Twistory

Bryan Salas, Boomer Truong, Terrence Maas, Wu-Yang Tao, Young Seo, Nick DeChant

Introduction

What is the problem?

The inspiration for Twistory is about finding a single solution for a two-fold problem. First, we always need more reliable knowledge to help us understand the world. Second, society's technical advances have left us trying to cope with and understand the enormous amounts of data we have amassed. This project aims to help bridge the gap between raw data and reliable knowledge by interpreting and organizing data from Twitter to form coherent historical narratives.

Twitter is one of the world's largest social networking and blogging platforms today. Founded in March of 2006, Twitter has revolutionized the sharing of information and how we look at trends in data on the internet. From people's favorite foods and sports teams to daily life experiences, users can tweet about anything they can think of. With the introduction of the hashtag (#) feature, posts can be tagged to particular strings corresponding to a specific topic at the user's whim (whether relevant to the post or not). This association of hashtag and tweet creates an interesting opportunity to have insight into the culture of social networking on the platform. It is because of this that Twitter could

be considered a large source of useful data for marketing, personal, and other professional reasons.

However, Twitter is limited in how it displays its data, including trends and other user information, to the general public. As a team, we decided to find a solution to this problem. Twistory is a website whose purpose is to provide not only isolated trends but also a coherent framework of connected events. Twistory categorizes hashtags into relevant clusters and provides more information on more popular users (including their hashtag using habits). Twitter arguably contains an overwhelming amount of data. Twistory organizes the data into more user-friendly form. Our goal is to provide the best possible user experience so that our users can obtain the most accurate and meaningful knowledge they are searching for.

What are the use cases?

A user might approach Twistory for a number of reasons. Twistory is a user friendly site where a user can easily view Twitter data. We currently organize our data into three main categories. First are Twitter handles. Twitter handles are the username of the tweeter prefixed by the '@' symbol. This category holds various bits of information about the tweeter such as their Twitter profile photo, username, any videos, biographical information described in their profile, graphs describing their tweets in relation to other data, a number of relevant tweets, and relevant pages in regards to the other two main categories. Second, data is organized into a category called hashtags. On the Twitter platform, hastags, prefixed by the octothorpe symbol, are named tags describing or

indicating a theme (accurately or not) for a Tweeter's tweet. This category will include data such as the usernames of the tweeters who have utilized a particular hashtag, relevant graphs marking popularity of the hashtag, descriptive information, popular tweeters who have used the hashtag, and related pages in regards to the other two main categories. Last we organize data into the clusters category. Clusters is a category that describes the type or "genre" of tweets. Clusters may include data on usernames, members, parents, timeline information, descriptions, key features, and related pages in regards to the other two main categories.

Twistory's greatest potential lies in applications related to politics, economics, and sociology. Most, if not all, social sciences struggle with the limitations of acquiring real-world data. Consequently the majority of statements, opinions, and conjectures in these fields have no serious way to be tested, proven, or disproven. For instance, a piece of political journalism in a given country is likely to contradict countless other pieces covering the same topic. Without a way to evaluate these pieces, people have no way of determining what to believe and what not to believe. Twistory brings to the table empirical evidence about what people really think and how sequences of events unfold.

