Answer LLD Question Guide

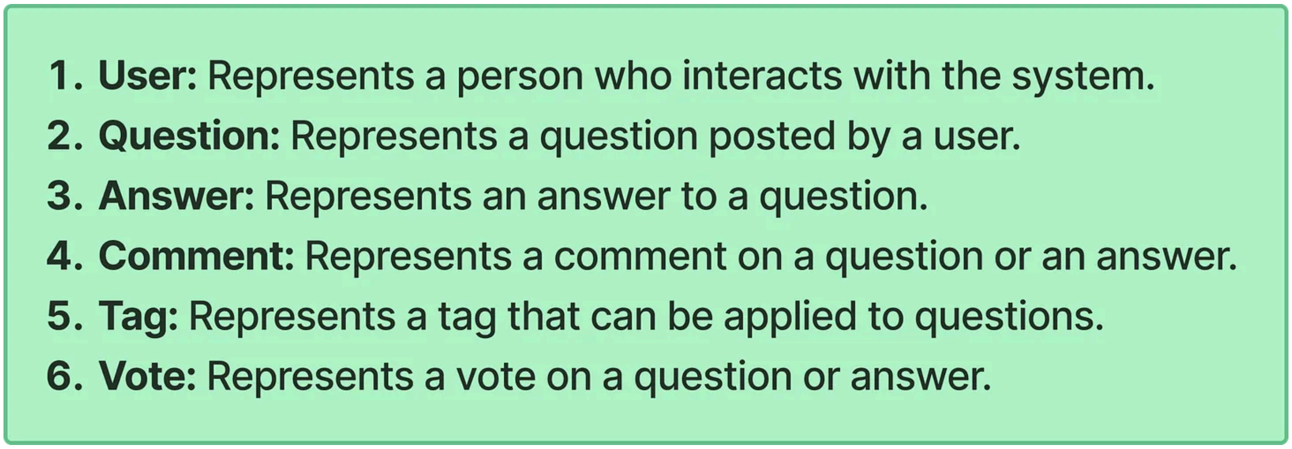
Design Stack Overflow.

Step 1: Clarify Requirements

- asking questions:
  - What are the core features we need to support?
  - any specific feature we need to prioritize?
  - Who are the primary users of the system?
  - What actions can users take?
  - any specific constraints or limitations?
  - Do we need to handle concurrency?
  - Do we need to handle errors, edge cases, exceptions, and unexpected input?
- For stack overflow system, questions:
  - Do we need comments on questions and answers?
  - should we implement tagging for questions?
  - should we design the voting system for questions and answers?
  - should we include the search functionality for questions and answers?
  - should we limit the length of questions?
- Let's say, Interviewer wants us to focus on:
  - Users can post questions, answer questions, and comment on Q &A.
  - Users can vote on Q&A.
  - Questions should have tags associated with them.
  - Users can search questions based on keywords, tags or user profiles.
  - system should assign reputation score to users based on their activity and quality of their contributions.

Step 2: Identify Entities

- for stack overflow, different entities we can have :



