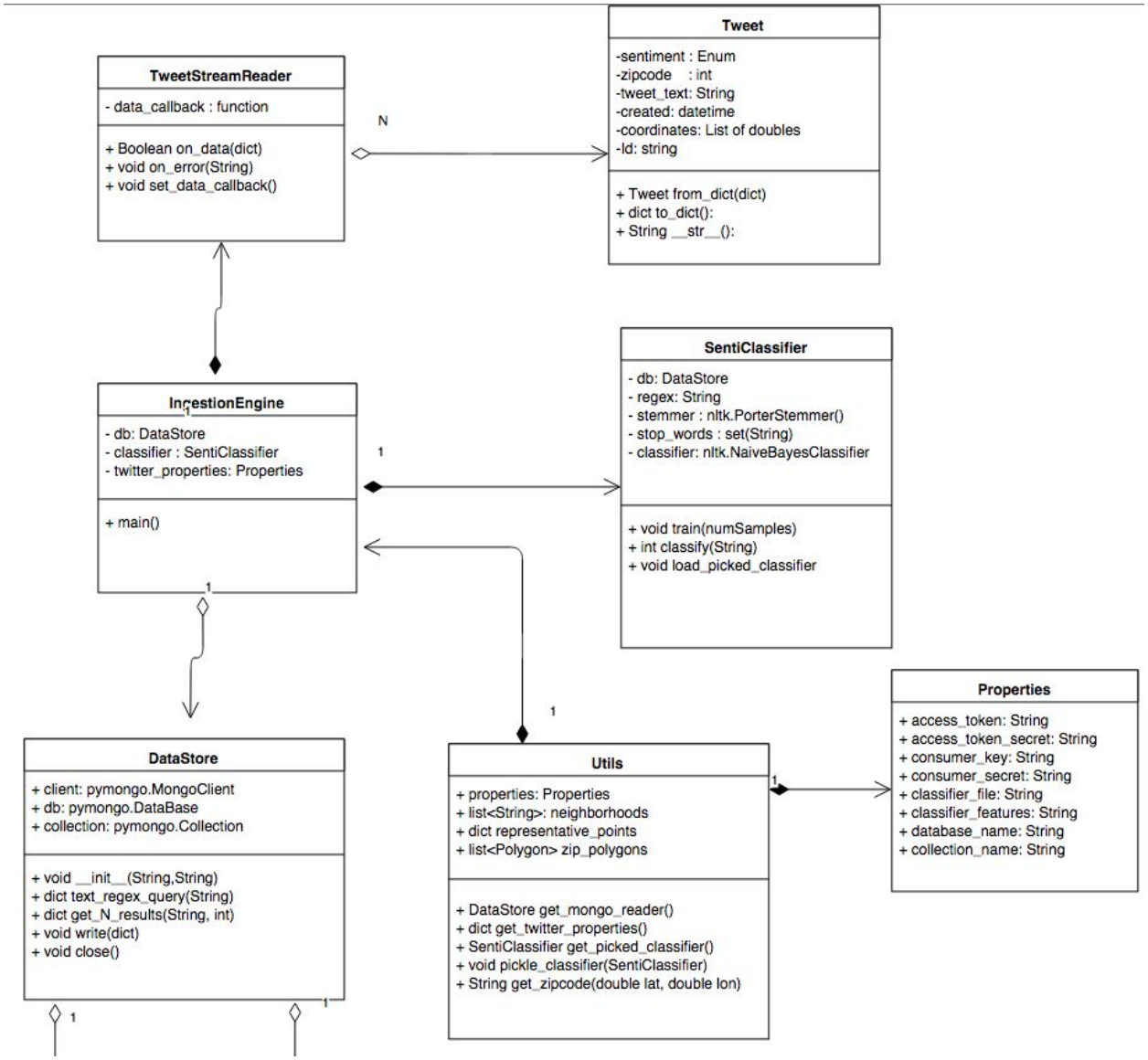
11. Package Diagram



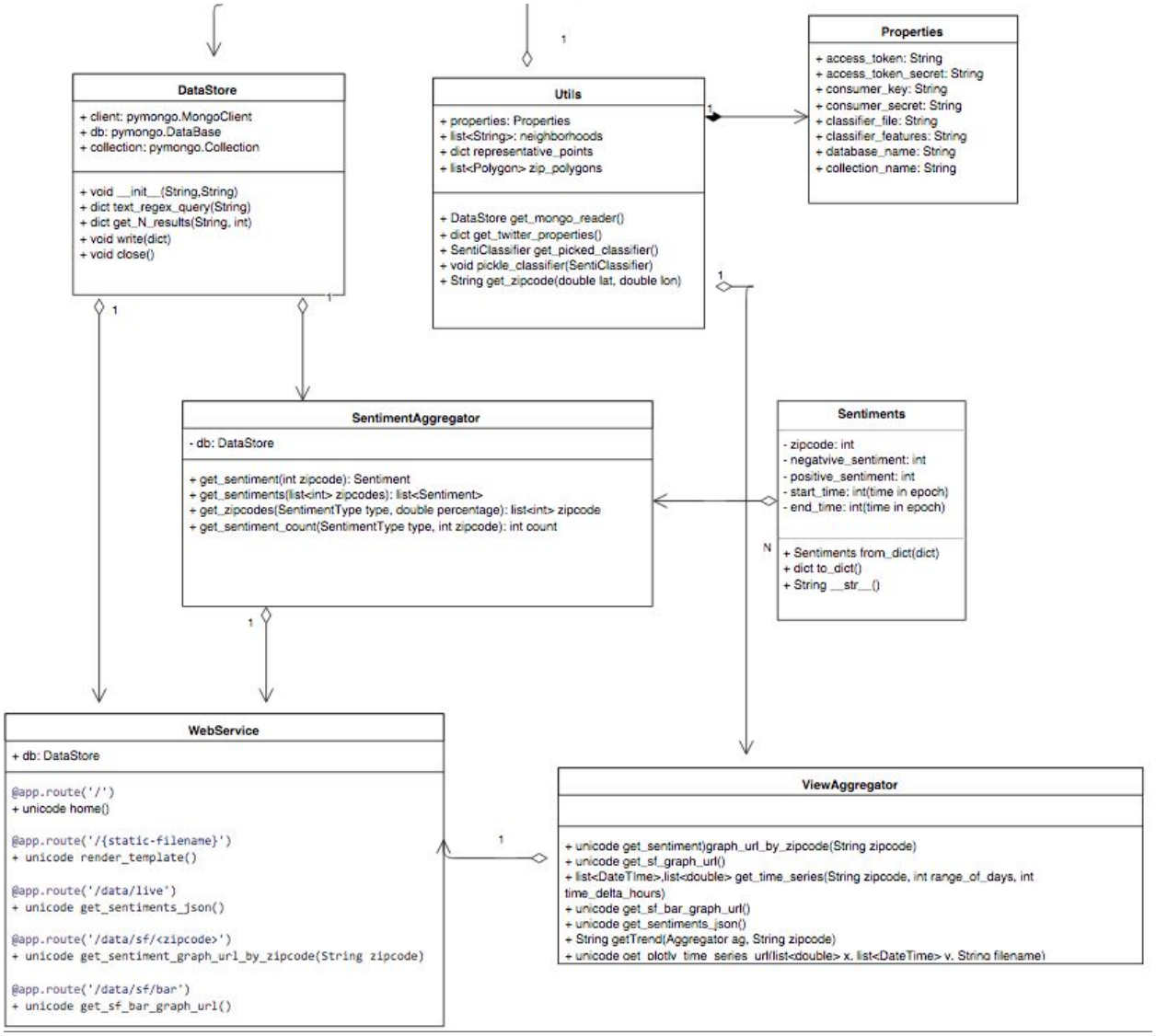

Atmospheres:resources
resources:classifier.ser
resources:features.ser
resources:sentiment.py
Atmospheres:scripts
This package contains all script files
Atmospheres:static
This package contains user pictures
Atmospheres:templates
templates:geo.css
templates:geo.html
templates:geo.js
templates:geo_new.html
templates:map-partial.html
templates:tmp.txt
Atmospheres:view
view:view_aggregator.py
Atmospheres:visualization
This package would contain all d3.js files

12. Class Diagrams

12.1 Data Ingestion



12.2 Web Server



12.3 Classes and their Roles

Package atmospheres.controller

WebService Class

This is a module which acts as controller for the RESTful web service. It has mapping of urls and the corresponding methods. These methods get invoked when request comes for the specific urls. This module have access to the SentimentAggregator to fetch the aggregated sentiments data and return it to the client.

Package atmospheres.db

DataStore Class

This class is the main interface between the MongoDB server and the rest of the system.

It manages the database connection. It opens a database connection and connection to the desired collection.

It exposes functions for querying and writing to the database. Deleting from the database is not supported.

Package atmospheres.ingestion

IngestionEngine Class

This is the entry point into the tweet reading and sentiment analysis service.

Initializes the TweetStreamReader class with proper credentials.

Sets the TweetStreamReader filter to only collect tweets from within a given geometry

Registers a callback function with the TweetStreamReader that gets called when a tweet is received

In the callback function, the SentiClassifier object is used to analyze the sentiment of the tweet, salient information is then written to the DB using the DataStore object.

SentiClassifier Class

This class is responsible for performing the data mining task of sentiment classification.

Training a classifier can be time consuming, so the SentiClassifier loads a pre trained Naive Bayes classifier.

It has the ability to train a new classifier that can be serialized and used as the default.

It exposes the function classify() that takes a String argument and returns 1 or -1 depending on outcome of the sentiment classification.

TweetReader Class

This is the main interface to the Twitter Streaming API.
It registers itsself with Twitter as an endpoint to receive tweets and then waits to receive tweets.
When a tweet is received, it performs the callback function that was specified by the IngestionEngine.

Properties Class

Holds properties information for the ingestion engine, such as twitter access tokens,
database/collection names and file locations.

Utils Class

A utility class that uses the data stored in Properties to perform useful actions.
It instantiated a DataStore object per the specs in Properties
It provides a dictionary of Twitter properties.
It has methods to serialize and deserialize the SentiClassifier object.

Package atmospheres.models

Tweet Class

This class is a model class which contains the tweets related information. It is also used by DataStore to save the tweets in the persistent data store for later processing.

Package atmospheres.resources

Sentiment Class

This is a resource class and it is exposed to the client. This class will be used by the SentimentAggregator to return result to the client after aggregating the data from Persistent data store. It has following methods
