Fritter - Part 2 Design

Harini Kannan

October 9, 2014

1 Grading directions

To view the app, go to http://fritter2-hkannan.rhcloud.com/. Note that this is different from the URL from Part 1.

1.1 Highlights

1.1.1 Feature 1 - Following Users

To implement following users, I chose to use the following data model (described later in the design section): for each user, I defined a list of followers and a list of followed users. In order to keep all of my logic in my model, I defined two functions : $follow\_user$ and $follow\_self$.

`https://github.com/6170-fa14/hkannan_proj2/blob/master/models/user-model.js#L14-L23`
`https://github.com/6170-fa14/hkannan_proj2/blob/master/models/user-model.js#L27-L35`

Instead of writing in a lot of model logic in my controller, I could now just make calls to my model to achieve the same functionality. For example, just one line of code is needed to change the model when the user presses the "Follow" button:

`https://github.com/6170-fa14/hkannan_proj2/blob/master/routes/users.js#L96`

Because a user follows himself or herself by default, so when a user is created, the function $follow\_self$ is called:

`https://github.com/6170-fa14/hkannan_proj2/blob/master/routes/users.js#L125`

1.1.2 Feature 2 - Favoriting tweets

To implement liking tweets, I had two lists: a list of liked tweets for each user, and a list of users who like a tweet, for each tweet. Again, to keep my data logic in my model, I defined the following function to update both lists in the model at once:

`https://github.com/6170-fa14/hkannan_proj2/blob/master/models/post-model.js#L18-L30`

This function is then used in one line here:

`https://github.com/6170-fa14/hkannan_proj2/blob/master/routes/posts.js#L140`

1.1.3 News Feed

By adding a method to my model, I was able to generate the news feed with one line of code:

`https://github.com/6170-fa14/hkannan_proj2/blob/master/models/post-model.js#L13-L15`

This function is then used here:

`https://github.com/6170-fa14/hkannan_proj2/blob/master/routes/posts.js#L29`

1.1.4 Organization of models

I had a central file called mongoose.js that set up the connection to mongoose. I then had individual files for each model (user-model.js and post-model.js) that required mongoose.js. Then, any code that requires a User simply requires user-model.js (and same for Post). I thought this was the most modular way to organize my code. For example, here is a line from one of my controllers, users.js:

`https://github.com/6170-fa14/hkannan_proj2/blob/master/routes/users.js#L6`

2 Design Challenges

2.1 Following a user

2.1.1 Design Challenge 1 - Defining the feature of following users

There are many different ways that following other users could be implemented. For example, if a user's tweets are published to the entire social network, following a user could mean simply consolidating all of their tweets into one place. This sort of implementation would be more of an online forum or bulletin board, where anyone can see anyone else's posts.

Especially as we are in an age of privacy concerns, I decided that users should only be able to publish tweets to their followers, which necessarily implies that they will only be able to see tweets by people they are following. Therefore, in my app, you can view and favorite tweets from a user only if you are currently following him or her. In other words, because my app is a social network, you must create an account and follow friends in order for Fritter to be of any use.

2.1.2 Design Challenge 2 - Schema choices for following users

A User in my schema has two lists: Followers, and Following. The other major schema choice I was considering was the tuple model, where I would have a entry for each "following" relation. Each entry would store a tuple (User A, User B), where User A follows User B. While my chosen schema does have redundancy, I chose it after considering the following analysis:

- Pros

  1. Writing code for lookups is simpler and more intuitive
     My app's main feature is a news feed that displays tweets from all users that the logged in user is currently following. Because each user has a list storing the users they are following, it makes the query to populate this newsfeed extremely simple:

     On the other hand, using the tuple model would have led to a longer, more convoluted query. Also, my data model allows my app to easily extend to other features such as displaying a list of followers. People are often interested in who is following them, so creating this view would be extremely simple with my data model, which stores a list of followers for each user.

  2. Queries become more efficient
     In order to have a news feed that displays tweets from all currently following users, the tuple model would require joins. When the social network becomes bigger, these joins will become more expensive. However, under my schema, no joins are required to display the news feed, which could be better for performance.

- Cons

  1. Maintaining an invariant
     The main issue I had with my chosen schema was that now I needed to maintain an invariant at all times. When User A follows User B, User B should be added to both User A's "Following" list, and User A should be added to User B's "Followers" list. If this invariant is not maintained,

there could be many consequences, including tweets from an intended follower not showing up on a News Feed. The tuple model does not have this issue because it is updated only once when a user follows another user.

   To maintain this invariant in the cleanest possible way, I defined a method in my model to update both lists at once. This way, whenever my app needs to follow a user, it simply calls the method in my model. Because this invariant is maintained (and therefore possibly violated) in exactly one place, I was confident that defining my model method would make it easy to detect any possible errors in the invariant if extending the app in the future.

2. Redundancy
   My schema also has the issue of redundancy because storing both followers and following users duplicates information. On the other hand, the tuple model simply has one entry for each "following" relation. However, I decided to make this tradeoff anyway because my application is small enough that duplicating data will not lead to a cluttered database or poor performance. Moreover, as the social network scales, performing frequent joins under the tuple model will be more expensive that simply storing more data in the database under my chosen schema.