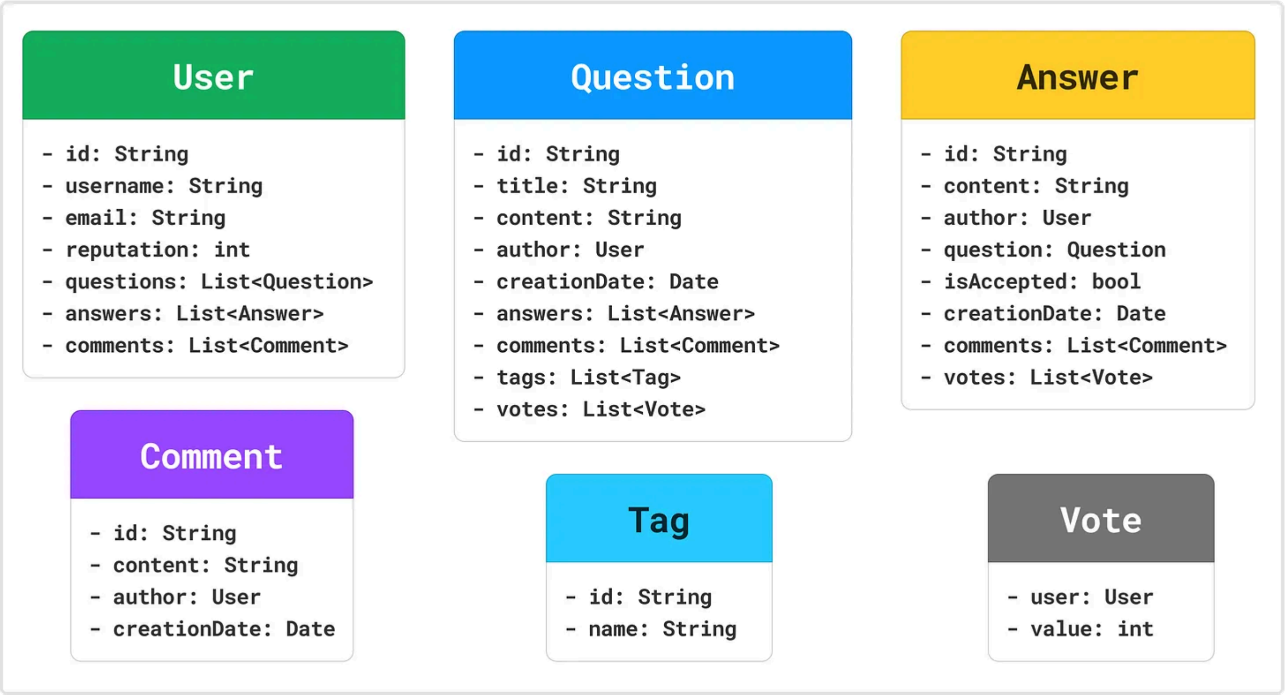
1. **User:** Represents a person who interacts with the system.
2. **Question:** Represents a question posted by a user.
3. **Answer:** Represents an answer to a question.
4. **Comment:** Represents a comment on a question or an answer.
5. **Tag:** Represents a tag that can be applied to questions.
6. **Vote:** Represents a vote on a question or answer.

Step 3: Class Design

- after identifying entities, next step is to design classes, Enums and interfaces that will represent the entities in the system.

Step 3.1: Define classes and relationships

- come up with attributes that we want to have in the entities (classes).
- draw UML diagram to illustrate the relationships between classes.



```
User
- id: String
- username: String
- email: String
- reputation: int
- questions: List<Question>
- answers: List<Answer>
- comments: List<Comment>
```

```
Question
- id: String
- title: String
- content: String
- author: User
- creationDate: Date
- answers: List<Answer>
- comments: List<Comment>
- tags: List<Tag>
- votes: List<Vote>
```

```
Answer
- id: String
- content: String
- author: User
- question: Question
- isAccepted: bool
- creationDate: Date
- comments: List<Comment>
- votes: List<Vote>
```

```
Comment
- id: String
- content: String
- author: User
- creationDate: Date
```

```
Tag
- id: String
- name: String
```

```
Vote
- user: User
- value: int
```

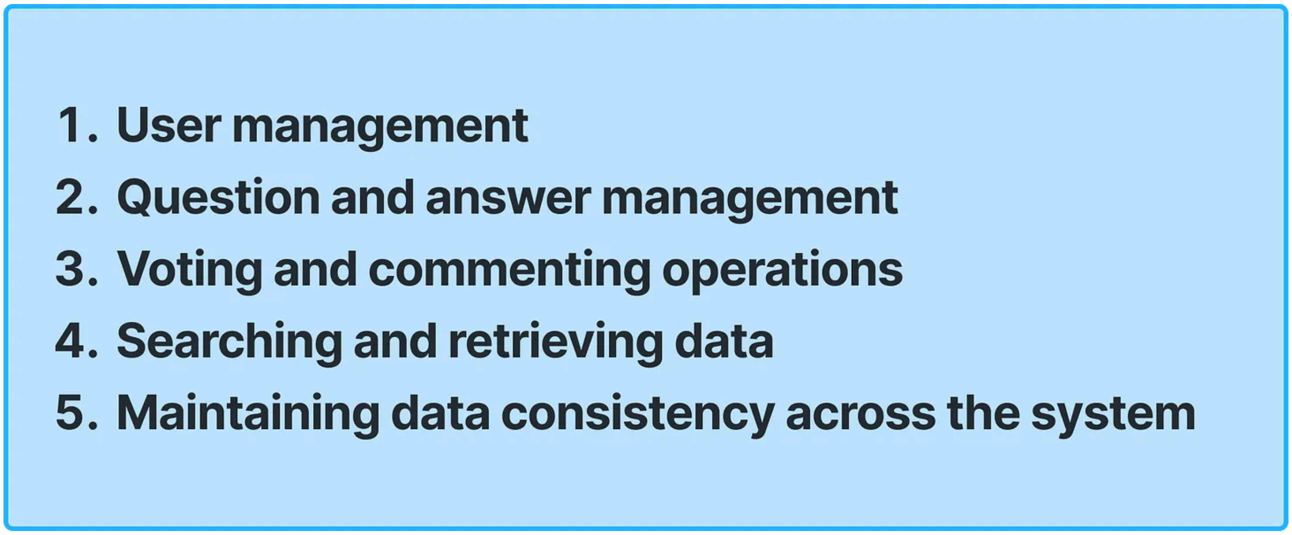
Step 3.2: Define Interfaces and core methods