from_dict(): This method creates sentiment data from a given sentiment dictionary.
to_dict(): This method creates sentiment dictionary from a given sentiment object.

SentimentAggregator Class

It exposes various methods to return aggregated data from datastore. It has DataStore instance to access Tweet data. It exposes different methods as shown below:

get_sentiment(int zipcode): It returns the aggregated sentiment information for a particular zipcode.

get_sentiments(list<int> zipcodes): This method returns the aggregated list of sentiment for provided zipcodes.

get_zipcodes(SentimentType type, double percentage): This method return the list of zipcode whose sentiment is higher than the given percentage for particular sentiment(positive, negative).

get_sentiment_count(SentimentType type, int zipcode): This method returns the number of sentiment count for particular type of sentiment for a specific zipcode.

Package atmospheres.view

ViewAggregator Class

This class works with the SentimentAggregator class to extract and compute the sentiment information from the database. It transforms the data into the necessary data structures to leverage the graph rendering libraries. It interfaces with the external libraries to generate the graph, then provides the URL to that graph to the server.

13. Milestones

868-668 Project Proposal

I. Brief Background

The micro blog platform Twitter generates millions of tweets per day. Methods for extracting trends and insight from this data has been an active area of research because twitter provides unprecedented insight into individual's real-time opinions. Previous research [1] proposed a method for measuring geographically specific sentiment using machine learning techniques.

This system used a static historical data set collected over a fixed period of several weeks. We propose a project to build on this research and develop a real-time web-based tool to analyze local sentiment of the different neighborhoods of San Francisco.

This kind of tool could be of use for local governments and civil entities tasked with understanding and responding to the needs of the community.

II. Brief overview of system

The system can be broken down into four major components:

1. Data Collection and Analysis Engine.
This will be responsible for reading from the live twitter stream and performing necessary analysis and data transformation. It will be responsible for populating the database.

2. Data Server
This will be a server built using Java ReSTful Services responsible for serving the initial web-app as well as providing access to the DB through ReST endpoints.

3. Web Application
This application will use an existing web framework such as Angular.js, Backbone.js, or React.js to structure the front end.

4. Data Visualization

Will be responsible for rendering pleasing visualizations of data such as heat maps of local sentiment of San Francisco divided by neighborhood. This will likely use D3.js or Processing.js

III. System Functionality

1. User navigates to webpage
2. User views high level visualization of all neighborhoods
3. User clicks on individual neighborhoods to see more detailed information
4. User modifies filter options to control what data is rendered to the screen.

IV. Expected Size of Code

5,000 (?) lines of Python, HTML/CSS, and JavaScript

V. References

1. Eshleman, Ryan, and Hui Yang. "" Hey# 311, Come Clean My Street!": A Spatio-temporal Sentiment Analysis of Twitter Data and 311 Civil Complaints."*Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on*. IEEE, 2014.