2.2 Favoriting tweets

2.2.1 Design Challenge 3 - Defining the feature of liking a tweet

I implemented this feature exactly how Twitter does it. When a user likes a tweet, the number of "likes" for that tweet increases, and you can also see who has liked the tweet. It is also possible to see the list of tweets that a user has liked in the past.

I thought that this was the most natural and personal interpretation of the "like" button. Other definitions I was considering was simply displaying the number of likes on a tweet (i.e. upvoting). However, I thought users would be interested in knowing who liked their posts, as it adds a more personal feel to the social network.

2.2.2 Design Challenge 4 - Schema choice for liking a tweet

Similar to the feature of following users, I had two choices of schema. The first was a tuple model that would store tuples of each "favoriting" relation - (User, Favorited Tweet). The second choice was storing a list of favorited tweets for each user, and a list of users who favorited a tweet for each tweet.

- Pros

  1. Simpler, more intuitive code for lookups
     I needed to implement two views. The first view displays the number of likes for each tweet and who liked it. The second view is a page displaying a list of liked tweets for every user. I decided that my model would be a natural choice - to display the first view, I simply needed to query the list of users associated with each tweet. For the second view, I simply needed to query the list of tweets associated with each user.
  2. More efficient queries
     Similar to the first feature, implementing the tuple model for favoriting would require joins to display the two views described above. As discussed previously, this could cause problems if the application were to scale to millions of users. Therefore, I thought that a model using two lists was the better choice here.

- Cons

  1. Maintaining an invariant
     Choosing to use the two list data model over the tuple model for this feature would mean maintaining the following invariant: if User A likes Tweet A, User A is in Tweet A's list, and Tweet A is in User A's list. If this invariant were violated, it could cause the number of likes for a tweet to be inaccurate, or could cause a liked tweet to not show up in a user's Favorited page.

In order to deal with this, I updated both lists in model in one method: postSchema.methods.like_post(). Whenever the user needs to like a post, this method would be called. This meant that any source of invariant violation would come from this method, making the code easier to maintain and debug.

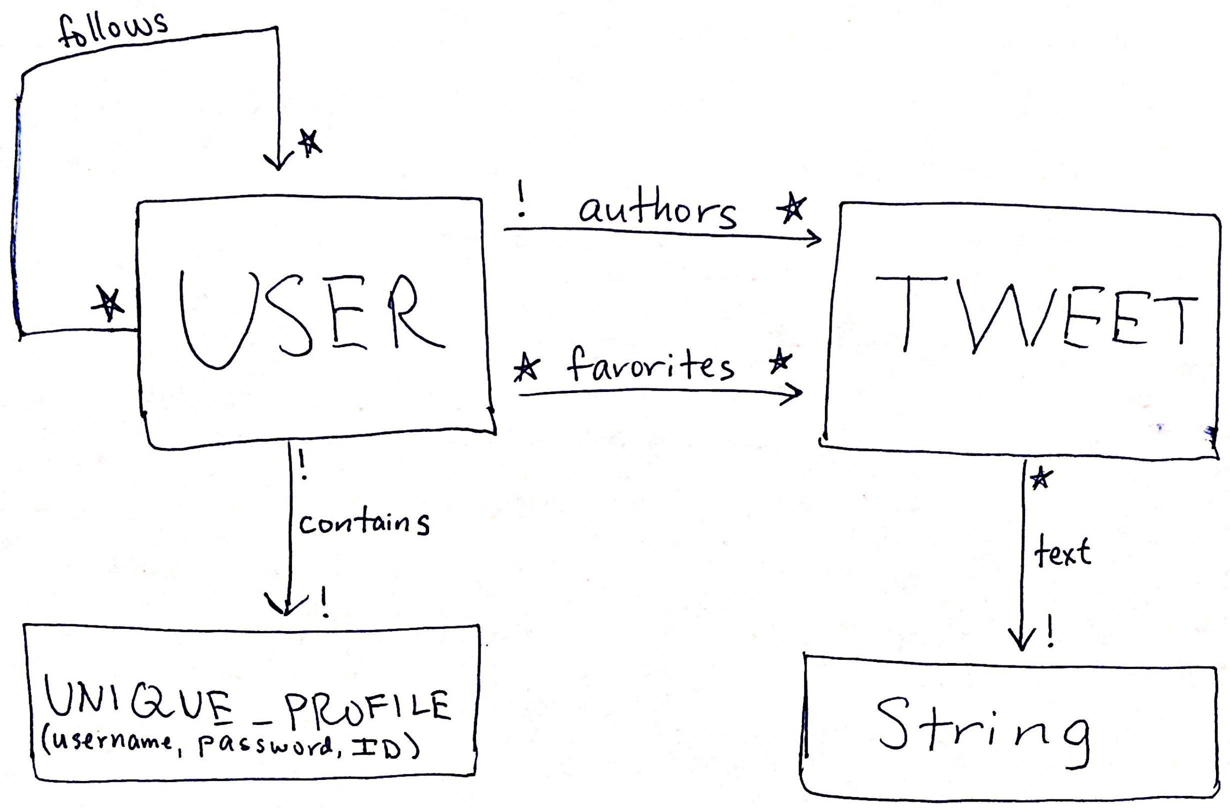
2. Redundancy
   Although the two list data model is redundant unlike the tuple model, I chose to go with the two list model for reasons described in the section about following users.