Twistory API Design

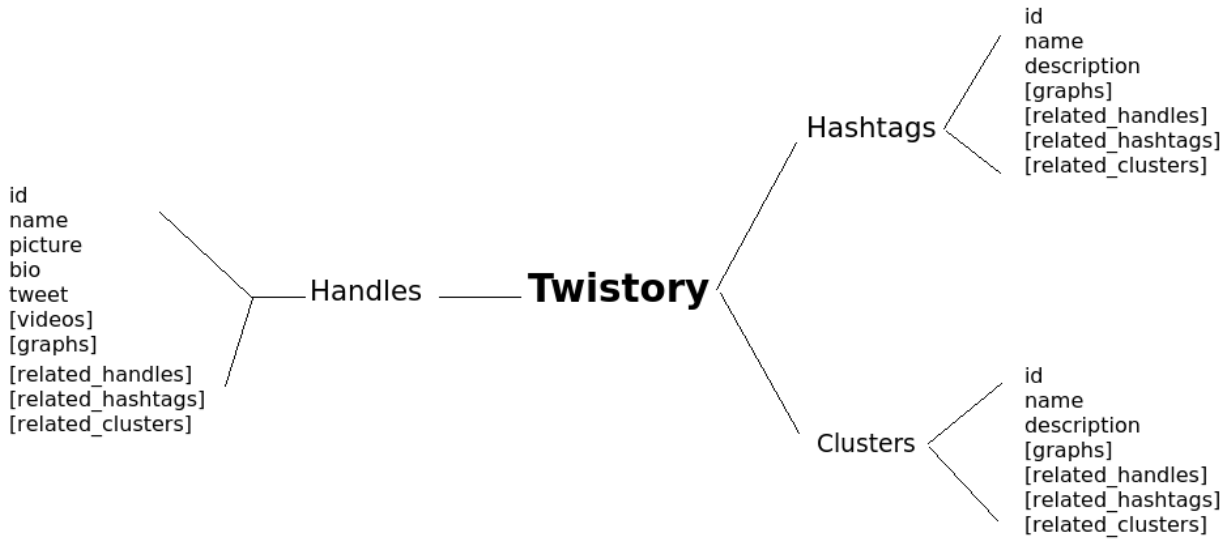
To adhere with the principles of REST, the general syntax for the resource URL should be undercase and plural, so if the API contained a resource for "dogs" the URI path

would be "/dogs" and corgis resources would be "/dogs/corgis". If a resource identifier should span longer than one word, a single underscore will be the delimiter to group the words together, "/ugly_cats" or "/dumb_kittens". The main goal on deciding a style was to keep resource URLS consistent for our users and future developers working on this API. With all of the Twistory resource paths, when a HTTP Request is made, there should be a version code with the resource path following it. The version code will help identify changes in the API and increase if there is a change to the API that could potentially give the users a lot of headaches. So if the developers decide to deprecate a resource the users would have to upgrade accordingly.

The next important part of the Twistory API is that the documentation is regularly kept up to date, detailing changes to a resource or documenting new resources, so users are able to take advantage of the API. REST explains that resources should be a noun, as the HTTP Requests describes the verb or action upon the noun: "GET /dogs/corgies". The Twistory Docs for any resource all follow a general outline by starting out with a description of the resource and its benefits. Next should describe a single instance or your resource and what the end point would look like,"Dogs [/dogs/id]", and then describe the endpoint to retrieve a collections of those resources. This format clearly shows how a user can easily request a single object or a collection of them and making the API more user friendly. Lastly, if the resources or collections have parameters like an id, there should be a row that includes the parameter, a description of it, whether this parameter is optional or required, and an example of how the request would look.

Assuming a successful request was made, The Twistory Docs shows the formats response model with a header and a body. Currently The Twistory API will respond in JSON format as the content-type. The Header of the response will contain metadata about the resource, i.e. date last updated, language, character encoding scheme. The body will be the resources and the attributes that were decided upon. Finally, handling errors or bad requests is done by choosing a numerical value within the HTTP status codes which identifies what the error was, as opposed to a 404 page that would be extremely difficult to debug.

Our API is designed to be read-only, therefore the users can only use the GET method. The 3 main resources we define are handles, hashtags and clusters. The mind map below shows all the attributes of each resource:



There are several attributes that can contain multiple elements, namely those with square brackets around in the mind map, such as the attribute "videos" of handles and "graphs" of hashtags. In our blueprint, we decided to make each of those attributes hold an array, and the entries in the array are the elements of each attribute. Note that it is possible for those arrays to contain nothing, allowing for the possibility that those attributes do not have any elements.

For each resource, the user can use 2 types of GET. One method will return a list of all ids and names in that resource. The id here is an integer we assign to each object for this website. This id is the primary key of each object in each resource, so if a handle object has id = 1 there can still be a hashtag object that also has id = 1, since handle and hashtag are two different resources. For handles, the name is the one created by the twitter user, not the twitter user's actual name. Also, when calling the list of handles, not only the id and name, but an attribute called "picture" will also show up. We intend to make this attribute returns an actual picture, but right now we only have "link to pic" in place. There are other attributes that have content with a similar format as "link to ...", and we intend to make them behave similarly like "picture" from handles.