13.1 Milestone 1

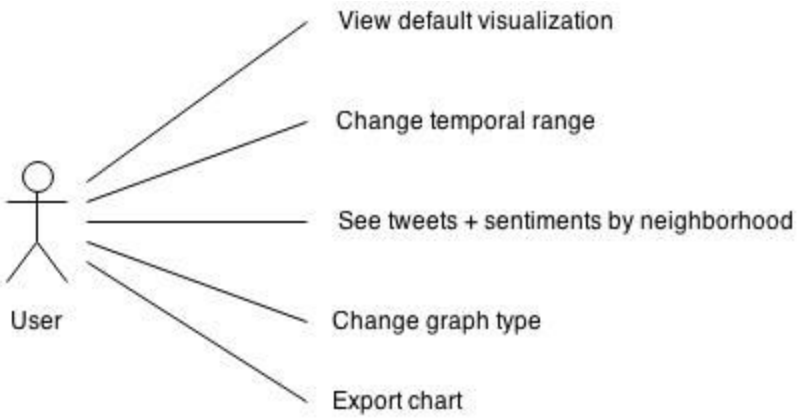
High Level Specs/Analysis

General functionality:

This web application will serve as a visualization tool that integrates several aspects of the data mining process into a streamlined user interface to support important task in data exploration and analysis. The application has the ability to:

1) Filter and collect tweets from a live stream.
2) Perform the task of sentiment analysis on the text contained within the tweet.
3) Store and retrieve results for future analysis
4) Visualize different aspects of the processed data, including spatial and temporal

Use Cases:



Use Case Description:

Use case #	**1**
Use case name	**View default visualization**
Summary	Main use case, the first visualization the user sees
Dependency	web server + ingestion engine running
Actor	User
Precondition	
Description	The user opens the webpage and sees the default visualization
Alternative	
Postcondition	

Use case #	**2**
Use case name	**Change temporal range**
Summary	The user can change the temporal range for the visualization
Dependency	
Actor	User
Precondition	user on web page
Description	The user can change the temporal range, which will update the visualization to the the selected temporal range
Alternative	
Postcondition	visualization updates according to new range

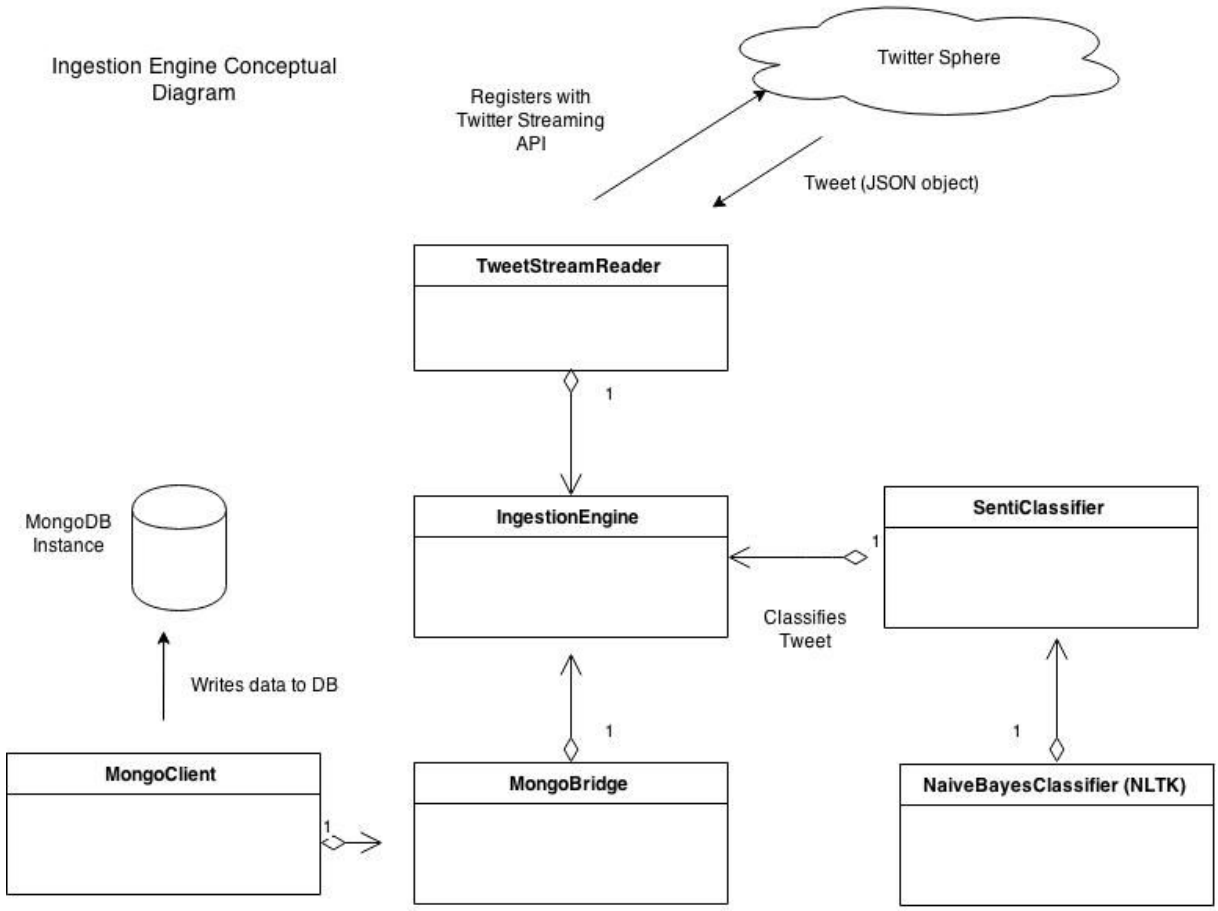
Use case #	**3**
Use case name	**See tweets + sentiments by neighborhood**
Summary	View tweets + sentiments from a specific neighborhood
Dependency	
Actor	User
Precondition	Click on a neighborhood to select it
Description	The user selects a neighborhood by clicking on the corresponding area, and can see the tweets and sentiments about that neighborhood
Alternative	
Postcondition	additional UI element added to display

Use case #	**4**
Use case name	**Change graph type**
Summary	The user can change the graph type
Dependency	
Actor	User
Precondition	
Description	The user clicks on the “Change graph type” button, which changes the visualization
Alternative	
Postcondition	type of graph displayed is changed