2.3 Final schema and data models

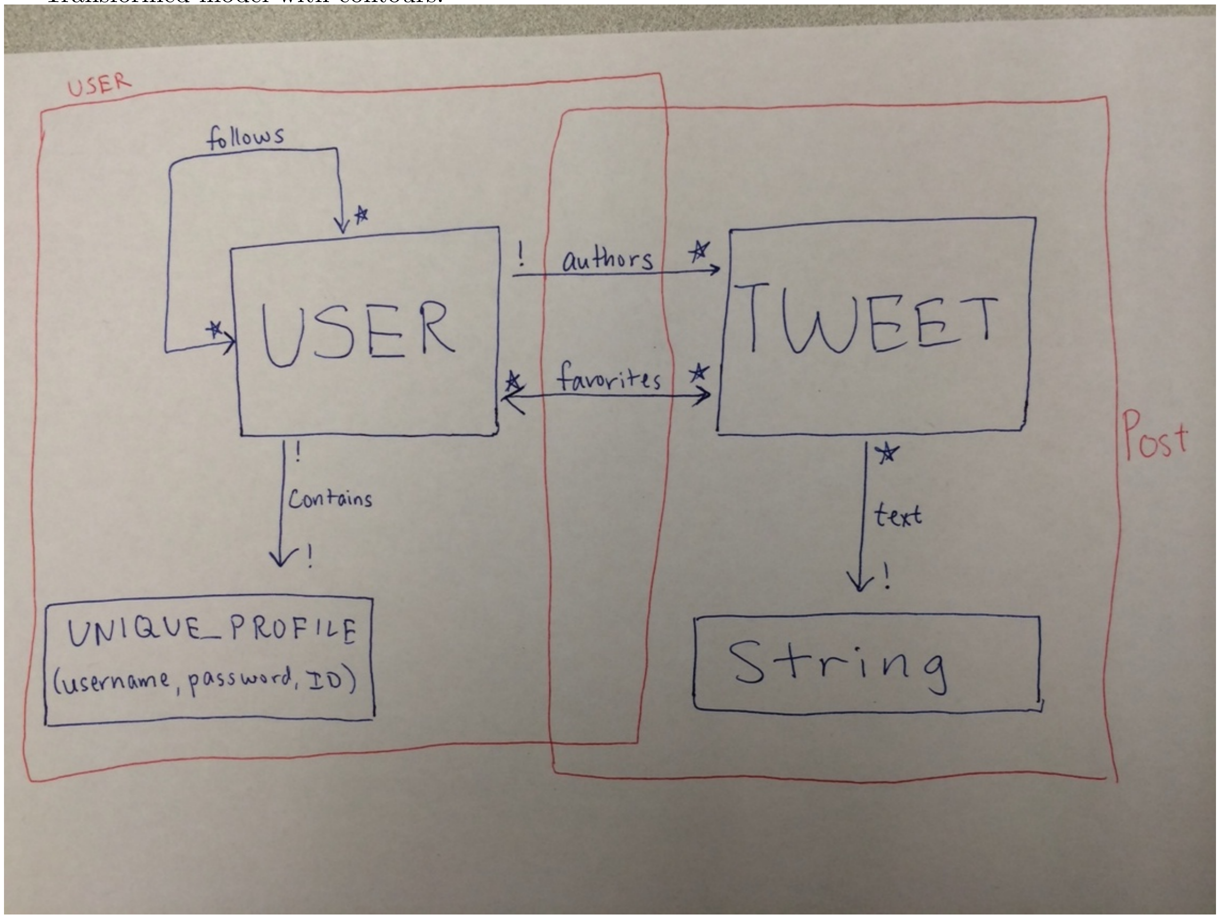
```
var userSchema = Schema({
  name    : String,
  password    : String,
  following : [{ref: 'User', type: 'ObjectId'}],
  followers : [{ref: 'User', type: 'ObjectId'}],
  liked_tweets : [{ref: 'Post', type: 'ObjectId'}],
});

var postSchema = Schema({
  _creator : { type: 'ObjectId', ref: 'User' },
  name : String,
  post    : String,
  likers : [{ref: 'User', type: 'ObjectId'}],
});
```

Data model:


follows
*
*
! authors *
USER
TWEET
* favorites *
!
contains
!
*
text
!
UNIQUE_PROFILE
(username, password, ID)
String

Transformed model with contours:


USER
follows
*
*
!
authors
*
USER
TWEET
*
favorites
*
!
Contains
!
*
text
!
Post
UNIQUE_PROFILE
(username, password, ID)
String