DESIGN DOCUMENT
FRITTER
Deployed at: http://saadiyahfritter-trial18800879648.rhcloud.com/

Abstract:
The general structure of the system is as follows. There are users who can sign into the system and there are posts that are made by various users. The users are stored in a collection called "users" and the posts in a "posts" collection.

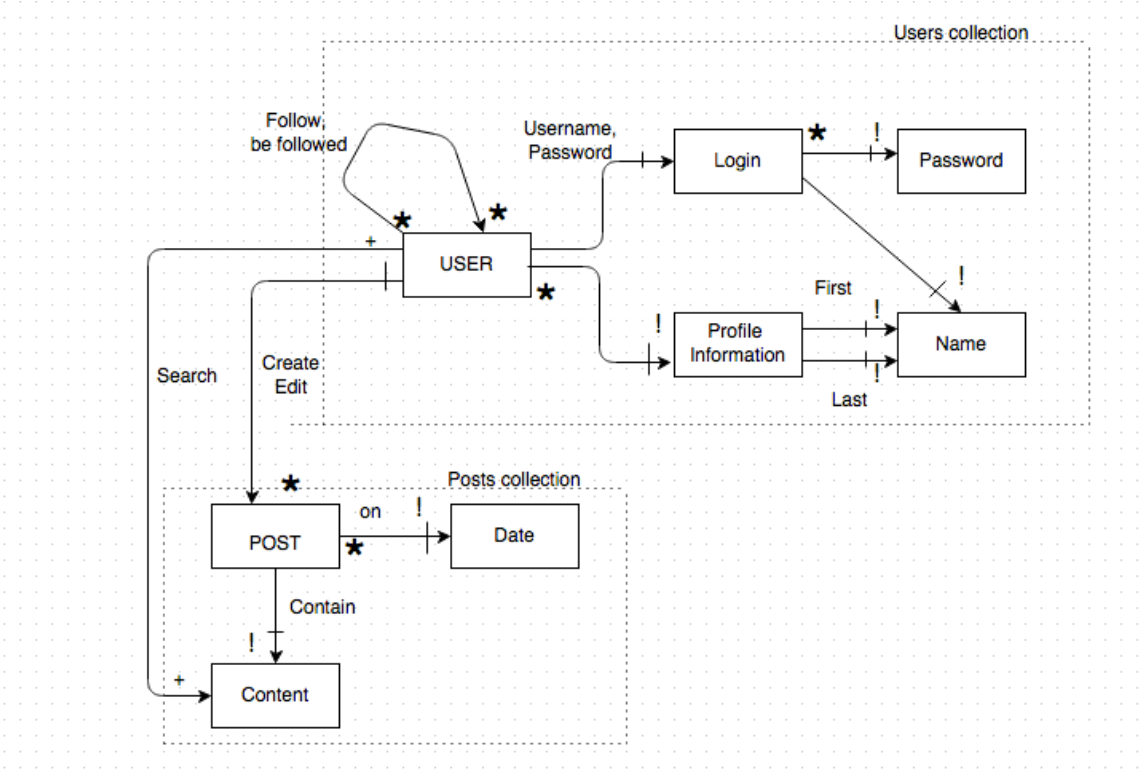
Key players:

- The User:
  The user has a default _id , a user_name and a password as well as a name field (not used for authentication). This is created during the sign-in process and enables logging in even when the app is restarted (persistent).
- A Post:
  Only a user who is authenticated can make a post. A post has a default *_id, a post field as well as a user*_name, indicating which user made this post.

Grading highlights:

- I believe that I enforced quite some modularity. By making the methods in routes very independent from one another. Each method, if need be would have a GET and a POST version, and if needed they would have a corresponding view. The textboxes for posts were formatted using the CSS file and not individually to enable changing the overall look very easily.
- Also, all the logic pertaining to users signing in/up was handled inside index.js and the logic for posts displaying a particular post (in the edit post page) (that is, delete and update) was handled inside posts.js. Other operations that only involve displaying posts was handled in index.js because they did not involve altering the post itself.
- I tried handling edge cases as much as possible. In particular, I like the way in which I enforced no posts or users could contain "" fields.
- I tried maintaining some consistency throughout the application. Luckily, I could use the same "require" parameter in all the forms in the app that required user input, to prevent awkward behavior (such as empty posts, or undefined users).
- I particularly like the pipelining of the pages. I tried to prevent weird behavior (such as unauthenticated users editing posts) as much as possible. Thus the pages were arranged in a logical order so as to ease navigation.