The other GET method requires the user to give an id, and it will return all attribute of the object with that id in the resource. If the user gives an id which we do not assign, a 404 "not found" error message will be returned.

There are 3 attributes which every object in every resource has, those are "related_hashtags", "related_handles", and "related_clusters". Each attribute displays all the related objects of the indicated resource to the object with the attribute. Those attributes are all arrays. The entries in these arrays all have the same format, and an example of an entry in "related_hashtags" array is {"id": 2, "name": "WorldCup", "page": "/hashtags/2"}. The "page" here refers to the address given to that object in this website, so /hashtags/2 will go the page of the hashtag object with id = 2.

Django Models

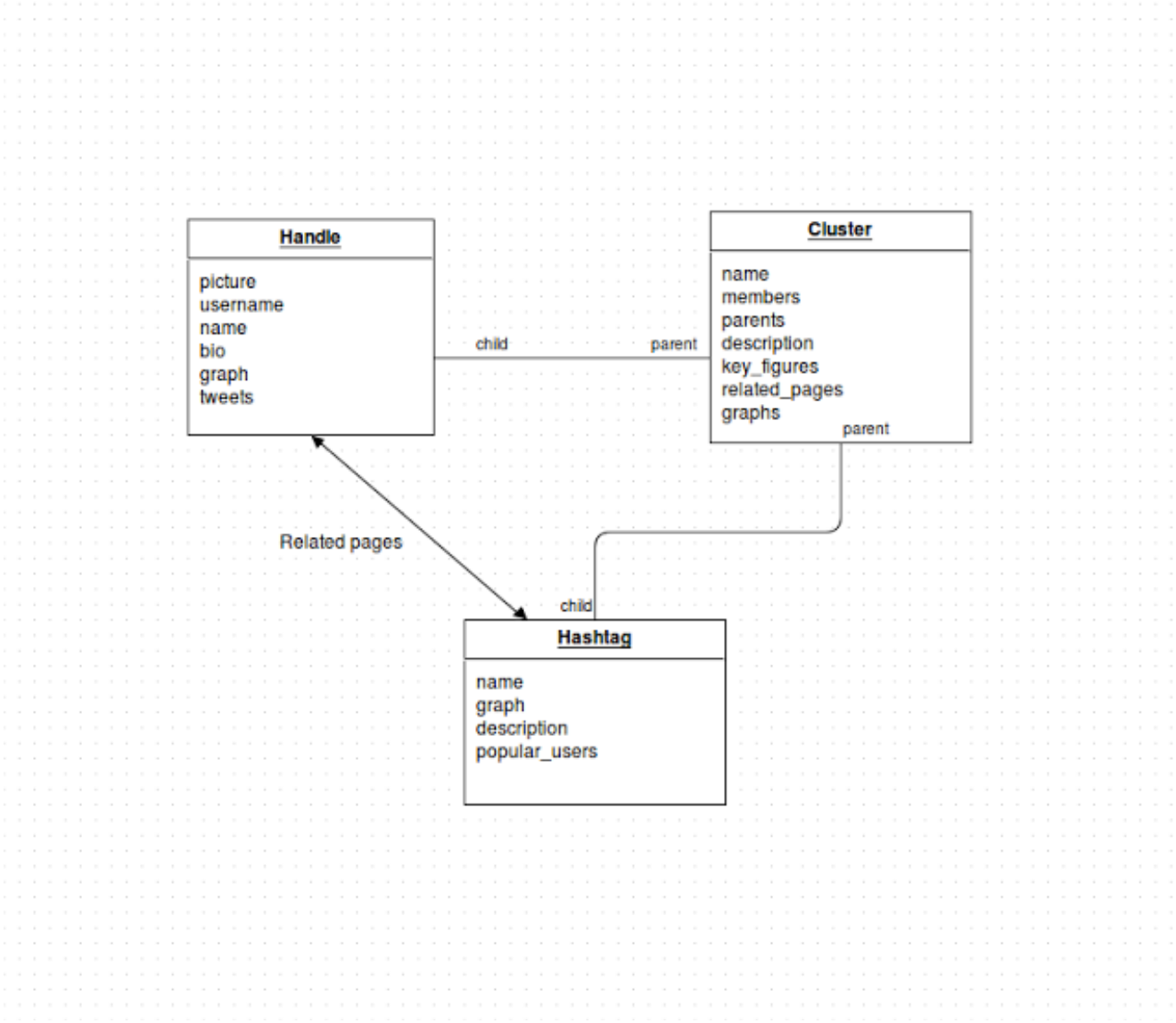
We designed our model to represent the three categories (handles, hashtags, and clusters) we discussed above. Each category represents a class which is defined by the attributes that are within that class.

