DESIGN REDDIT API

Date
M T W T F S S

1 POST : (postId: str, creatorId: str, subredditId: str, createdAt: timestamp, votesCount: int, commentsCount: int, currentVote → UPVOTE / DOWNVOTE, awardsCount: int)

(i) Create (userId: str, subredditId: str, title: str, desc: str.) ⇒ Post
(ii) Get (userId: str, postId: str) ⇒ Post
(iii) Delete (userId: str, postId: str) ⇒ bool
(iv) Edit (userId: str, postId: str, title: str, desc: str.) ⇒ Post
(v) List (userId: str, subreddit: str, pageSize: int, pageToken int) ⇒ Post[], nextPageToken

2 COMMENT: (commentId: str, creatorId: str, postId: str, createdAt: timestamp, votesCount: int, content: str, parentId: str, isDeleted: bool, awardsCount: int)

to reply to a comment

(i) Create (userId: str, postId: str, parentId: str, content: str)
(ii) Edit (userId: str, commentId: str, content: str)
(iii) List (userId: str, postId: str, pgSize: int, pgToken: int)
(iv) Delete (handled by UI based on 'isDeleted' field of comment

3 VOTE : (voteId: str, creatorId: str, targetId: str, type: enum UP/DOWN)

(i) Create (userId: str, targetId: str, type: UP/DOWN)
(ii) Edit (userId, voteId, type)
(iii) Delete (userId, voteId)

4 AWARDS

Buy Award (userId: str, paymentToken: str, qty: int)
Give Award (userId: str, targetId: str)

Question 1
Q: To make sure that we're on the same page: a subreddit is an online community where users can write posts, comment on posts, upvote / downvote posts, share posts, report posts, become moderators, etc..--is this correct, and are we designing all of this functionality?

A: Yes, that's correct, but let's keep things simple and focus only on writing posts, writing comments, and upvoting / downvoting. You can forget about all of the auxiliary features like sharing, reporting, moderating, etc..

Question 2
Q: So we're really focusing on the very narrow but core aspect of a subreddit: writing posts, commenting on them, and voting on them.

A: Yes.

Question 3
Q: I'm thinking of defining the schemas for the main entities that live within a subreddit and then defining their CRUD operations -- methods like Create/Get/Edit/Delete/List<Entity> -- is this in line with what you're asking me to do?

A: Yes, and make sure to include **method signatures** -- what each method takes in and what each method returns. Also include the types of each argument.

Question 4
Q: The entities that I've identified are Posts, Comments, and Votes (upvotes and downvotes). Does this seem accurate?

A: Yes. These are the 3 core entities that you should be defining and whose APIs you're designing.

Question 5
Q: Is there any other functionality of a subreddit that we should design?

A: Yes, you should also allow people to award posts. Awards are a special currency that can be bought for real money and gifted to comments and posts. Users can buy some quantity of awards in exchange for real money, and they can give awards to posts and comments (one award per post / comment).

1. GATHERING REQUIREMENTS

As with any API design interview question, the first thing that we want to do is to gather API requirements; we need to figure out what API we're building exactly.

We're designing the core user flow of the **subreddit** functionality on Reddit. Users can write posts on subreddits, they can comment on posts, and they can upvote / downvote posts and comments.

We're going to be defining three primary entities: *Posts*, *Comments*, and *Votes*, as well as their respective CRUD operations.

We're also going to be designing an API for buying and giving awards on Reddit.

2. COMING UP WITH A PLAN

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, potentially contentious parts of our API? Why are we making certain design decisions?

The first major point of contention is whether to store votes only on Comments and Posts, and to cast votes by calling the *EditComment* and *EditPost* methods, or whether to store them as entirely separate entities--siblings of Posts and Comments, so to speak. Storing them as separate entities makes it much more straightforward to edit or remove a particular user's votes (by just calling *EditVote*, for instance), so we'll go with this approach.

We can then plan to tackle Posts, Comments, and Votes in this order, since they'll likely share some common structure.

3. POSTS

Posts will have an id, the id of their creator (i.e., the user who writes them), the id of the subreddit that they're on, a description and a title, and a timestamp of when they're created.

Posts will also have a count of their votes, comments, and awards, to be displayed on the UI. We can imagine that some backend service will be calculating or updating these numbers when some of the Comment, Vote, and Award CRUD operations are performed.

