Project 2: Fritter

By Elliott Marquez

Highlights:

Signup / login general structure

- I reused this click listener structure several times within my code. It validates the form, strips the info from the form, and then sends an asynchronous Ajax call to the server. The server then parses the incoming JSON object, checks if the username and password match, and then passes back a JSON object that the client parses and uses to determine whether or not to allow the user through, or show an error message. Additionally, I have empty callbacks on some success : function() callbacks in the Ajax calls. This gives makes the code more modular for future feature implementations.

Edit Post

- This uses a similar structure to the signup / login asynchronous calls. A menu is dynamically generated and destroyed by the Semantic UI framework, so instead of adding a click listener using jQuery, I had to call a function when clicked in the HTML. Knowing this exists, I am able to find the clicked element using jQuery. Upon click I use an asynchronous animation function offered by Semantic UI. Since it is asynchronous, I have to put the rest of my code in a callback. I remove the inner HTML and replace it with another form that is used to edit the code. I then follow a similar structure to the login and signup structure with an Ajax call to the server requesting a post edit.

Post Ordering

- I reverse the traversal of the list of posts so that they are new to old rather than old to new. The reason that I bring this up as a highlight is because it was a very simple fix to the sorting issue. I do not waste time sorting it with a built-in method that would have taken O(nlogn) calculation compared to the O(n) time it would take to iterate through the order of IDs. This is better for scaling.

Design Challenges:

How to Handle Signup / Login Forms

- Options
  - Use a redirect and load an appropriate homepage accompanied by a session cookie upon form submission
  - Use ajax to submit and redirect using javascript accompanied by a session cookie
- Used
  - I ended up using ajax to submit rather than going through the routers. This allowed for dynamic responses to the user input. Such as telling them what was wrong. It also prevents the user

from having to reload the entire page when signing up / logging in and thus losing their previously inputted information.

What to Store on a Session Cookie

- Options
  - Store all user data sans posts
  - Store some hash
  - Store a username
- Used
  - I used a “hash”. Storing all of the data would have been convenient for me but it was too vulnerable and easy to steal information. I did not use an encrypted cookie for the sake of implementation time. Storing a username also would be easy to change since there was no encryption in the cookie. I stored a “hash” with the hash being the account database object id. This is not entirely safe because simply increasing the hash by one number would log you in as another user. I still chose this because it would be difficult to figure out the first hash and it also made it difficult to find a specific person’s hash.

How to Handle Posting

- Options
  - Have a posting page where you submit your post
  - Have a posting form within the feed
- Used
  - I used a posting form within the feed. I felt it gave a more-intuitive interface like Facebook and Twitter. It would have taken the same implementation time either way since I end up reloading the page after every submit despite using an ajax call. I did this because I did not want to have a post.ejs and the same information in jQuery.html() call. Updating one would have required updating the other and that would have caused more issues with modularity.

Schemæ:

In mongoose I created two models with two schemæ. The first model is an Account model.

Account:

```
{
    firstName : String
    lastName  : String
    name      : String
    userName  : String
    password  : String
}
```

The reasoning is as follows:

firstName
This is a String because first names can vary and they are not typically a single type. String encapsulates this. I also chose a first name field for future implementation and modularity. It is more appealing to the user for the site to be “human” and talk as a human would by referring to them by their first name.

lastName
This is a String for the same reason as *firstName*, but the reason as to why I included it was for the sake of having a full-name field.

name
This is the full name of the person. It is a String for the same reason as *firstName* and *lastName*, but is included for the sake of having a full name variable to refer to rather than having to reconstruct the name from the two variables each time. This is not good for scalability and can be removed for that argument.

username
This is a String to allow the users to have more personalization and allowing them an easier way to remember their username rather than having them memorize random numbers. I chose to include this because this is what is used to authenticate accounts. It also is unique across the database.

password
This is a String and is included for the same reason as *username.* The only difference is that it is not unique across the database.

Post:

```
{
    username  : String
    name      : String
    title     : String
    body      : String
    timestamp : String
}
```

username
This is included this because future implementation would allow the user to filter posts by user, and this would be the first step in doing so.

name
This is included because it is included as information on each post such as “*name* posted:”

title
This is included because each Fritt in my interpretation of the website has a title. It is a string because it is a message that the user types in and a string makes most sense.

body
This is a String and is included for the same reasons as *title*.

timestamp
This is a string because that is the easiest implementation given that the Moment module returns times in String format. I included this because each Fritt needs a timestamp for context.