The handle class has the attributes picture, username, name, bio, graph, and tweets. The picture attribute holds a picture of the user under the handle that we are currently concerned with. The username is the handle created by the user. The name is the actual name that the user goes by in person. The bio is a short biography of the user. The graph will be a picture of a graph of how much attention the user gets from her or his social media platform. The tweets will be a list of the user’s most popular tweets (hashtags).

The hashtag class has the attributes name, graph, description and popular users. The name is what follows after the hashtag (#) sign. The graph is of the hashtag's usage by users over time. The description explains the hashtag's meaning as some may not be so obvious. The popular users are a list of popular users who have used/retweeted the hashtag.

The cluster class has the attributes name, members, parents, description, key_figures, related_pages, and graphs. The name of the cluster refers to a grouping or a genre of the hashtags that relate to some events. The members are a list of related hashtags that fall under the same cluster. Clusters are important because they can mark a specific event or happening. A popular combination of hashtags used together can indicate a specific event taking place or trending topic. The parents are a larger grouping of the hashtags that group different clusters together. The description explains the cluster's meaning. The key figures attribute is a list of popular users that have tweeted the hashtags in that cluster. The related pages is a list of pages of different hashtags that are related to the cluster. The graphs show the different hashtags in the cluster's usage over time.

The handle class has a many-to-many relationship with hashtags. The hashtag class has a many-to-many relationships with handles and clusters. The cluster class has a many-to-many relationship with hashtags. (describe special cluster-member relationship).



Unit Tests of Django Models

For phase 1 of the project, we do not have a running database. With this in mind we still wanted to write effective unit tests for its future implementation, so that we know exactly how we want it to work. The following is a technical explanation for our unit tests:

function create_handle():

- This function puts data to our django model for handles. This function is called by the other handle tests to set up model objects to check for correctness.

functions test_handle() 1 - 3:

- These functions check the different model types for username, name, and bio by calling create_handle() and using assertEqual with __unicode__ to check for correctness. This is important because the models implement charFields for these variables and we want to make sure they are being set up correctly in the database.

function test_handle4():

- This function checks for the corner case where blank data is expected from the models. Different fields handle null and blank entries differently so it is important to check this for correctness.

function create_hashtag():

- This function puts data to our django model for hashtags. This function is called by the other hashtags tests to set up model objects with different values to check for correctness.

functions test_hashtag() 1 - 3:

- These functions check the different model types for name and description models. We are not sure exactly how we want to implement the graph and popular_users fields yet so we left those for future experimentation when we have more solidified data. These functions use assertEqual with __unicode__ and the expected strings to ensure that the strings are returned correctly.

function test_hashtag4():

- Similar to the test_handle4() function, test_hashtag4() checks for the case of null and empty inputs to our models as different fields handle null and empty strings differently. It is important to check for correctness here so that our future product will not break down over null input and retrievals.

function create_cluster():

- This function sets up a cluser model object. This function is called by the tests to put different values in the model to check for correctness.

functions test_clusers() 1 - 3:

- These tests check the name, parent, and description fields. The other elements of our models include lists of data from the databases. As we are not sure how we wanted these formatted, we did not include expected data from them in these tests. As we gather more information about how exactly we will be extracting Twitter data, these tests will become more robust. These functions use assertEqual with __unicode__ and expected string checks to ensure the appropriate returned strings.

function test_cluser4():

- Like the 4th tests of the previous models, this important tests checks null and empty string values for our models. Different fields handle null and empty strings differently so it is important to check to make sure these corner cases are handled correctly.