Lastly, Posts will have optional **deletedAt** and **currentVote** fields. Subreddits display posts that have been removed with a special message; we can use the **deletedAt** field to accomplish this. The **currentVote** field will be used to display to a user

whether or not they've cast a vote on a post. This field will likely be populated by the backend upon fetching Posts or when casting Votes.

- postId: *string*
- creatorId: *string*
- subredditId: *string*
- title: *string*
- description: *string*
- createdAt: *timestamp*
- votesCount: *int*
- commentsCount: *int*
- awardsCount: *int*
- deletedAt?: *timestamp*
- currentVote?: *enum UP/DOWN*

Our *CreatePost*, *EditPost*, *GetPost*, and *DeletePost* methods will be very straightforward. One thing to note, however, is that all of these operations will take in the **userId** of the user performing them; this id, which will likely contain authentication information, will be used for ACL checks to see if the user performing the operations has the necessary permission(s) to do so.

I/O For different methods API will work with :

```
CreatePost(userId: string, subredditId: string, title: string, description: string)
  => Post

EditPost(userId: string, postId: string, title: string, description: string)
  => Post

GetPost(userId: string, postId: string)
  => Post

DeletePost(userId: string, postId: string)
  => Post
```

Since we can expect to have hundreds, if not thousands, of posts on a given subreddit, our *ListPosts* method will have to be paginated. The method will take in optional **pageSize** and **pageToken** parameters and will return a list of posts of at most length **pageSize** as well as a **nextPageToken**--the token to be fed to the method to retrieve the next page of posts.

```
ListPosts(userId: string, subredditId: string, pageSize?: int, pageToken?: string)
  => (Post[], nextPageToken?)
```

4. COMMENTS

Commments will be similar to Posts. They'll have an id, the id of their creator (i.e., the user who writes them), the id of the post that they're on, a content string, and the same other fields as Posts have. The only difference is that Comments will also have an optional **parentId** pointing to the parent post or parent comment of the comment. This id will allow the Reddit UI to reconstruct Comment trees to properly display (indent) replies. The UI can also sort comments within a reply thread by their **createdAt** timestamps or by their **votesCount**.

- commentId: *string*
- creatorId: *string*
- postId: *string*
- createdAt: *timestamp*
- content: *string*
- votesCount: *int*
- awardsCount: *int*
- parentId?: *string*

- deletedAt?: *timestamp*
- currentVote?: *enum UP/DOWN*

Our CRUD operations for Comments will be very similar to those for Posts, except that the *CreateComment* method will also take in an optional **parentId** pointing to the comment that it's replying to, if relevant.

```
CreateComment(userId: string, postId: string, content: string, parentId?: string)
  => Comment

EditComment(userId: string, commentId: string, content: string)
  => Comment

GetComment(userId: string, commentId: string)
  => Comment

DeleteComment(userId: string, commentId: string)
  => Comment

ListComments(userId: string, postId: string, pageSize?: int, pageToken?: string)
  => (Comment[], nextPageToken?)
```

5. VOTES

Votes will have an id, the id of their creator (i.e., user who casts them), the id of their target (i.e., the post or comment that they're on), and a type, which will be a simple UP/DOWN enum. They could also have a **createdAt** timestamp for good measure.

- voteId: *string*
- creatorId: *string*
- targetId: *string*
- type: *enum UP/DOWN*
- createdAt: *timestamp*

Since it doesn't seem like getting a single vote or listing votes would be very useful for our feature, we'll skip designing those endpoints (though they would be straightforward).

Our *CreateVote*, *EditVote*, and *DeleteVote* methods will be simple and useful. The *CreateVote* method will be used when a user casts a new vote on a post or comment; the *EditVote* method will be used when a user has already cast a vote on a post or comment and casts the opposite vote on that same post or comment; and the *DeleteVote* method will be used when a user has already cast a vote on a post or comment and just removes that same vote.

```
CreateVote(userId: string, targetId: string, type: enum UP/DOWN)
  => Vote

EditVote(userId: string, voteId: string, type: enum UP/DOWN)
  => Vote

DeleteVote(userId: string, voteId: string)
  => Vote
```

6. AWARDS

We can define two simple endpoints to handle buying and giving awards. The endpoint to buy awards will take in a **paymentToken**, which will be a string that contains all of the necessary information to process a payment. The endpoint to give an award will take in a **targetId**, which will be the id of the post or comment that the award is being given to.

```
BuyAwards(userId: string, paymentToken: string, quantity: int)

GiveAward(userId: string, targetId: string)
```