P3.2 Design Analysis

Stephen Freiberg

Overview

Key Design Challenges

- How can the Posts pages and Comments pages be updated without reloading the entire page?
- How often should the pages be updated?
- How can we evaluate "rank" operations, which inherently involve O(n) users, in a way that does not create an unacceptable performance hit on the application?
- How can we credit "power users" that are large contributors of the community?
- How can we moderate content to ensure quality?

Details

Data Representation

The current implementation stands as a very similar structure to that of the Object Model. The only differences are that the PostVote and CommentVote objects replace the PostUpvote, PostDownvote, CommentUpvote, CommentDownvote objects. This decision was made because there are only two types of vote (up and down), and thus the difference between them can be represented with the Boolean “up” value. This does not reduce the amount of data storage required for votes, but does simplify the object structure and reduce the amount of code that would need to be repeated.

The PostVote and CommentVote objects are almost the same (except that they affect different objects). The decision to keep them separate was made because it logically makes more sense to keep them in separate tables rather than have an “iscomment” switch. This also allows for greater extension in the future, as Post and Comment may diverge in their representations in the future. It is essential that the PostVote and CommentVote objects exist (rather than storing integers in the Posts and Comments) so that users are unable to have their vote count multiple times on the same content.

The only invariant that is nontrivial in the code is that a user may vote Up, Down, or abstain, for each comment and post. Users may vote multiple times, but only the most recent vote will “count”. Once a User has voted (either up or down), he can not go back to the state of abstaining.

Users have a boolean "admin" field that determines whether or not they are able to delete posts and comments. These administrators are able to moderate content in the form of a hard delete. Deleting a post also deletes all comments on that post, and all votes associated. Likewise, deleting a comment deletes the votes associated with it.

Key Design Decisions

- Short polling. In order to update Post and Comment pages without requiring a refresh, the pages request fresh content from the server every 8 seconds. This time period is a compromise between refreshing every second or less, which would give the user the most responsiveness but would also create immense load on the server, and refreshing every minute or more, which would decrease server load at the cost of greater latency between postings and viewings for the user.
- When the user submits new content, a poll to the server is triggered. This allows the user to not notice any extra latent time between content submission and actual viewing of the material.
- There are only two levels of content posted on Red-Dot: Posts and Comments. Each Comment belongs directly to a Post, and is in reply to that Post.
- Anonymous users that do not have accounts may not contribute content. This is similar to the Reddit model; if users want to disassociate themselves from an account for a submission, they can create a novelty account to do so. This small, but existent, barrier increases the likelihood of repeat contributions and decreases the likelihood of one-off contributors.
- Administrators are able to delete content. This prevents spam or inappropriate content from accruing on Red-Dot. Power users are rewarded by notoriety at the top of a "Top Users" list.
- Rank operations that take O(n) operations for *n* users are cached. This means that the rank data can become stale if the cache is not updated frequently enough, but also allows the cost of a rank operation to be amortized over multiple reads. The current implementation, because scale is not a problem, refreshes the cache at every request. If the overhead of this implementation were to be problematic because of scale demands, the ability to change the "staleness" measure of cache data is present in the codebase.

Rejected Designs

- Comment and Post being one object type, with a boolean "is_post" to represent the difference. Similarly, this rejected model would also have CommentVote and PostVote in one object type. This design was rejected because of the limited future extensibility that it offers. If Comment and Post were always treated as the same type of object, there would be no ability for their roles to diverge in the future.
- Backbone.js. Backbone allows MVC type language in javascript. A considered design included the use of Backbone to mimic the RoR models on the client side, and request changes from the server using JSON. This design was rejected because implementation required duplicating much of the functionality provided already by Rails, and also required custom implementations of forms, security, etc, that would have to be kept in sync with the RoR application.
- Nested comments. A possible design allowed users to comment on other comments, creating a nested comment interface (similar to Reddit). This design was rejected because allowing nested comments can create tangential and unrelated points that are not salient to the original post.

Design Reviews (one of Stephen Freiberg, one of Joe Henke)

Stephen Freiberg assessment of Stephen Freiberg

Summary assessment from user's perspective
The ease of use for the user is good here. The site mirrors the user interactions from popular sites like Reddit and Digg, and the user is familiar with the semantics of Post, Comment, Upvote/Downvote. The interface is not excessively pretty (not too many rounded corners on those divs…), but the desired functionality is present. The fact that the user can acquire karma may serve to create a dedicated user base.

Summary assessment from developer's perspective
The most challenging and interesting design problem for this project was the asynchronous update. Asynch update was achieved through short polling of the server (8 seconds) to get the new list of comments, posts, users, etc. For ease of implementation, the entire table was returned to the client rather than just the new or updated data. While this presents a long-term stress on the server, for such a small reference implementation a more sophisticated update strategy was not required.

Most and least successful decisions
Most successful design decision was the object model, which clarified the relationships between posts, comments, and votes on each item. This allowed for an easy implementation that was clear both through the code and through the object relationships. The least successful decision was the short polling, which has the potential to DDOS the server and is not sustainable for the future.

Analysis of design faults in terms of design principles
While the code is extensible and modular, it is not scalable. While this is a problem that is not present yet, it would present more of an issue with growth. Not only would having more users be a problem, but also the steady growth of a post, comment, and vote library.

Priorities for improvement
First priority is to change the short polling update mechanism to an algorithm more robust against large numbers of users. Some type of server push would be much more sustainable. Next priority is the user interface and the user experience, which are severely limited by the basic html and css currently present.

Stephen Freiberg assessment of Joe Henke

Summary assessment from user's perspective
The user is presented with a flashy interface that is responsive to updates and visually appealing. It's really pretty ☺. All desired functionality is present, from creating questions to viewing comments, creating comments, and editing posts. One question that quickly arises is:

how does the user get his questions answered? Since only the top 10 most recent posts are shown, it seems likely that questions might fall through the cracks and never be seen.

Summary assessment from developer's perspective
It is not clear how or if the question homepage is dynamically updated. When multiple accounts are viewing the homepage and one adds a new question, the other account's view does not update. There is also no remote request made as visible through Chrome's developer tools. Is there no asynch update?

Most and least successful decisions
The top 10 design was a great decision. This allows the page to be displayed and refreshed without handling large numbers of pages, and prevents hanging when the questions are reordered onscreen.

Analysis of design faults in terms of design principles
In the future it may be difficult for the browser to rerender large numbers of comments and questions with animations, but this scalability issue is not a factor yet and may not ever be if the pages are limited to 10 items each.

Priorities for improvement
First priority would be the update issue, and then the problem of users' questions being squashed (becoming invisible under load). The UI is great right now, so no updates are needed there.