fritter™ Design Documentation

fritter-seropian.rhcloud.com ● github.com/6170-fa14/seropian_proj2

Lily Seropian ● seropian@mit.edu

Grading Directions: Phase 2

Highlights

- I'm happy I fixed the thing where users could edit other user's tweets because I forgot about a UI feature I had...not proud about that, but I'm proud that I fixed it in the front end AND the back end for that double validation.
- I'm proud that adding new features did not require changing the structure of the data model (I just had to add new attributes) because that showed that I made good decisions in phase 1 and planned ahead adequately.

Help Wanted

- Data model diagram (in this doc): is this correct? The information I'm trying to get across is:
    - Each user can follow an arbitrary number of users
    - Each user can favorite an arbitrary number of tweets
    - Each tweet has exactly one immutable creator
    - A user can have an arbitrary number of tweets

Grading Directions: Phase 1

Highlights

- I'm really happy with how both my front end and schema provide input validation. For example, the new tweet text area has maxlength set to 140, preventing a user from entering in a tweet longer than 140 characters. If the user were to send a POST request directly to my server with a tweet longer than 140 characters, the mongoose validation would send back an error because the tweet's content is defined on the schema to be 140 characters or less. This double validation supplies very quick error detection on the front end (because the user doesn't have to wait an RTT to get an error message from the server) and malicious user protection on the backend (by validating before allowing data to enter the database).
- I'm proud of the error-handling in the backend. No matter what the user does, or what database error is encountered, the app should still function, and display an error message to the user (if applicable).
- I'm happy with the structure of the application. It follows a pretty standard node-express directory layout, with modular code so no single file is very long.

Help Wanted

- What is a session secret, how do I choose a good one, and is it bad form to have it hardcoded in the source code? server.js line 12
- Is there a more standard way to determine if your server is running on localhost vs. openshift? server.js line 18
- How can I avoid putting my database password in my source code? server.js line 19
- How can I separate my routes into different files in a cleaner way? I'm currently doing weird things with module.exports to get all the routes on the same object. index.js line 93, tweets.js line 9

Design

User Authentication

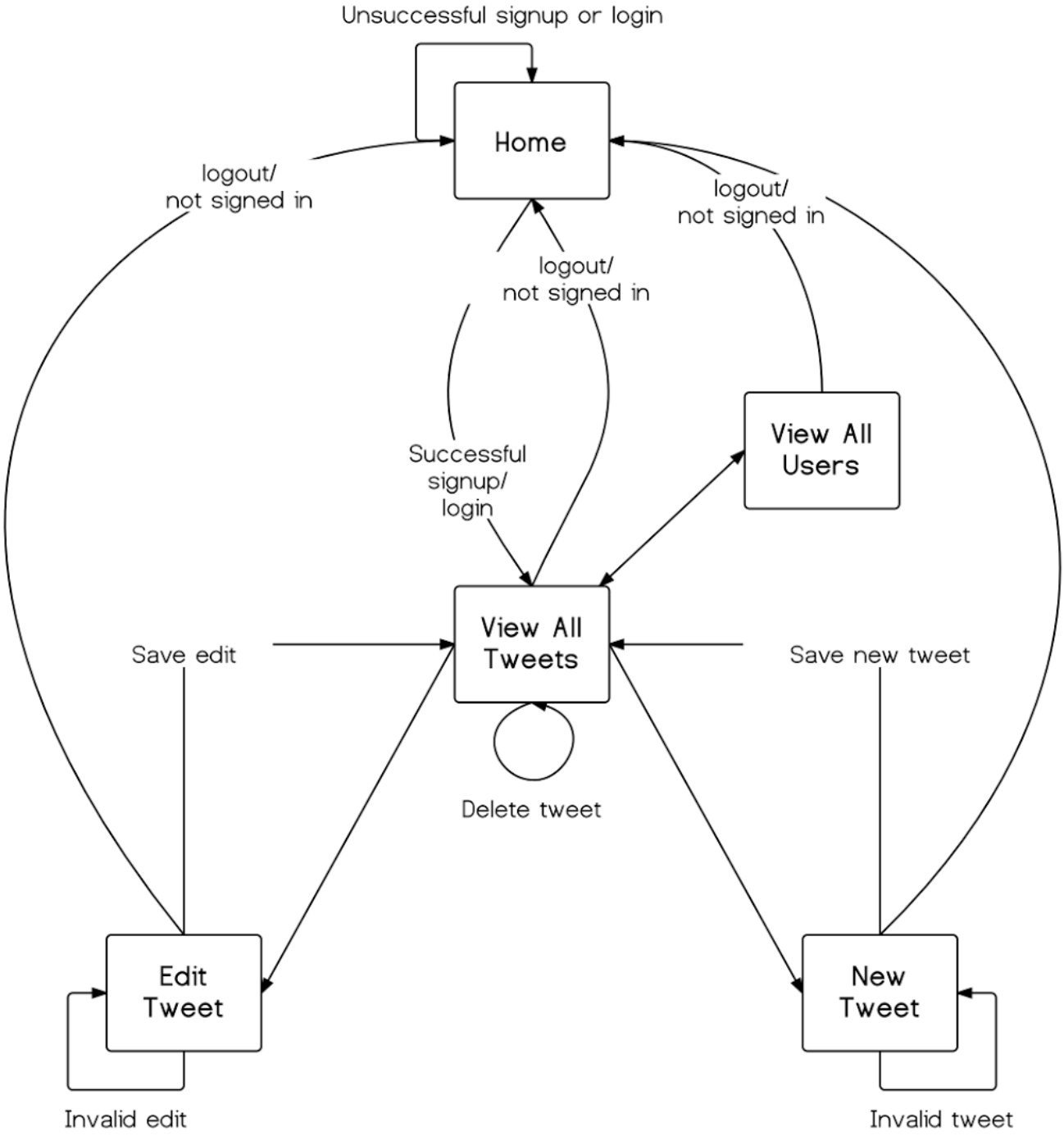
Username and password is a pretty standard route to take for this type of web application. Users will be asked to log in once at the beginning of a session. The server will remember that the user is logged in by using express-sessions. If a user tries to access a page (other than the home page) without being logged in, they are redirected to the home page. When a user is viewing tweets, the edit and delete options are only part of the DOM if the tweet they modify is owned by the user currently logged in, preventing the user from modifying or deleting others' tweets. When the user makes a tweet, the creator is set behind the scenes based on the user logged into the current session, so users cannot create tweets that seem to be coming from a different account.

