Design Document

By Victoria Gong

A. Design

I. Purpose

This project is a clone of the twitter and aims to provide a space where users can:

- login / signup
- post, edit and delete tweets
- view own tweets
- view tweets of others
- see other users
- follow other users
- visit user's page
- retweet other user's tweets

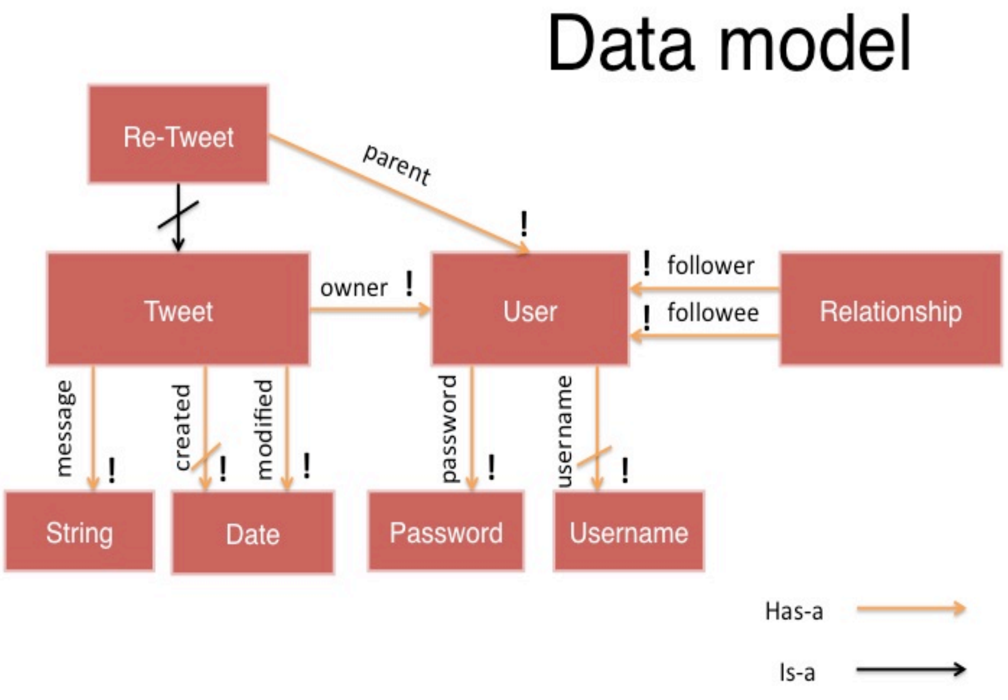
The application is an Express Application built on MongoDB as its database.

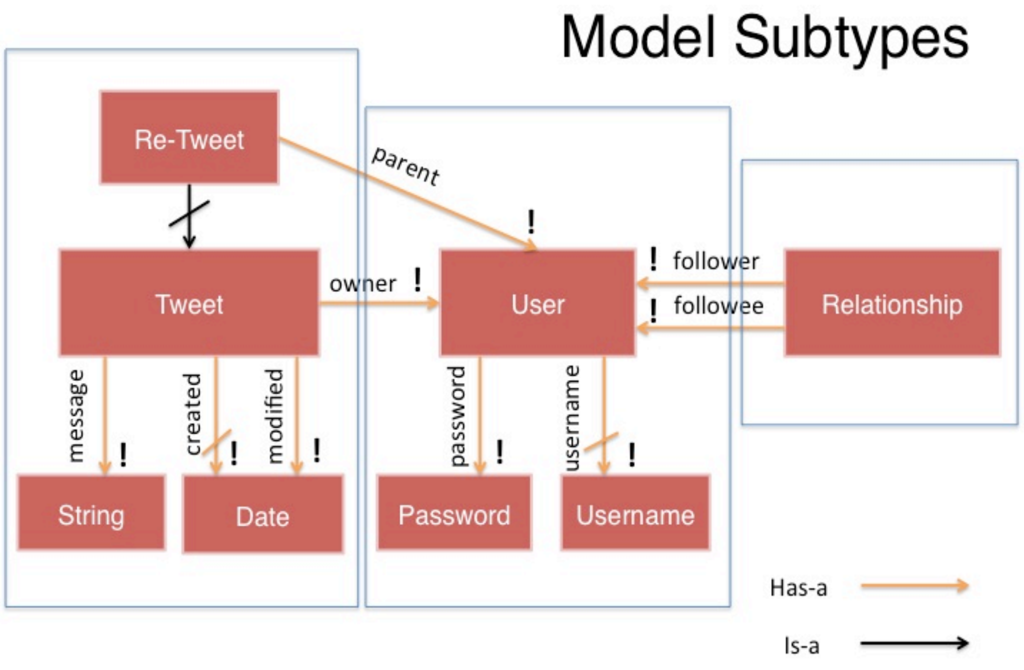
II. Approach

The design of this project revolves around a MVC based approach. The models are Moongoose schema models, the viewer is rendered through EJS and HTML, and the controller on Express/NodeJS.

