FRITTER DESIGN
FOR A MORE POTATO-EY TOMORROW

EVAN LYNCH

1. Terminology

Since this is *Fritter* and not *Twitter*, the *Tweet* analog is called a *Frite*. To deliver a *Frite*, one must *Fry* it, making the deliverer a *Fryer*. Additionally, since one often pays for a potato with dollars and cents, to tag a *Frite* one would use a "$cashtag." If one found a *Frite* to their liking and wanted to *Fry* it themselves without modification, this would be called *refry*-ing.

2. Highlights

The potato spinner GIF that shows up when the UI is blocking is the clear winner. However, another highlight is the simple API. The API is good because it is fully specified in terms of inputs, outputs, and side-effects, and though limited in scope, it is designed without a particular client in mind. This made writing the REACT client-side app much easier.

The "features" I chose to implement beyond the basic requirements of a Twitter-like app are:

(1) Following – the user may add other users to their following list; then they can see only what their friends are frying in a special aggregated view.
(2) Refrying – the user may, with a single click, quote and share the frite of another user.
(3) Search - the user may choose to see only frites that include selected $cashtags or only frites from selected other users denoted in the search bar as @username. Message body search is not implemented.

3. Design Challenges

The main design decisions to make were the following.

(1) What schema and structure to select for the data model
(2) How much view rendering to do client-side / what view end-points to support
(3) What interface to expose through an API to the client
(4) How to model data on the client side
(5) How to render views on the client side

Now, I will describe the decision I made for each and why.

3.1. **Data Model.** The data model is presented in Figure 1. I chose to use the mongoose wrapper for MongoDB. Out of the data model come two obvious structures which have been represented with the following schema defined in model.js

Date: October 1, 2014.

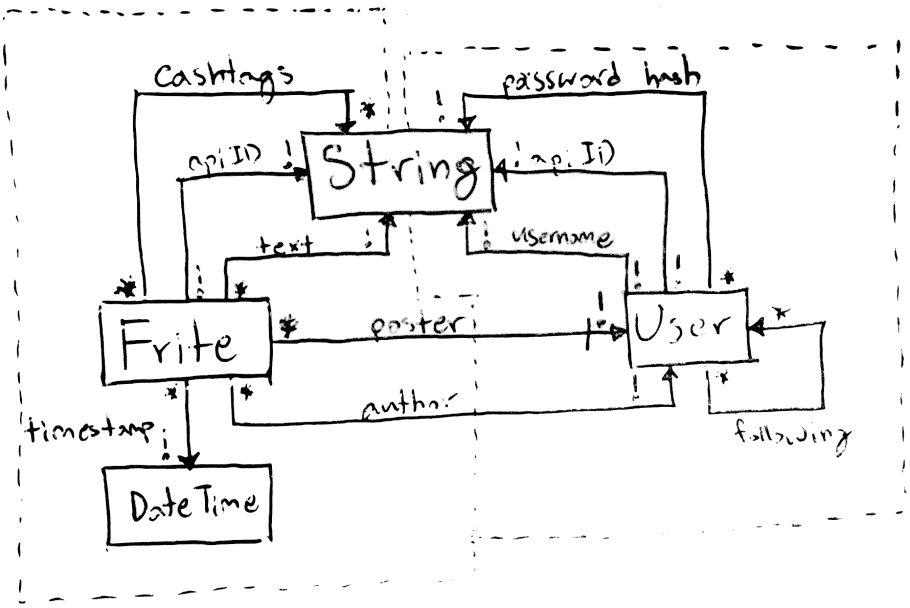

FIGURE 1. Data Model

```
User:
    username: String
    hash: String
    apiID: String of the form u_??????????
    following: Array of (ObjectID referencsing users collection)

Frite:
    text: String
    timestamp: Date (serialized by mongoose)
    apiID: String of the form fr_??????????
    cashtags: Array of Strings
    poster: Number (ObjectID referencing users collection)
    author: Number (ObjectID referencing users collection)
```

The data model was chosen to facilitate the functions of the application. First, each Frite carries a reference to a User document rather than a nested user to avoid excesive repetition and to make it easier to make modifications the the user part of the data model.

Second, both an "author" and a "poster" reference are stored for by the each Frite to facilitate the 'refry' feature. When a Frite is refried the author of the new Frite is set to the original Frite's author. If it is later edited, the new Frite's author is changed to the poster.

Third, the User carries an array of references to other users implementing a list to be followed. This relation was chosen (rather than the reverse), because the primary function of following is to allow the user to quickly see the posts of the users they are following (rather than to see who is following them, although this could be desired as well).

Finally, the data model was selected so that only one level of mongoose.populate would be required for the functionality of the application.

3.2. **Views.** I opted to do the vast majority of view rendering on the client side. The server only responds with HTML at two end-points "/" and "/login" routed in views.js. The "/login" end-point allows a user to log in regardless of what username is set in the session while the "/" end-point goes to the main site unless there is no username set in the session in which case it goes to the login page. The HTML response on both of these do little more than load the client-side javascript and set globally scoped variable "bootstrap" in which it passes the username from the session. For the small amount of templating this required, I used the Jade template engine.

3.3. **API.** The API was designed as a "public" API taking the perspective that any API that the server presents as an interface is "public" to begin with in the sense that anyone can send it requests not just the designated client app. Therefore, it sends and receives JSON and attempts to return helpful error messages and status codes when requests are formatted wrong or fail for other reasons. The interface is well documented in api.js.

3.4. **Client-side.** I chose to use Backbone.js for storing a model of responses from the API in the client app. I use little of Backbone's functionality otherwise, but the Backbone model was a convenient choice for parsing performing API fetches and saving them in a coherent manner and keeping them in sorted order.

The client implements the view with REACT.js which is a framework for writing reactive client-side applications in a declarative fashion. The structure of a view in REACT is a hierarchy of components each which can carry and update state as well as pass down properties to children. The UI is declared to appear a certain way based on the state and properties that the component has. My REACT code has the following structure.