Project 2: Twitter Clone

Grading directions

To run the web app, go to the URL http://fritter-eunicel.rhcloud.com/, which is also linked in the README.md file in the root directory. The files are available on project 2 github. Click the "Sign up here!" link to create a new account. Enter username and password to login. From the dashboard, navigate to the different pages by clicking the buttons. To log out, click on the "Log Out" button on the upper right corner.

Part 2

Highlights

Pointers to best parts of project and justification:

- *Modularity between model and controller*
  In additional to being separated from the views, in phase 2 of the project, the model and controller are also separated. There is a user model and tweet model, both contained in a model folder. This contains the schema of both models and the methods that modify these models in the database. The control is stored the routes folder, which contains three files – index.js, users.js, and tweets.js. These files control the routing between the files, such as redirecting and rendering the pages, and call the models' methods to modify the models.

- *Like and unlike feature*
  One of the best parts of this project is the like and unlike feature. Each tweet in the newsfeed has either a like or unlike button. The user can click the button to add or remove his or her self to the list of users that like this particular tweet. It also displays the other users who like this post, unless there are too many users to display. In that case, it will display two users and the number of other users who like it. This prevents the message from crowding up the screen if many people liked the tweet.

Help wanted

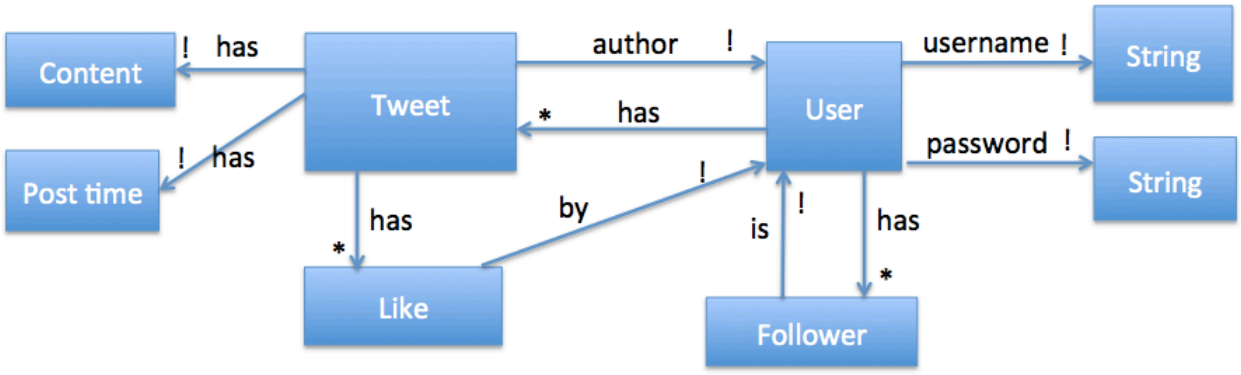
Requests for advice:

- *For loop with callbacks*
  I was not sure what is the best way to include functions with callbacks inside a for loop. For example, I wanted to have a for-loop to go through all the followers of a user and then populate each of the followers to get the tweets of each follower. However, the callback function would cause the function to go somewhere else. The best way I thought of was to use recursion and keep a counter so that it breaks out of the recursion when the counter reaches the final follower.

Design

Design challenges

- *Data model explanation*



The two main components of this project is the Tweet and the User. The User model consists of a first name, last name, username, password, a list of followers, and a list of Tweets. The first name and last name are just stored for displaying purposes so they are not pictured in the diagram. The username is used to identify the user since every username must be unique. The password is used for authenticating the user when they log in. The followers' list is a list of zero or more Users and the list of Tweets is a list of zero or more tweets. The Tweet model has a content, a post time, an author, and a list of likes. Each tweet has exactly one author, who is a User and it may have zero or more likes, which is also a list of Users who like this tweet.

- *Data model: followers vs. users followed*
  The main design challenge of this phase was designing the data model for following. One model would be to have a list of users that each user is followed by. This way, I would g through all the tweets and check to see if the current user is a follower of the author of the tweet. The benefit of this design is that the tweets would be ordered correctly, however, I would have to populate the author of each tweet before I can check whether the current user is following them or not.