The models provide the structure for the data.
The viewers render each page.
And the controller retrieves and functionalizes the data for the viewer to use.
For more information on the file structure, look below.

III. Data Model





Mongoose Schemas
The data model of this application revolves around three simple but important models:

User: Only contains information needed on the minimal account

Tweets: Holds all the 'tweets' ever created on the website

Relationships: Holds all the relationships between any two users

Considerations

1) Mongoose vs. Monk

Many argue that one of the many benefits of MongoDB is that it is schemaless. However, any good design also must have some data model, which is a less formalized version of a schema. Moreover, this project will soon contain many complicated relationships (relationships between users and message, users and other users) and features. Thus, I have decided to use Mongoose instead of Monk so that I can go through an object oriented approach in building my model. I also felt that using Mongoose allowed an easier way for me to modularize my code (MVC).

2) Tweet + User Schema - Relational vs. embedded

I first weighed several options:

a) Having one large document: User contains username, password, a list of messages
b) Having several tables: User contains username, password. Message contains username, message information.

I opted for b) because while a) would provide fast retrieval of information for a specific user (i.e. showing each user their own mesages), it was not ideal for displaying tweets of all users, searching through tweets for some common feature, sorting tweets by date or even editing messages. Instead, such actions would require the retrieval/sort/search to go through another level in each object (user --> message_list --> message). For example, for a) editing messages would require fetching a user and then fetching a message. However, for b) editing messages would only require fetching the message object.

Meanwhile, we can still search through all the messages with a given username pretty fast since the username is a top-level attribute of messages.

Benefit:
- Implementation of needed features was very simple
- Fast access
- Clear concept

3) Relationships Schema - Relational vs. embedded

I also weight several options for this design:

a) Having each user contain a list of those he/she has followed
b) Having each user contain a list of followers and followees
c) Having a separate table just showing the relationships

I opted for c) because of the following points:
Analysis of a). The benefits of this design is that if we wanted to get a list of people a user has followed, it is quite simple. Updating a list of users (while it requires 2 layers of access) just needs to modify one user's object. If I wanted to retrieve a list of people who follow a certain user, however, I would have to search through every followed list of every user, which is very inefficient.

Analysis of b). The benefits of this design is that getting both followers and followees is quite simple. Creating updating or deleting a relationship, however, requires writing to two different users: One user's followers list and the other user's followees list. Moreover, this means there will be duplicated data (data is in both user's object).

Analysis of c). The benefits of this design is that both creating/updating/deleting a relationship and finding out a relationship is quite simple. Creating a relationship is very simple in that you just set follower and followee and add it to the table. Updating/deleting a relationship just requires a simple query into the relationships table where followee or follower is defined, and simply removing it and for updating, creating a new one. Finding a relationship is a simple query as well with only either followee or follower defined. Neither of these functions require accessing the user table at all, whereas the above two required diving into the user's object.

I ended up choosing c) because it gives the most flexibility in adding in new features related to followers/followees (such as showing filtered newsfeed as I describe below, showing a list of followers, showing a list of followees) and because it is very simple in both creation, editing and access.

4) Schema for implementing retweeting

I decided to just add another attribute to the Tweet schema (parent) because the very essence of a tweet and a re-tweet is quite similar. Their message content is essentially the same at that point in time, and there is someone it comes from. We can copy over the message quite easily, and now we just need to hold the parent information which is the new attribute. Having them have the same schema means that a re-tweet still has all the same functionalities as a tweet (can edit, delete, etc) quite easily without much additional implementation. If, instead, we had a separate rep for re-tweet, we would have a lot of repeated functionality and code.

IV. Additional Challenges & Design Choices

1) Designing the edit/delete mechanism for tweets

Figuring out the best way to edit/delete messages was a bit difficult. However, because I can easily access all messages through simply passing in the ID, I chose to store the model.ID as the div.ID of each div that contained the message. Although this was not as secure, it provided an easy way for handlers to make a proper post request. Any button can just get the ID of the parent div and use that as the paramters for a post.

I did think about implementing a hashing function so that the mongodb created ID was not revealed in the HTML.

2) Designing the user-facing edit/delete mechanism
There were two considerations I had: popping up a modal to change the message or using javascript to turn the text into an input box. I chose the latter because I felt it was easier for the user to make changes, and editing posts might be a more frequent habit.

3) Designing the additional feature of retweeting
At the moment a user chooses to retweet, there can be various possibilites that can occur. There were two which I considered to implement:
a) Get the tweet message and owner through event handling and passing that information to create a new tweet
b) Getting just the tweet id, finding the tweet in the database and use the information in the database to create a new tweet.