The Data model



Navigating the app and design choices:

The app, upon starting renders the **log_in page** (*welcomepage*). This enables a user to log in or to be redirected to a sign up page.

The **sign-up page** collects new user data, if the info is valid (a username not in the database and not equal to “”), redirects the user to the sign in page.

If unsuccessful, the page redirects to a **failure page** that enables the user to go back to the sign-up page to retry.

If a login is successful, the user is sent to the **make_new_post page**. There he can post a non-empty post to the posts collection or be directed to the *allposts page*.

If posting is successful, the user is redirected to a **success page** and from there he can go to the *allposts* page. If unsuccessful, he will be redirected to the *sign-in page* (because it makes sense to prompt a user to sign in to make changes).

The **allposts page** allows users to see posts by all users of the app. Each post has a link says, “edit post”. If a user is the user who made the post, he is redirected to the update_post page. Else, he is redirected to the *allposts* page.

The **allposts page** also supports a search feature and also enables the user to navigate to the user-list page where he can manage who he follows.

The **allusers page** enables the current user to manage who he follows.

The **update_post page** allows the user to do two things: delete post or update post. If an unauthenticated user navigates to this page, he will not be able to make changes and will be sent to the log-in page if he tries.

The **logout page** is accessible by clicking one of the logout links on pages such as: make_new_post, allposts and edit post. This is so that a user can leave if he does not want to do a particular action, such as post a post or modify it.

Design challenges and decisions:

- Authentication:

  For this section, I could either use cookies or session data. Cookies has the advantage of being persistent (so a loss in server connection would not kick the user out). However, the sessions were easier to implement and given the scale of the application, seemed like a better choice. If need be, I will later make use of other frameworks such as Redis (some googling revealed that It is widely used), thus the simple sessions implementation can easily be replaced whereas a cookies approach will be a little trickier to move to Redis.

  Checks were also included on each page, especially the edit post page, to ensure the user editing a post is actually the one who made the post.

- Managing login input and ensuring users input data before submitting a request (such as search/update post):

  There were essentially two options to solve this. Either integrate checks for each text box for the fields and somehow tell the user when some input is wrong OR to enforce that the user can only make the request with valid inputs. He first method involved including checks inside the JS code, then somehow notify the user about invalid inputs/corrections to make.

  However, the second option merely involved putting constraints on the textboxes that gather data. This enabled me to not clog the js file with checks and keep the code straightforward and easy to understand and modify. Thus the second option was chosen and was implemented in the ejs files (HTML) by requiring those fields. So if the user tries to submit an empty form, the form prevents the form from being sent and points to the field that requires some changing.

  The same logic was applied to posting as well as editing a post, a user cannot simply delete the post content and update it, he will have to delete the post to get rid of all its content. To see this behavior, simply try submitting something with empty field.

  Observation: if one of the above were not used, then a user could submit an empty sign-up page then anybody could click login on the login page and be signed-in as "".

- Following

  There were two choices to be made when it came to the wrapper to be made, I could either stick with monk, which I used in phase 1 or move to mongoose. I chose the latter because of the schemas that enable me to use populate. Thus made the subscription and rendering of the followers process much easier. Another choice was whether a user also keeps track of the people following him or not. I chose to make it a two way relationship so that a user

can know who follows him. This was done, so that in the future, I can implement a notification system, when a user makes a post, for example, I can notify all his followers.

- Other decisions:

  One can go to a URL and view posts (allposts) or even view the posting page (for now), later I plan to allow outside users to make a post to fritter as anonymous users. Which is why I chose to leave in this feature.

  The pages were kept simple and made intuitive by allowing the users some options via a logout link or a go back link or essentially a way to be redirected elsewhere (in as logical a place as I could think of).

  My additional feature is a search feature that enables users to search through tweets. I could either enable searching by author or word content (In this case, I could either search by one or more words/Phrase). I chose, to implement the latter, the user can enter a phrase and I split it to get the words and search for al posts that contain at least one of those words. In order to show the results, I basically reused the view for the allposts page by changing the title parameter passed in. This was better than creating a new view for this feature as this page required the same features (link to logout, make a post, or to manage follows).