Projects 3 & 4: Fritter

Part 1: Due at noon on 10/05/2016

Part 2: Due at noon on 10/19/2016

Fill out submission forms! v1 Submission Form

Objectives

In this assignment, you will explore and gain hands-on experience with central concepts of web programming: client and server, HTTP requests and responses, user authentication, and server-side persistent storage. You will also use data models (part 2) and dependence diagrams (part 1 and part 2) as tools for elegantly designing your software. In the first part of the assignment, you will get a basic version of your application up and running with Node.js and the Express framework. In the second part of the assignment, you will add proper server-side persistence using MongoDB, add full user authentication, and apply your knowledge of data models to enrich your app beyond the basic features implemented in the first part.

Overview

Your challenge in this assignment is to build Fritter, a clone of a well-known social app whose primary purpose sometimes seems to be to fritter your time away. The actual purposes of Fritter include: enabling short and timely communications between users; helping users promote themselves with public postings; and letting users filter their views to focus on other users of interest. Fritter has a persistent store holding information about the users, their relationships, and the communications they post. In the version you'll produce in the first part, however, this store will be so simple that building it will require no serious contemplation of the data model. In the second part, you'll add some new features that will require a data model and a more elaborate implementation, including use of MongoDB. Only high-level specs are given. You have considerable design freedom, as long as your app is functional, and you follow the best practices discussed in class. The usual constraint is that you may not use off-the-shelf code that already provides the entire functionality or almost all of it: this would include, for example, an authentication framework for handling user accounts (such as Passport.js). (The purpose of implementing an authentication scheme yourself in this project is to ensure that you become familiar with the mechanics of creating accounts, maintaining sessions, and so on. Later on, you'll be free to use an authentication framework that does all this for you.)

Specification

Part 1

Requirements: Build a version of Fritter with which users can post and delete tweets as well as see the tweets of other users. Users should be able to enter a username, and switch to a different one. No explicit account registration or passwords are required, however. No tweets should be lost if a page is reloaded. For this portion of the assignment, you should not use a database; instead, you will just store a list of tweets in an array or other data structure and persist that structure to a file (see persist.js in your starter code).

You should document the request handlers and the public methods of any supporting modules you build in **JSdoc** style. You should also construct a dependence graph showing the dependences between all files in your project, as an aid to improving the structure of your code.

Part 2

Requirements: In Part 1, you persisted state in a file. A file is an inefficient and fault-intolerant way to store application data like Fritter's data. Therefore, in this part of the assignment, you will properly persist application data using MongoDB. This means that all application data (tweets, user accounts, etc.) should survive a server restart.

Additionally, you should supplement your Fritter implementation from Part 1 with the following features:

- Users should be able to properly register accounts with a username and password before being able to access any content. Each username should be unique, that is, an attempt to register a new account with an existing username should fail. Users should be able to log in using the username and password they specified when they registered their account. The server should properly authenticate these credentials, and use them to enforce security constraints. For instance, one user should not be able to post tweets in the name of another user, or delete another user's tweets.
- Users should be able to "retweet" other users' tweets. Retweets should be displayed with both the name of the original poster and the user who retweeted it.
- Users should be able to follow other users, and see a page of tweets and retweets by users they follow only (in addition to the page that shows all tweets by all users).

You should document your code as in the first part, and provide a dependence graph. You should also construct a data model describing (at an abstract level) the structure of the state of the application: what data items are stored and how they are related to one another.

Grading

Each part will be graded as follows:

- Functionality (45 points). Does your implementation include the required functionality? Does it execute without crashing and handle erroneous input robustly?
- Separation of concerns (15 points). Have appropriate concerns been separated? In particular, you should ensure that you have a clean separation of models (handling data storage and access), views (handling how data is presented), and controllers (handling the logic of requests and responses). Note that even in the first part, though the data model is very simple, you should make sure that it is modularized appropriately. In both parts, your controller code should only be "thick" enough to process a request, and should not include model logic.
- Code Quality (15 points). Is your code well structured into files and modules? Are the names of variables and functions well chosen? Are you using proper JavaScript idioms? Are your routes RESTful and cleanly organized? Is the code judiciously commented and appropriately documented?
- Tests (10 points). Are automated tests present? Are they intelligible and well organized? Do they provide reasonable coverage of functionality?
- Data Model and Dependence Diagram (15 points). Are your dependence diagrams drawn correctly and do they accurately reflect your application? Have you eliminated unnecessary dependencies? Is your data model drawn correctly, is it consistent with your database design, and does it accurately capture the requirements inherent in our specification of Fritter?

Together, Parts 1 and 2 will be equivalent in grade value to two programming assignments, but the first assignment will be worth only half as much as the second.

Deployment

For both parts of the assignment, you should deploy your project to a public URL where graders can access it. See the **Deployment Guide** for instructions on how to do this.

Deliverables

You will be responsible for submitting the following:

- Your repo containing your code for Fritter and automated tests for model-related code
- Your dependence graphs (one for part 1, one for part 2 = two in total)
- Your data model (for part 2)
- Your deployed website (see **Deployment Guide**)
- Your submission form

Hints

Current User You will need to track which user is active so that new tweets are associated with the appropriate user, and are subsequently displayed together with the username of the user who posted them.

JavaScript Idioms: Make sure you've used the best practices explained in lecture and recitation, including the proper use of closures, functional idioms, etc.

Separation of Concerns: The Express framework does not impose any restrictions on the way that your app is structured. That doesn't mean that all of your application logic should reside in one file! Delegate data fetching and database operations to model functions and UI rendering to views, and minimize the dependencies between these modules.

Data Model: The construction of the data model in the second part is important because it will help you separate two distinct design questions: what data there is, and how it is represented in the code. Your model will be judged on how well you've achieved this separation. Note that an 'abstract' data model does not mean a vague or incomplete one. Think of the abstract data model as representing a concrete database implemented using a graph structure of nodes and links, and make sure that structure is rich enough to support all the functionality.

Access Control: For Part 2, Make sure that your user authentication scheme supports all the required functionality. Users should be able to create accounts as well as log in. In a typical structure, there are often routes dedicated to the create-account and login forms, as well as routes that those forms use to perform the server-side authentication. Sessions can be used to remember the user that is logged in. Recall the differences between GET and POST and use them to make the routes make sense. You do not need to implement anything fancier than a basic login (e.g. email confirmation, changing passwords, retrieval of lost passwords).

Testing: Use the Mocha testing framework for Node/Express. The initial git repository for this project includes some sample code and documentation for using Mocha.

AJAX: Using AJAX may make your work easier. Think carefully about which actions you can implement by making an AJAX call to the server and updating the DOM based on the response.