### **Design Quora/Reddit/HackerNews (A Social Network + Message Board Service)**

**REQUIREMENTS**

**FUNCTIONAL REQUIREMENTS**

1. User should be able to post questions, providing title and description.
2. User should be able to post answer and can comment on questions, answers.
3. User can upvote/downvote questions, answers and comments.
4. User can follow/unfollow topics of interest.
5. User can search questions, answers and topic.

**OPTIONAL REQUIREMENTS**

1. Notification
2. Analytics

**NON-FUNCTIONAL REQUIREMENTS**

1. Highly Scalability
2. Low latency
3. Highly available(if answers or question posted comes to different user at some time difference, then also it is ok).
4. Eventual Consistency(Eventual consistency is a data modeling concept that ensures that updates made to distributed NoSQL databases will eventually be reflected across all nodes. This ensures that identical database queries will return the same results after some period of time).

**ESTIMATIONS**

Total user : 50M

Daily active users : 10M

Daily action performed by 1 user = 5

Daily action performed by 10M user = 5\*10M = 50M users

Request per second = 50/(24\*3600) = 600/per Sec

**Storage**

Avg data generated by 1 user = 1MB

Avg daily data generated = 1\*10=10 Million MB = 10TB/day

**Bandwidth**

Storage = 10TB/day

10/(24\*3600) = 115GB/sec

**DATA DESIGN MODEL**

**TABLES**

**User**(userId, firstName, lastName, email, createdAt)

**Question**(questionId, userId, title , description, createdAt)

**Answer**(AnswerId, userId, questionId, content, createdAt)

**Comment**(commentId, userId, content, targetId, targetType, createdAt)

**Topic**(topicId, name, createdAt)

**UserFollowsTopic**(Id(pk), userId, topicId) (MANY TO MANY)

Certainly, I can provide you with a textual representation of the data flow between services using SQL and NoSQL databases:

**User Registration and Authentication:**

Client -> Authentication Service (SQL) -> SQL Database

**Posting a Question:**

Client -> Content Service (NoSQL) -> NoSQL Database

**Posting an Answer:**

Client -> Content Service (NoSQL) -> NoSQL Database

**Following a Topic:**

Client -> Topic Service (NoSQL) -> NoSQL Database

**Searching for Content:**

Client -> Search Service (Full-Text Search) -> NoSQL Database

**Receiving Notifications:**

Comment Service (NoSQL) -> Notification Service (NoSQL) -> User's Client

**Moderation and Reporting:**

Client -> Moderation Service (SQL) -> SQL Database

**Analytics and Monitoring:**

Client -> Analytics Service (SQL or NoSQL) -> Analytics Database

**Real-Time Updates:**

Real-Time Service -> User's Client

**File Upload and Storage:**

Client -> File Storage Service (Object Storage) -> Object Storage Service

Please note that the choice between SQL and NoSQL databases depends on the specific requirements of each service. SQL databases are suitable for structured data with complex relationships, while NoSQL databases offer flexibility for unstructured or semi-structured data. Additionally, some services may require a combination of SQL and NoSQL databases based on their data models and access patterns.

**API Design**

Designing a well-structured API is essential for the success of your Quora-like app. An effective API design makes it easier for developers to interact with your system and for different components to communicate seamlessly. Here's an example of how you might design your API, keeping it simple for demonstration purposes:

Base URL:

https://api.yourquoraapp.com/v1/

Authentication:

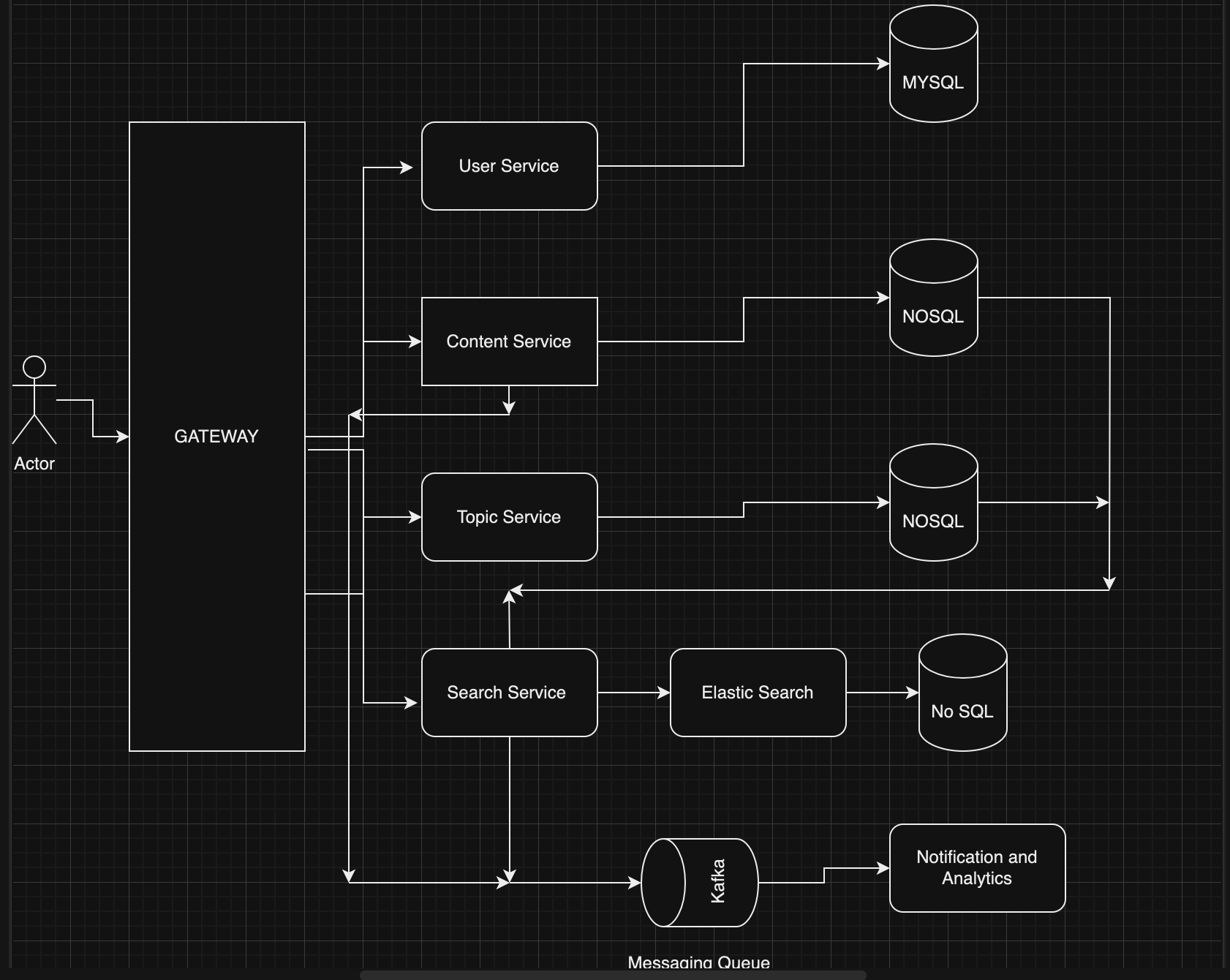
* Use token-based authentication (JWT) for secure user access to endpoints.