- since both Question and Answer classes need to support comments and votes , we can define interface for these feature.
- Interfaces we can have in our design:

    - **Commentable**: Defines contract for objects that can receive comments (eg. Que, Ans)
        - `addComment(comment)`
        - `getComments()`
    - **Votable**: Defines contract for objects that can be voted on.
        - `Vote()`
        - `getVoteCount()`
- Each class need to have methods for the tasks it can perform.
- `User` Class:
    - `askQuestion(title,content,tags)`
    - `answerQuestion(question,content)`
    - `addComment(commentable,comment)`
    - `updateReputation(value)`: updates user's reputation score.
- `Question` class:
    - `addAnswer(answer)`
    - `addComment(comment)`
    - `vote(user,value)`
    - `addTag(user,value)`
- `Answer` class:
    - `addComment(comment)`
    - `vote(user,value)`
    - `markAsAccepted()` marks this answer as accepted.

Step 3.3: Define a central class

- we don't want to manipulate classes directly from outside.
- so, we need a central class that provides a unified interface for interacting with system.
- this simplifies API and makes it easier to use and understand the system.
- `StackOverflow` central class, as central coordinator for entire system.
- manages creation, retrieval, interaction of all major components.

- 

1. User management
2. Question and answer management
3. Voting and commenting operations
4. Searching and retrieving data
5. Maintaining data consistency across the system

- `createUser(userName, email)`
- `askQuestion(user, title, content, tags)`
- `answerQuestion(user,question,content)`
- `addComment(user,commentable,content)` allows user to add a comment on an existing question/answer.
- `voteQuestion(user,question,value)`
- `voteAnswer(user,answer,value)`
- `acceptAnswer(answer)`
- `searchQuestions(query)`
- `getQuestionByUser(user)`

Step 4: Implementation

- after defining class structure, start implementing full solution.

Step 4.1: Follow good coding practices

- favor composition over inheritance to promote flexibility and avoid tight coupling.
- avoid duplicate code.
- use interface to define contracts and enable loose coupling between components.
- only implement what is required.
- strive for modularity and separation of concerns.

Step 4.2: Implement necessary methods

- we might not have enough time to implement all the methods.
- check with interviewer to understand which methods are important for the interview.
- if the expectation is to demo and test the code, create a separate demo class like `StackOverflowDemo`.

Step 4.3: Address concurrency

- If system servers multiple users simultaneously, we may need to handle race conditions and other concurrency related issues.
- check with interviewer if u need to handle concurrency in the design.
- few strategies to address **concurrency**:
  - `synchronization mechanism`
  - `atomic operation`
  - `immutable` objects where possible to eliminate risk of concurrent modification.
  - `Thread safe` data structure that handle synchronization internally.
- for stack Overflow example, here are few concurrency considerations:
  - **Voting System**: Implement atomic operations for vote counts to prevent race conditions.
  - **comment System**: Use a Thread-safe Data structure for storing and retrieving comments.
  - **User reputation**: Use Synchronization when updating user reputation to ensure consistency.

Step 5: Exception Handling

- If it is required to handle errors, edge cases, exceptions, and unexpected input.
- for stack Overflow:
  - What if user tries to vote on their own question/answer?
  - What if a user tries to vote multiple times on the same content?
  - What if a user posts a question with empty title or content?
  - Can the user reputation go negative?

> It’s always a good idea to check with the interviewer on what all is expected from the design.