  The other model would be to have a list of users that each user is following. I chose the second model because fritter needs to be able to display all the tweets of the user and all of the users the current user is following. Therefore, it would make more sense to know the users that the current user is following rather than the users who are following the user. A problem I ran into in this method was that since I was getting the tweets from each follower, the tweet ordering would be based on the follower, such that all the tweets from one user would be grouped together, rather than having all the tweets be ordered chronologically. I solved this problem by recording the time each tweet was posted and sorting them based on that.

Part 1

Highlights
Pointers to best parts of project and justifications:

- *Modularity between views and model/controller*
  The view files are separated from the model and controller. The view file is in a views folder and the model/controller is in the routes folder. This organizes the project and makes it more modular, which will make it easier for testing different parts of the project later on.

- *Modularity of view pages*
  The view files are separated into different folders. For example, the index folder contains the index (home) page, the user folder contains everything related to the user (signup, login), and the tweets folder contains everything related to tweets (compose, edit, dashboard).

- *Modularity of controller*
  The routing code is separated into different files. The routes relating to user, such as logging in, signing up, and logging out are all in a file that is different from the routes relating to tweets, such as composing, editing, and deleting.

- *Flow between pages*
  The pages of this web application flow very nicely. There is no homepage, because the only purpose of the homepage would be to lead to sign up or dashboard pages, acting as an extra barrier page for most users most of the time. User's login information is stored in the session so that whenever a user navigates to a different page or if they close the tab and reopen it, they do not have to sign in again. This makes application easier to use. However, when a user logs out or before they log in, any page will redirect them to the Log In page where they can log in before they view the content.

Help wanted
Requests for advice:

- *AJAX*
  I think it would've been helpful for us to learn about AJAX earlier in the process, because it would allow us to implement posting tweets from the dashboard page and having them appear in the feed immediately without refreshing or navigating to a different page. It would allow us to implement editing the tweets without going to a separate edit page.

- *OpenShift documentations/instructions*
  I think it would be helpful if there were more documentation or instructions earlier on why and what code should be added, especially in server.js, to deploy the project to OpenShift.

Design

Design challenges

- *Use of database*
  There are various ways to store the information in the database. One option is to store all the users in database, each username field and a password field, and store all the tweets in a separate database, each with a username (author) field and a content field. This option keeps the data separate and makes it easier to search for users or tweets using each of their fields. However, it would be more difficult to find all the tweets of a user given their username. Another option is to store all the users in a database, with a username field, a password field, and a list of tweets contents. This makes it easier if we wanted to get the list of tweets given a username, but makes it more complicated if we just want to get all the tweets. I chose to implement the first option because this part of the project only requires us to display all the tweets without filtering them by the user. Thus, it would be easier to have one database just for tweets and a separate database for the users.

- *Choice of HTTP verbs (get vs. post vs. put vs. delete)*
  There are various way to edit and delete posts. For example, deleting a tweet can be implemented using a HTTP post or a HTTP delete. HTTP delete is good because it is an idempotent method, so if the resource is deleted multiple times, it will not cause a problem. HTTP post is good because it can be used to send more information. It may not matter right now, but if tweets were allowed to be very long, or if tweets had more information associated with it in the future, such as followers, tags, links, etc. then using a post may be a better option. I chose to implement deleting a tweet using HTTP post because in the post callback function, I remove the tweet from the tweet collection. If the post method was called multiple times, the tweet cannot be removed more than once, so even though HTTP is not an idempotent method, it will not cause problems in this case.

- *Links vs. forms*
  Users can navigate to pages through links or forms. I used a combination of both for this project. By clicking a link, the user is simply taken to another page. This useful when the link's sole purpose is to take the user to a different page. For example, this project used a link to take the user to the sign up page if they need to create a Fritter account. By clicking a form, the user can get to another page but also send information through the form. This is useful when we would like to pass along the information as the user is directed to a new page. I used forms when logging out, logging in, editing, and deleting. This is because, for example, when a user clicks Edit, in addition to taking the user to the Edit page, we also need to pass along information, such as the id of the tweet, in order to find the content to display in the edit box.