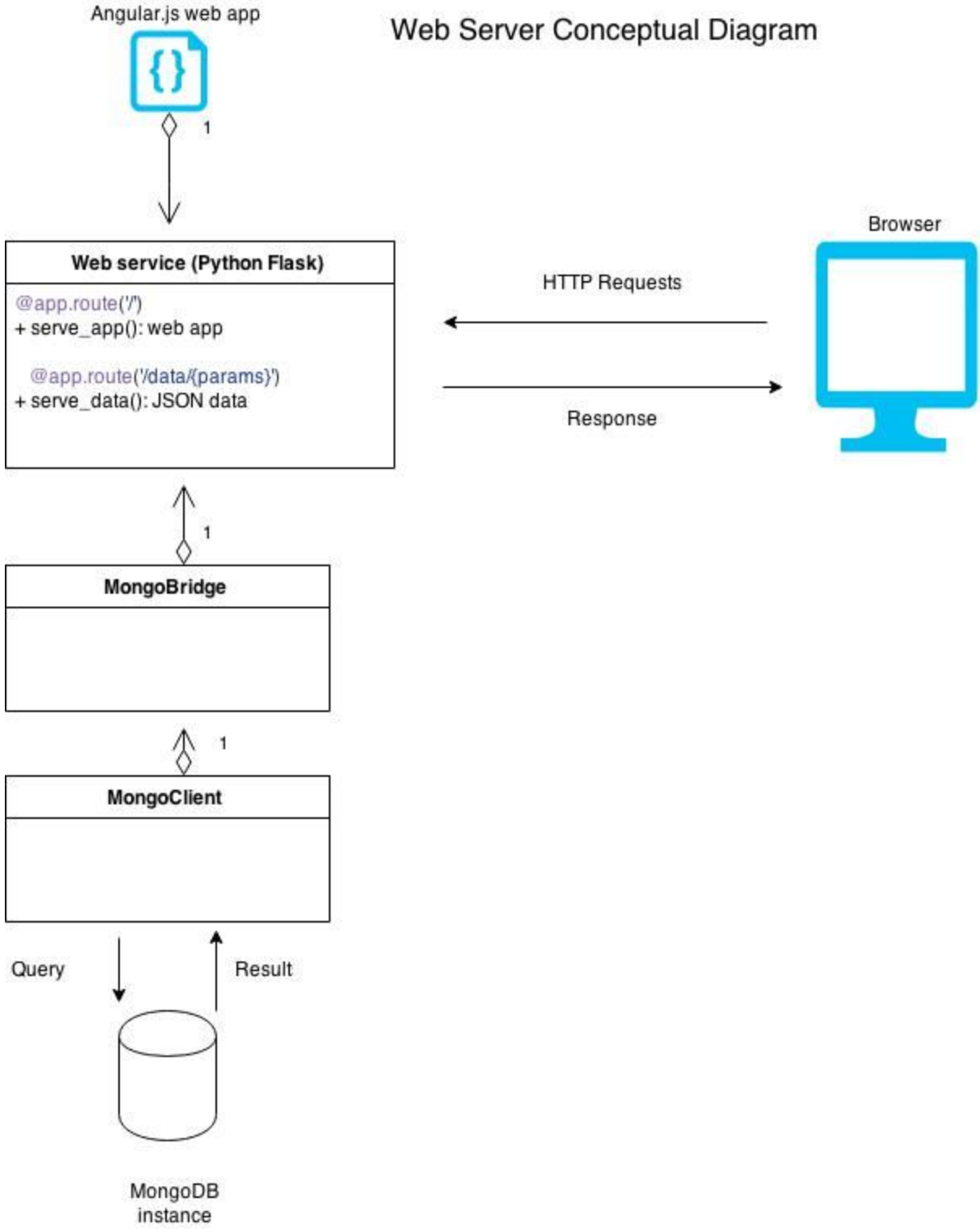
Use case #	**5**
Use case name	**Export chart**
Summary	The user can export the chart and save it on his computer
Dependency	chart has been chosen
Actor	User
Precondition	
Description	The user click the “Export chart” button, and a download starts containing the active map visualization
Alternative	
Postcondition	use has copy of chart

Conceptual Models

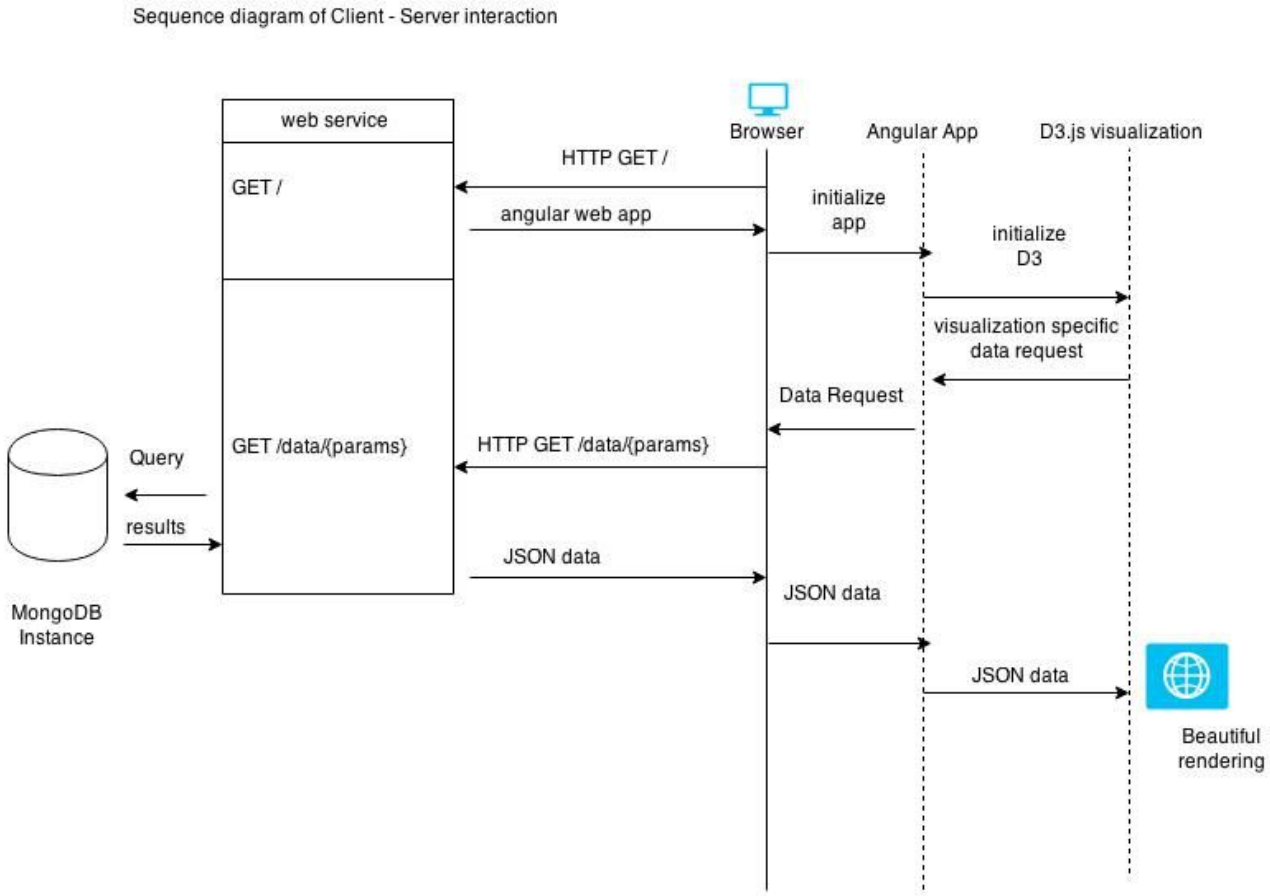
Ingestion Engine:



Web Server Diagram:



Client Server Interaction Diagram:



Description of Components:

Web Application

The web application serves as the interface to the user. The application will be built on the Angular.js MVC framework. When the user makes the initial HTTP request to the server, the entire application is provided in response. As the user interacts with the application, data requests are sent to the server, the responses are used to update the view of the app.

Web Server

The web server is responsible for serving the initial application to the requesting browser. It then becomes a ReST endpoint for data requests sent by the application. It is the interface between the application and the MongoDB instance. It only has read access to the database.