An alternative to username and password is MIT certificates. This would be slick for MIT students, but since it is a MIT-only phenomenon, my learning how to do that would not be as useful for real world application development and limits the use of the app to computers with certificates. It would also prevent me from having multiple test accounts.

Another option would be to use email addresses instead of usernames. This could come with the benefit of password recovery, at the cost of having to add an extra verification step when creating an account. For the ease of both my own testing and for the sake of the TA's, I have decided not to go this route.

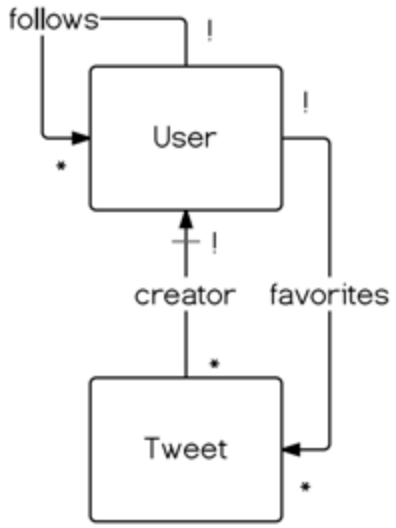
Routing/Site Layout

When a user first visits the site, they go to the home page, where they must either login or create an account. Once this is complete, the user has access to tweet-specific pages, including viewing all tweets, editing their own tweets, and creating tweets. If the user attempts to access a tweet-specific page when not logged in, they are redirected to the home page. The user may log out at any point, which also redirects them to home. The diagram below shows how the user navigates between pages and the actions the user can perform on tweets.



Data Model

The two key pieces of data here are users and tweets. Users can create, update, and delete their own tweets, and read the tweets of all other users. Users can also follow other users, and favorite tweets. Please see the diagram below that details the relationships between users and tweets.



We use the lovely Mongoose to interact with our data. Mongoose has several advantages over just MongoDB: Mongoose provides schema validation, which helps keeps our database in a consistent state, and provides workarounds for MongoDB's lack of joins.

There are two Mongoose models: User and Tweet. Users have a username, password, list of users they follow, and list of tweets they have favorited. Tweets have the content of the tweet, a reference to the User that created the tweet, and the number of times that tweet has been favorited. This is a tweet-centric approach, in that when all references are followed, the root of the resulting nested document is a Tweet.

Another option would be to use a User-centered approach, where a user, in addition to the properties outlined above, would have a list of references to all of their own Tweets (and a Tweet would not have a User reference). However, we more often need to know the creator of a Tweet than the tweets by a specific User, making the Tweet-centric model more practical. In the event that we need to get a list of the User's tweets, we could either find all of the tweets of a given user via a find query on the Tweet model, or we could implement a circular model.

A circular model would have each User have a list of references to their tweets, *and* each Tweet having a reference to its creator. This model, while allowing quick lookups of both all of a User's Tweets and a specific Tweet's User, requires extra effort whenever creating a new User or Tweet to add all the references. Because read performance is not a key factor here, this model is inferior to the tweet-centric one.