Low-Level Design Document

Introduction

This document provides a low-level design (LLD) for a discussion application. The application includes user management, discussions, comments, likes, follows, and view counts. It uses a microservice architecture implemented using Spring Boot and MySQL.

Components

1. **User Service**
2. **Discussion Service**
3. **Comment Service**
4. **Like Service**
5. **Follow Service**
6. **View Count Service**
7. **Authentication Service**

Each service is designed as a separate microservice with its own database tables and APIs.

Detailed Description of Each Component

1. User Service

Responsibilities:

- Manage user accounts (CRUD operations).
- User search functionality.

APIs:

- `POST /api/users`: Create a new user.
- `PUT /api/users/{id}`: Update an existing user.
- `DELETE /api/users/{id}`: Delete a user.
- `GET /api/users`: Get a list of users.
- `GET /api/users/search?name={name}`: Search users by name.

Database Tables:

- `users`: Stores user information.

Entities:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String mobile;
    private String email;
    private String password;
    Private set<Discussions>disc

}
```

2. Discussion Service

Responsibilities:

- Manage discussions (CRUD operations).
- Filter discussions by hashtags and text.

APIs:

- `POST /api/discussions`: Create a new discussion.
- `PUT /api/discussions/{id}`: Update an existing discussion.
- `DELETE /api/discussions/{id}`: Delete a discussion.
- `GET /api/discussions/tags?hashTags={tags}`: Get discussions by hashtags.
- `GET /api/discussions/search?text={text}`: Get discussions by text.

Database Tables:

- `discussions`: Stores discussion information.
- `discussion_hashtags`: Stores the relationship between discussions and hashtags.

- `hashtags`: Stores hashtags.

Entities:

```
@Entity
public class Discussion {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long userId;
    private String text;
    private String imageUrl;
    private Timestamp createdOn;
    @ManyToMany
    @JoinTable(
        name = "discussion_hashtags",
        joinColumns = @JoinColumn(name = "discussion_id"),
        inverseJoinColumns = @JoinColumn(name = "hashtag_id")
    )
    private Set<HashTag> hashTags;
    // Getters and setters
}

@Entity
public class HashTag {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    // Getters and setters
}
```

3. Comment Service

Responsibilities:

- Manage comments (CRUD operations).

APIs:

- `POST /api/comments`: Create a new comment.
- `PUT /api/comments/{id}`: Update an existing comment.
- `DELETE /api/comments/{id}`: Delete a comment.

Database Tables:

- `comments`: Stores comment information.

Entities:

java
Copy code
```java
@Entity
public class Comment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long discussionId;
    private Long userId;
    private String text;
    private Timestamp createdOn;
    // Getters and setters
}
```

4. Like Service

Responsibilities:

- Manage likes for discussions and comments.

APIs:

- `POST /api/likes`: Create a new like.
- `DELETE /api/likes/{id}`: Delete a like.

Database Tables:

- `likes`: Stores like information.
- `comment_likes`: Stores like information for comments.

Entities:

java
Copy code
```
@Entity
public class Like {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long discussionId;
    private Long userId;
    }

@Entity
public class CommentLike {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long commentId;
    private Long userId;
    // Getters and setters
}
```

5. Follow Service

Responsibilities:

- Manage following relationships between users.

APIs:

- `POST /api/follows`: Create a new follow relationship.
- `DELETE /api/follows/{id}`: Delete a follow relationship.

Database Tables:

- `follows`: Stores follow relationship information.

Entities:

```
@Entity
public class Follow {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long followerId;
    private Long followedId;

}
```

6. View Count Service

Responsibilities:

- Track view counts for discussions.

APIs:

- `POST /api/viewcounts`: Create a new view count.
- `GET /api/viewcounts/{discussionId}`: Get view count for a discussion.

Database Tables:

- `view_counts`: Stores view count information.

Entities:

```
@Entity
public class ViewCount {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long discussionId;
    private int count;
    // Getters and setters
}
```

7. Authentication Service

Responsibilities:

- Handle user authentication (login/signup).

APIs:

- `POST /api/auth/signup`: Sign up a new user.
- `POST /api/auth/login`: Login an existing user.

Database Tables:

- Uses the `users` table from the User Service.

System Architecture Diagram

Explanation:

- **User Service**: Manages users and their related operations.
- **Discussion Service**: Handles discussion-related operations and interactions with hashtags.
- **Comment Service**: Manages comments on discussions.
- **Like Service**: Handles likes on discussions and comments.
- **Follow Service**: Manages follow relationships between users.
- **View Count Service**: Tracks view counts for discussions.
- **Authentication Service**: Manages user authentication.

Communication:

- Each service communicates with its own database.
- Services interact with each other through REST APIs.

Summary

This low-level design provides a detailed description of each component in the system, their responsibilities, APIs, database schemas, and entities. The architecture diagram illustrates how these components interact in a microservice-based infrastructure, ensuring modularity, scalability, and maintainability.