Ingestion Engine

The ingestion engine is run as a background process on the server that is responsible for keeping the database of tweets and sentiment up to date. It is the only component with write

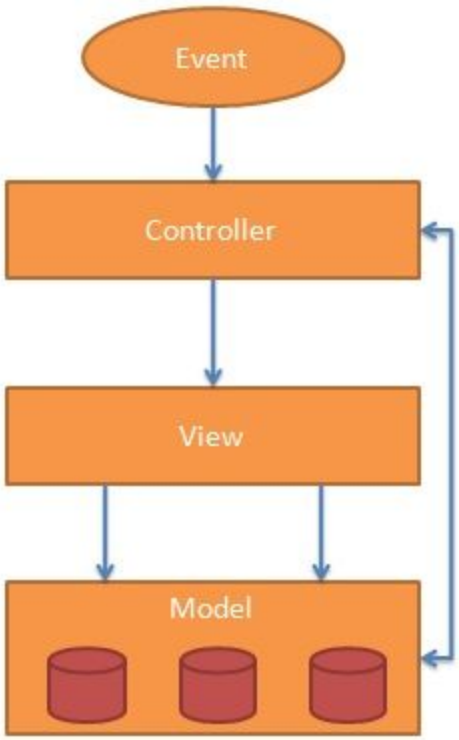
access to the database. On startup, the engine registers itself as an endpoint for the Twitter Streaming API with appropriate filters. When a tweet is received, it performs appropriate analysis of the text, strips unnecessary information from the tweet, then stores it in the MongoDB instance.

13.2 Milestone 2

Design

A) Design Patterns Used

Model-View-Controller



It is a design pattern embedded in Angular.js's nature. Using it is considered a good practice, and heavily recommended. It is generally very popular because it allows for a clear separation between the application logic and the user interface. The main three components of the MVC design pattern are:

- Model: it is the component responsible for the gathering and processing of the data
- View: it is the component responsible for displaying all of the elements on the screen
- Controller: the component responsible for all communication between Model and View

In our project, MVC is a clear choice, as the main object is a map, which can be displayed using the View component, while the Model component can take care of fetching the data from the server, with the Controller taking care of the communication between them.

Dependency Injection

Dependency Injection is a software design pattern that deals with how components get hold of their dependencies. The Angular injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

Dependency Injection in our application

In our application, the component can have the dependency passed to it whenever needed. For example, consider the Angular module MainLayout that displays the main page:

```
var mainLayout = angular.module('MainLayout', []);
```

The MainLayout module will load a function to render a map on to the page:

```
mainLayout.factory('renderMap', function($window) {
	//...
	return {
		$window.resolve();
	};
});
```

If we want to inject this function in another page, we create a new injector that can provide components defined in the MainLayout module and request our function renderMap from the injector.

```
var injector = angular.injector(['MainLayout'', 'newPage']);
var renderMap = injector.get('renderMap');
```

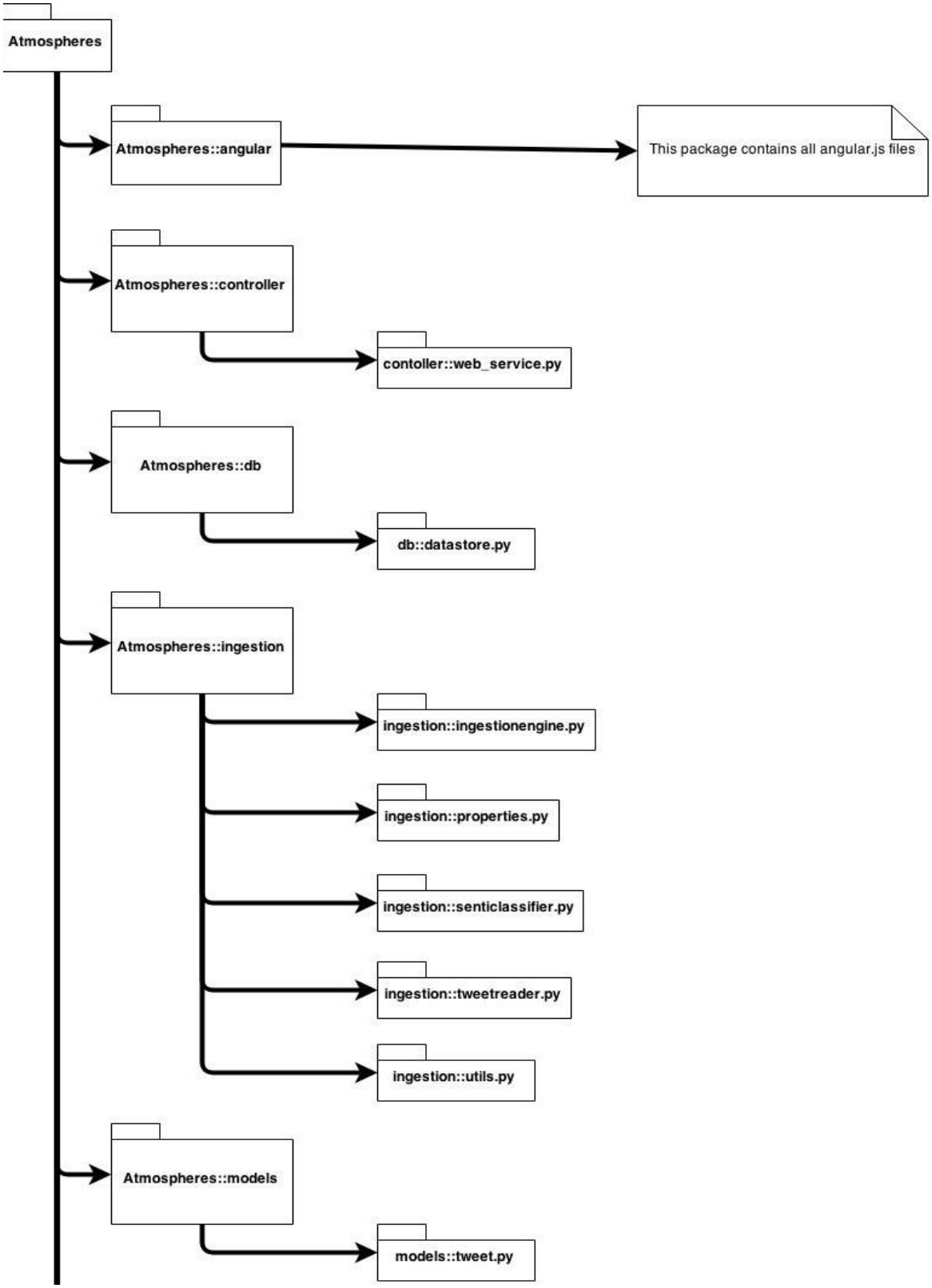
To hand the responsibility of creating components over to the injector, we use a declarative notation in our HTML templates. For example:

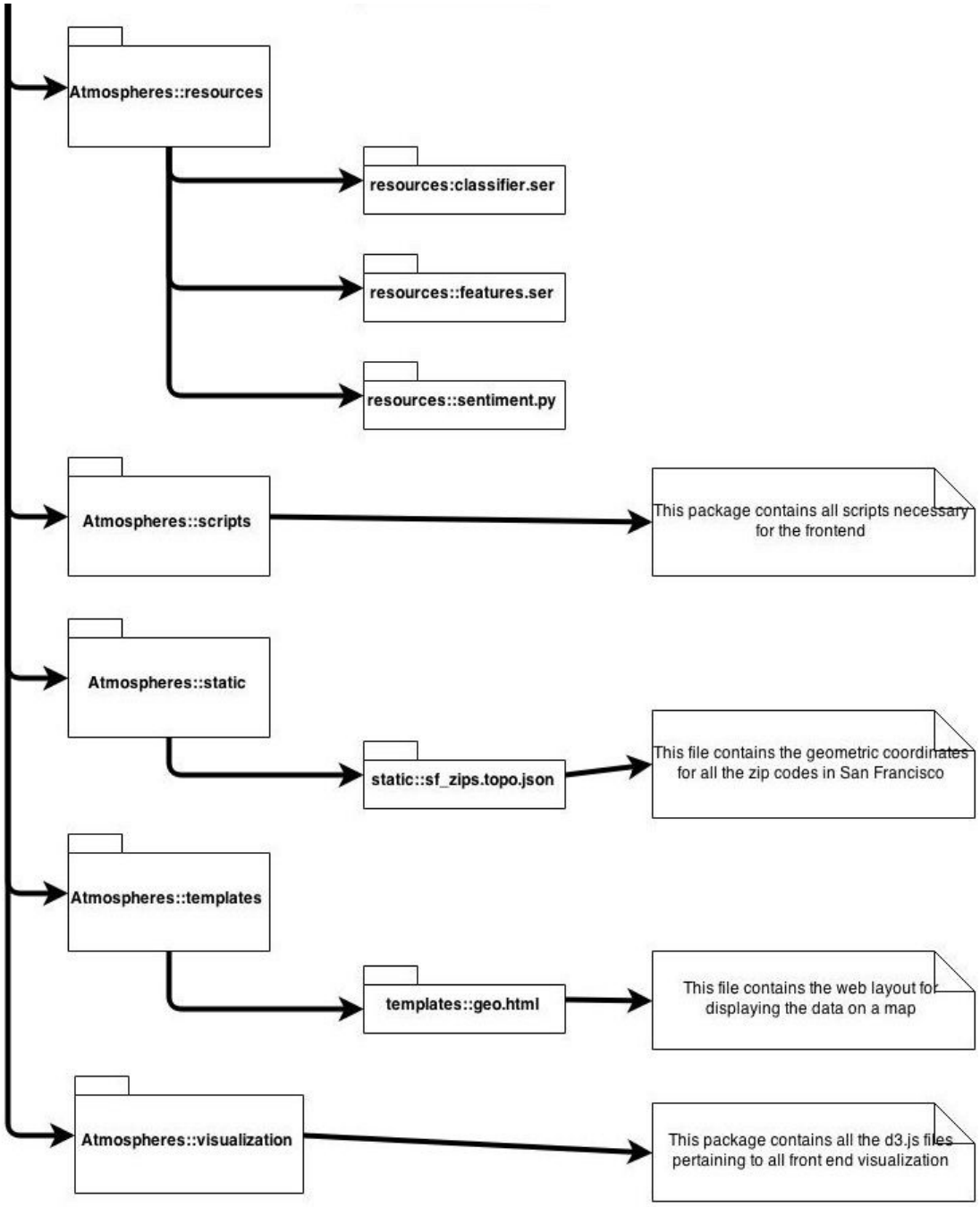
```
<div ng-controller="MyController">
  <button ng-click="sayHello()">Hello</button>
</div>
```

```
function MyController($scope, greeter) {
  $scope.sayHello = function() {
    greeter.greet('Hello World');
  };
}
```

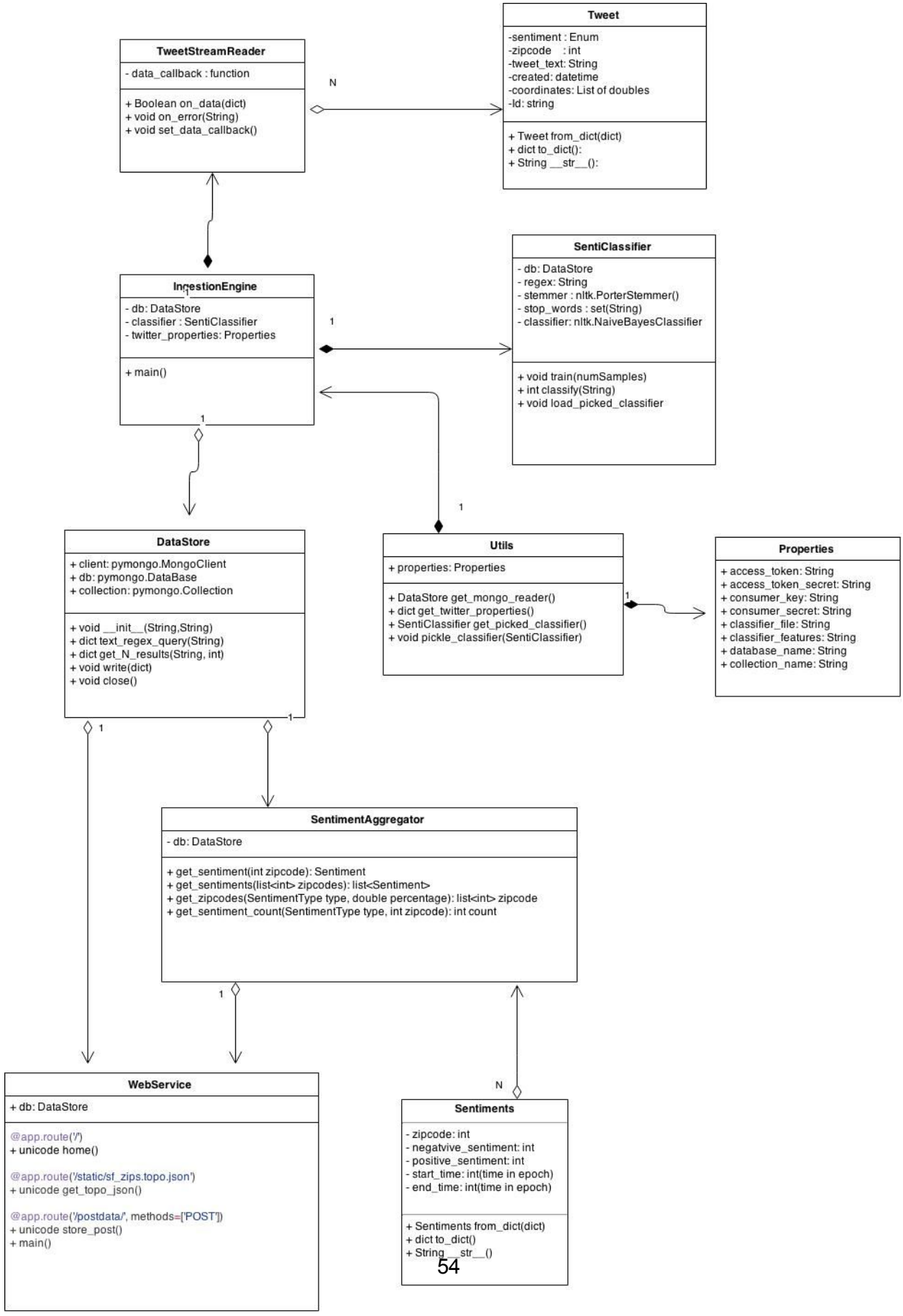
Source: angularjs.org documentation

B) Package Structure


Atmospheres
Atmospheres::angular
This package contains all angular.js files
Atmospheres::controller
contoller::web_service.py
Atmospheres::db
db::datastore.py
Atmospheres::ingestion
ingestion::ingestionengine.py
ingestion::properties.py
ingestion::senticlassifier.py
ingestion::tweetreader.py
ingestion::utils.py
Atmospheres::models
models::tweet.py