Endpoints:

1. User Registration:
   * POST /users/register
   * Request Body: { "username", "email", "password" }
   * Response: { "message": "Registration successful" }
2. User Login:
   * POST /users/login
   * Request Body: { "email", "password" }
   * Response: { "token": "your.jwt.token" }
3. Get User Profile:
   * GET /users/profile
   * Headers: { "Authorization": "Bearer your.jwt.token" }
   * Response: { "user\_id", "username", "email", "created\_at" }
4. Create Question:
   * POST /questions
   * Headers: { "Authorization": "Bearer your.jwt.token" }
   * Request Body: { "title", "description" }
   * Response: { "question\_id", "title", "description", "created\_at" }
5. Get Question Details:
   * GET /questions/:question\_id
   * Response: { "question\_id", "title", "description", "created\_at", "user\_id" }
6. Post Answer:
   * POST /questions/:question\_id/answers
   * Headers: { "Authorization": "Bearer your.jwt.token" }
   * Request Body: { "content" }
   * Response: { "answer\_id", "content", "created\_at", "user\_id" }
7. Get Answers for a Question:
   * GET /questions/:question\_id/answers
   * Response: [ { "answer\_id", "content", "created\_at", "user\_id" }, ... ]
8. Post Comment:
   * POST /comments
   * Headers: { "Authorization": "Bearer your.jwt.token" }
   * Request Body: { "target\_type", "target\_id", "content" }
   * Response: { "comment\_id", "content", "created\_at", "user\_id" }
9. Get Comments for a Target (Question/Answer):
   * GET /comments?target\_type=:target\_type&target\_id=:target\_id
   * Response: [ { "comment\_id", "content", "created\_at", "user\_id" }, ... ]
10. Follow a Topic:
    * POST /topics/:topic\_id/follow
    * Headers: { "Authorization": "Bearer your.jwt.token" }
    * Response: { "message": "You are now following the topic" }
11. Search:
    * GET /search?q=:query
    * Response: [ { "question\_id", "title", "created\_at" }, ... ]

This is a basic API design to illustrate the concept. Depending on your app's features and complexity, you might need to expand and refine the endpoints, request/response structures, error handling, pagination, rate limiting, and more. It's essential to follow RESTful principles and provide clear and consistent documentation for your API.

**HIGH LEVEL DESIGN**



User Registration and Authentication:

Client -> Load Balancer -> Authentication Service (SQL) -> SQL Database

Posting a Question:

Client -> Load Balancer -> Content Service (NoSQL) -> NoSQL Database

Posting an Answer:

Client -> Load Balancer -> Content Service (NoSQL) -> NoSQL Database

Following a Topic:

Client -> Load Balancer -> Topic Service (NoSQL) -> NoSQL Database

Searching for Content:

Client -> Load Balancer -> Search Service (Full-Text Search) -> NoSQL Database

Receiving Notifications:

Comment Service (NoSQL) -> Notification Service (NoSQL) -> User's Client

Moderation and Reporting:

Client -> Load Balancer -> Moderation Service (SQL) -> SQL Database

Analytics and Monitoring:

Client -> Load Balancer -> Analytics Service (SQL or NoSQL) -> Analytics Database

Real-Time Updates:

Real-Time Service -> User's Client

File Upload and Storage:

Client -> Load Balancer -> File Storage Service (Object Storage) -> Object Storage Service

Caching:

Load Balancer <-> Cache (Redis, Memcached) <-> Backend Services

Load Balancing:

Client -> Load Balancer -> Multiple Instances of Backend Services