Although the latter makes sure that if a user A deletes one of his tweets and user B did not refresh and still sees that tweet and wants to retweet, user B can't actually retweet, I decided it was more important for the user doing the retweeting to be able to retweet exactly what he sees at the moment. Thus in the scenario where user A is editing his/her tweet and user B wants to retweet the original tweet, user B can do so without having to worry about whether user A is editing or not. The most optimal solution would actually to have the interface update so that all updates are reflecting in the user interface, but for the scope of this class, such real time updating of the ui is not implemented.

4) Designing the additional feature of followers
There are numerous ways where you can have the feature of followers.
For my project, the main idea of followers is for any user to have a way to easily keep track of people who they are interested in, keep track of people who are interested in them and to see the content of these people, all for the larger goal of being able to discover and dive into a different relationships.

Thus, I decided to implement this feature such that you can only follow a user when you land on a user's page (when a user lands on a user page, the entire focus is on that user). However, once you follow that user, that user will appear as a name in box on your right in your homepage, where clicking on the name brings you directly to their user page. Thus, this allows you to always easily access thier homepage again to see their tweets. There is also a box on the right telling you which users have followed you, and the reason for its prominient behavior is so that you can also see if you are interested in the people who are also interested in you.

I decided to leave the dashboard page a feed of ALL user's and not just of those you are following because it allows for more discovery of different people, and the tweets of those you follow and those who follow you can already be accessed through your home page (and through a more individual way).

Once there are a lot more people on the site, however, the dashboard should certainly show a filtered newsfeed (or have the option to do so) because there will be many tweets that might not be of interest. However, since the website has a small user base as of now, I have decided to leave it out as part of a purposeful design decision.

I did, however, design the model with the idea that it could be easily extended to sure a filtered newsfeed. Here is what getting a filtered newsfeed entails:

- Find all Relationships where those you followed is the followee.
- Use that list as a query for finding messages with owner as those usernames.

Thus, the implementation is quite easy with this design.

V. Design Benefits

1) Simple and concise semi-relational schema

- Look to explanation above

2) Server-side timestamping and time storage

- Everytime a message is created, a Date.now() object is created so that each message is timestamped right before it gets saved in the database. This makes the time more accurate, especially if there would be delay between client and server.
- Storing Date objects allow us to transfer the later json encodded string back to a date and thus display date's in different versions. I.e. I can show year, month and/or day by some logic (as I do) instead of just displaying a String that was saved right at message creation.

3) Modules

- This is more of a design benefit for the code, but by creating modules for messages and login/signup, I can :

1) modify the look and functionality very easily by replacing just the module files
2) Make many copies of either messages or login/signup boxes so that I can put them on different routes/pages.

Note: I decided to put them in a separate module folder under public instead of putting it into css and javascript because it is easier to understand and see the difference between page css/javascript and css/javascript related only to specific parts of the page (i.e. modules).

4) View separation into dashboard, home and user instead of user and index.

I made the views separate into dashboard, home and user because I felt that the dashboard could be a feature that contains many other features. For example, your dashboard (which shows other users and tweets) can only show you various modified dashboards by extending that view.

    dashboard/followed -- show a dashboard of tweets form people you follow
    dashboard/interest -- show a dashboard of tweets based on an interest

I also made home by itself because ths would take care of all user-specific related pages. For the future, this could be

    home/account
    home/my-tweets
    home/my-followers

Index takes care of all the required views for getting setup with the site (landing, logging in, and signup).

Thus, the view separation has been built and structured by keeping future possible features in mind and grouping them together.

B. File Structure

app.js: The main server to be run. Contains all setup

controllers/: Contains the controllers for each model.
(Connector between model and view)
- messages.js: Controller for Message Model
  - Takes care of creation, modification and removal of messages
- users.js: Controller for User model
  - Takes care of creation, login, logout, authentication and sessions of users.

models/: Contains the models
- message.js: Message model
- user.js: User model
- relationship.js: Relationship model (TBU in part 2)

views/: Contains all the views
- dashboard/: All the files related to dashboard view
- home/: All the files related to home page view
- index/: All the files related to landing and logging in

routes/:
- dashboard.js: routes for /dashboard
- home.js: routes for /home
- index.js: routes for /

public/:
- images/: contains all images used
- javascripts/: contains all javascript used in pages
- modules/: contains modules
- stylesheets/: conains all css used in pages

public/modules:
- message/: contains js & css related to module that renders
  the individual messages.
- signup-login/: contains js & css related to module that renders
  both signup and login box.

C. UI

Design is currently structured in the following ways:
/: Is the landing page where users can sign up
/login: Is the page where users can login
/home: Is an authenticated user's home page of his/her tweets
Also where user can create, edit and delete tweets
/dashboard: Is where any user / visited can see tweets and
current users.
/users: This holds the user page for each user where you can view their
tweets as well as see the number of tweets they have.
For example, a use would have /user/vicky>

D. Viewing

http://fritter-vickyg.rhcloud.com/