C) Class diagrams backend:


TweetStreamReader
- data_callback : function
+ Boolean on_data(dict)
+ void on_error(String)
+ void set_data_callback()
N
Tweet
-sentiment : Enum
-zipcode : int
-tweet_text: String
-created: datetime
-coordinates: List of doubles
-Id: string
+ Tweet from_dict(dict)
+ dict to_dict():
+ String __str__():
IngestionEngine
- db: DataStore
- classifier : SentiClassifier
- twitter_properties: Properties
+ main()
1
1
1
SentiClassifier
- db: DataStore
- regex: String
- stemmer : nltk.PorterStemmer()
- stop_words : set(String)
- classifier: nltk.NaiveBayesClassifier
+ void train(numSamples)
+ int classify(String)
+ void load_picked_classifier
1
DataStore
+ client: pymongo.MongoClient
+ db: pymongo.DataBase
+ collection: pymongo.Collection
+ void __init__(String,String)
+ dict text_regex_query(String)
+ dict get_N_results(String, int)
+ void write(dict)
+ void close()
1
1
Utils
+ properties: Properties
+ DataStore get_mongo_reader()
+ dict get_twitter_properties()
+ SentiClassifier get_picked_classifier()
+ void pickle_classifier(SentiClassifier)
1
Properties
+ access_token: String
+ access_token_secret: String
+ consumer_key: String
+ consumer_secret: String
+ classifier_file: String
+ classifier_features: String
+ database_name: String
+ collection_name: String
SentimentAggregator
- db: DataStore
+ get_sentiment(int zipcode): Sentiment
+ get_sentiments(list<int> zipcodes): list<Sentiment>
+ get_zipcodes(SentimentType type, double percentage): list<int> zipcode
+ get_sentiment_count(SentimentType type, int zipcode): int count
1
N
WebService
+ db: DataStore
@app.route('/')
+ unicode home()
@app.route('/static/sf_zips.topo.json')
+ unicode get_topo_json()
@app.route('/postdata/', methods=['POST'])
+ unicode store_post()
+ main()
Sentiments
- zipcode: int
- negatvive_sentiment: int
- positive_sentiment: int
- start_time: int(time in epoch)
- end_time: int(time in epoch)
+ Sentiments from_dict(dict)
+ dict to_dict()
+ String __str__()

D) Classes and their Roles

Package atmospheres.controller

WebService Class

This is a module which acts as controller for the RESTful web service. It has mapping of urls and the corresponding methods. These methods get invoked when request comes for the specific urls. This module have access to the SentimentAggregator to fetch the aggregated sentiments data and return it to the client.

Package atmospheres.db

DataStore Class

This class is the main interface between the MongoDB server and the rest of the system.

It manages the database connection. It opens a database connection and connection to the

desired collection.

It exposes functions for querying and writing to the database. Deleting from the database

is not supported.

Package atmospheres.ingestion

IngestionEngine Class

This is the entry point into the tweet reading and sentiment analysis service.

Initializes the TweetStreamReader class with proper credentials.

Sets the TweetStreamReader filter to only collect tweets from within a given geometry

Registers a callback function with the TweetStreamReader that gets called when a tweet

is received.

In the callback function, the SentiClassifier object is used to analyze the sentiment of the

tweet, salient information is then written to the DB using the DataStore object.

SentiClassifier Class

This class is responsible for performing the data mining task of sentiment classification.

Training a classifier can be time consuming, so the SentiClassifier loads a pre trained

Naive Bayes classifier.

It has the ability to train a new classifier that can be serialized and used as the default.

It exposes the function classify() that takes a String argument and returns 1 or -1 depending on outcome of the sentiment classification.

TweetReader Class

This is the main interface to the Twitter Streaming API.

It registers itsself with Twitter as an endpoint to receive tweets and then waits to receive tweets.

When a tweet is received, it performs the callback function that was specified by the IngestionEngine.

Properties Class

Holds properties information for the ingestion engine, such as twitter access tokens,

database/collection names and file locations.

Utils Class

A utility class that uses the data stored in Properties to perform useful actions.

It instantiated a DataStore object per the specs in Properties

It provides a dictionary of Twitter properties.

It has methods to serialize and deserialize the SentiClassifier object.

Package atmospheres.models

Tweet Class

This class is a model class which contains the tweets related information. It is also used by DataStore to save the tweets in the persistent data store for later processing.

Package atmospheres.resources

Sentiment Class

This is a resource class and it is exposed to the client. This class will be used by the SentimentAggregator to return result to the client after aggregating the data from Persistent data store. It has following methods

from_dict(): This method creates sentiment data from a given sentiment dictionary.

to_dict(): This method creates sentiment dictionary from a given sentiment object.

SentimentAggregator Class

It exposes various methods to return aggregated data from datastore. It has DataStore instance to access Tweet data. It exposes different methods as shown below:

get_sentiment(int zipcode): It returns the aggregated sentiment information for a particular zipcode.

get_sentiments(list<int> zipcodes): This method returns the aggregated list of sentiment for provided zipcodes.

get_zipcodes(SentimentType type, double percentage): This method return the list of zipcode whose sentiment is higher than the given percentage for particular sentiment(positive, negative).

get_sentiment_count(SentimentType type, int zipcode): This method returns the number of sentiment count for particular type of sentiment for a specific zipcode.

