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.

3. Design Challenges

The main design decisions to make were the following.

(1) What schema 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.**

I chose to use the mongoose wrapper for MongoDB. The data model consists of two schema defined in model.js.

```
User:
    username: String
    hash: String
    apiID: String of the form u_??????????

Frite:
    text: String
    timestamp: Date (serialized by mongoose)
    apiID: String of the form fr_??????????
    cashtags: Array of Strings
```

Date: October 1, 2014.

```
user: Number (ObjectID referencing users collection) --- ‘‘poster’’
refry: Number (ObjectID referencing users collection) --- ‘‘author’’
```

Much of these schema are obvious; however, two decisions stand out. First, each Frite carries a reference to a User document rather than a nested user to allow users to change their usernames, passwords, etc. in the future.

Second, when a Frite is refried (“retweeted”), a new Frite is created with the original Frite's text and the refry field set to the original Frite's user. Another option would have been to store a separate kind of object instead of a full Frite which would references the original Frite. The choice to save only the original Frite's user was made to simplify the data model (no recursive structure) and to avoid confusing situatiosn like when the original Frite is edited (should the refrys change to?), etc.

However, Frites are mutable and we don't want to allow a user to refry a Frite and then change the text (essentially attributing their own words to someone else). Therefore, we have to maintain the invariant that the refry field of a Frite is only set to a user if the text of the Frite corresponds to the text of another Frite which that user at one point made. This is a hard invariant to verify, but easy to maintain and was kept in mind when implementing the API.

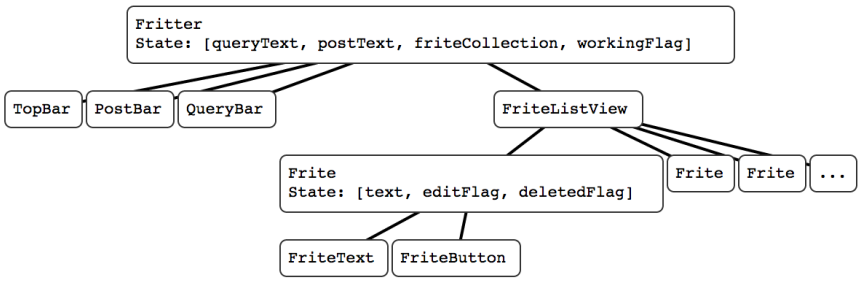
Finally, the data model was selected so that only one mongoose.populate call would need to be made to get a complete picture of a Frite.

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.


Fritter
State: [queryText, postText, friteCollection, workingFlag]
TopBar
PostBar
QueryBar
FriteListView
Frite
State: [text, editFlag, deletedFlag]
Frite
Frite
...
FriteText
FriteButton