13.3 Milestone 3

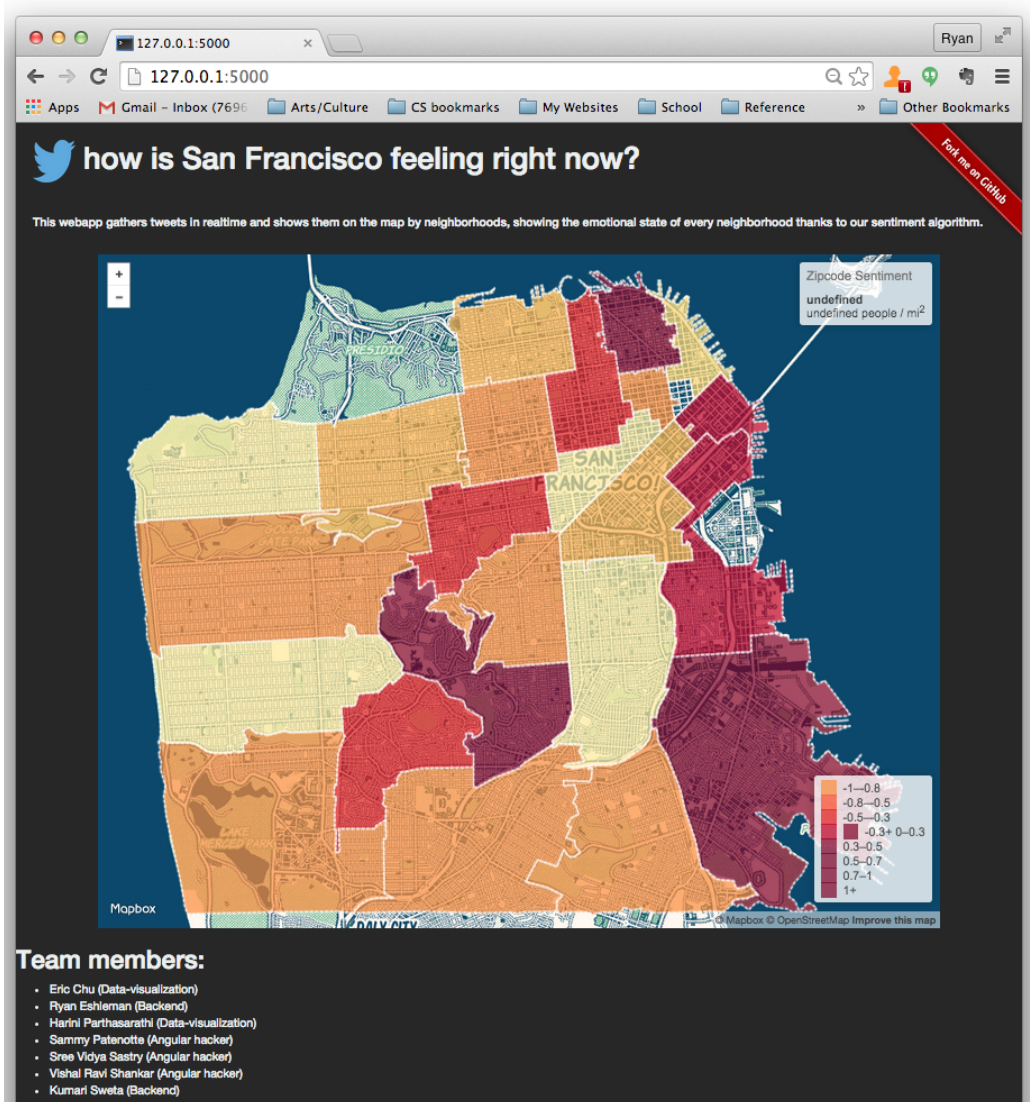
Specification of GUI

disclaimer: Some of these images need to be cleaned up, and will be updated accordingly

High Level user story:

A curious reader comes across the web application with the intention to find the state of sentiments in San Francisco. He or she loads the main page to locate a map of San Francisco in its full glory. The map is interactive and the reader gets to play with the colorful zipcodes shaded in different colors. Each color representing a sentiment state. Hovering the mouse pointer over a zipcode displays the sentiment of the noted zipcode in a neat little popup bubble. Clicking on the zipcode displays a time series graph of the zipcode's sentiment history.

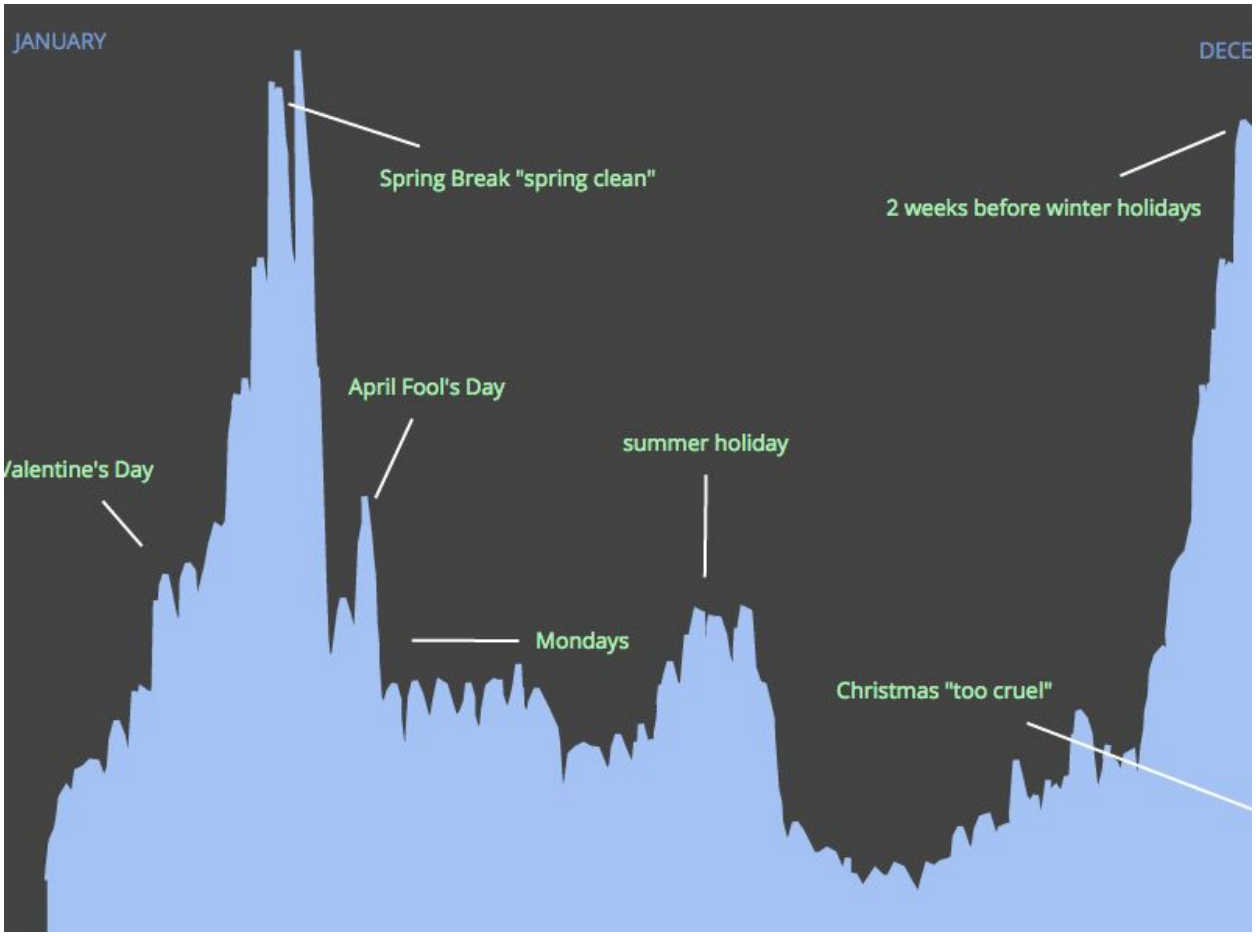
Home Page



1. **User navigates to homepage. page is updated with the current sentiment of each neighborhood. Zipcodes are color coded based on sentiment.**

2. A key in the bottom left corner explains the meaning of each color.

3. User hovers over neighborhood, the border changes for visual feedback and an info box in the upper right corner is updated with information about the neighborhood.



4. User clicks on a neighborhood and is presented with a graph of the sentiment change over time for the neighborhood. (The format of this graph has not yet been specified, similar to the above, though)