Documentation for operational group project

May 2024

Mahmoud Najmeh

Maria Ibañez Rubio

Aleksadra Frej

Michal Frost

Contents

A 3 M

1.Define the Problem Statement:

❖ Problem Statement:

The problem our software aims to solve is the need for a modern, user-friendly social media platform that facilitates seamless communication, content sharing, and social interaction among users. In today's digitally connected world, individuals and communities seek platforms where they can express themselves, connect with others, and stay updated on topics of interest. However, existing social media platforms may lack certain features, have complex interfaces, or raise concerns about data privacy and security.

Our social media platform aims to address these issues by providing a comprehensive solution that prioritizes user experience, privacy, and security. The platform will offer a range of features to cater to diverse user needs, including creating profiles, sharing posts, interacting with content, and connecting with other users. It will be designed to be intuitive, visually appealing, and accessible across devices, ensuring a seamless user experience.

❖ Context:

The rise of social media has transformed the way people communicate, share information, and build communities online. From individuals sharing personal updates to businesses promoting their products and services, social media platforms have become integral parts of everyday life. However, concerns about data privacy, algorithmic bias, and online harassment have raised questions about the role and impact of social media in society.

❖ Significance:

Our social media platform aims to offer a safer, more inclusive, and user-centric alternative to existing platforms. By prioritizing user privacy, providing robust

security measures, and fostering a positive online community, we seek to create a platform where users feel empowered to express themselves, connect with others, and engage in meaningful interactions. The significance of our platform lies in its potential to promote digital well-being, fostster genuine connections, and contribute to a healthier online ecosystem.

2. Identify Use Cases:

- **User Registration:**
  - Inputs: User's name, email, password.
  - Processes: User provides registration information, system validates inputs, generates unique user ID, creates user profile.
  - Expected Output: Successful registration confirmation message.
- **User Login:**
  - Inputs: User's email and password.
  - Processes: User submits login credentials, system verifies user identity.
  - Expected Output: Successful login redirects user to the home feed.
- **Create Post:**
  - Inputs: Post content (text, image, or video).
  - Processes: User composes post, attaches media (optional), submits post.
  - Expected Output: Post is published and appears on user's profile and followers' feeds.
- **View Feed:**
  - Inputs: N/A.
  - Processes: System retrieves posts from users followed by the logged-in user.
  - Expected Output: Display of posts in chronological order with options to like, comment, or share.
- **Comment on Post:**
  - Inputs: Comment text, post ID.
  - Processes: User enters comment text, submits comment on a specific post.
  - Expected Output: Comment is added to the post, visible to other users.
- **Like Post:**
  - Inputs: Post ID.
  - Processes: User clicks on like button associated with a post.
  - Expected Output: Post's like count increments, user's like is registered.

- **View Profile:**
  - Inputs: User ID or username.
  - Processes: System retrieves user's profile information and posts.
  - Expected Output: Display of user's profile with options to view/edit profile details and posts.
- **Edit Profile:**
  - Inputs: User's profile details (name, bio, profile picture).
  - Processes: User updates profile information, submits changes.
  - Expected Output: Profile details are updated and reflected on the user's profile page.
- **Follow User:**
  - Inputs: User ID or username of the user to follow.
  - Processes: User selects the option to follow another user.
  - Expected Output: User is added to the follower list, their posts appear in the follower's feed.
- **Search Users:**
  - Inputs: User's name or username.
  - Processes: User enters search query, system retrieves matching user profiles.
  - Expected Output: Display of user profiles matching the search query.
- **Logout:**
  - Inputs: N/A.
  - Processes: User selects the option to log out.
  - Expected Output: User session is terminated, redirected to the login page.

These use cases cover the primary interactions users will have with the social media platform application, from account management to content creation and interaction with other users' content. Each use case outlines the inputs required, the processes involved, and the expected outputs or outcomes.

3. System Architecture:

For the social media platform application, we will adopt a microservices architecture due to its flexibility, scalability, and ease of maintenance. Each

microservice will be responsible for specific functionalities, and they will communicate with each other through well-defined APIs. Below is the proposed system architecture:

- **Components:**

1.User Service:

- Responsible for user management, including registration, authentication, and profile management.
- Handles user-related operations such as user creation, retrieval, update, and deletion.
- Manages user sessions and authentication tokens.

2. Post Service:

- Manages posts created by users, including creation, retrieval, and deletion.
- Handles interactions related to posts, such as liking, commenting, and sharing.
- Generates users' personalized feeds based on their followers and interests.

3. Comment Service:

- Manages comments on posts, including creation, retrieval, and deletion.
- Handles interactions related to comments, such as replying to comments and editing comments.

4. Follow Service:

- Manages user connections and relationships, including following and unfollowing other users.
- Provides functionalities for discovering new users based on common interests or mutual connections.

5. Search Service:

- Provides search functionalities for users, posts, and other content within the platform.
- Implements efficient indexing and searching algorithms to deliver fast and accurate search results.

6. Notification Service:

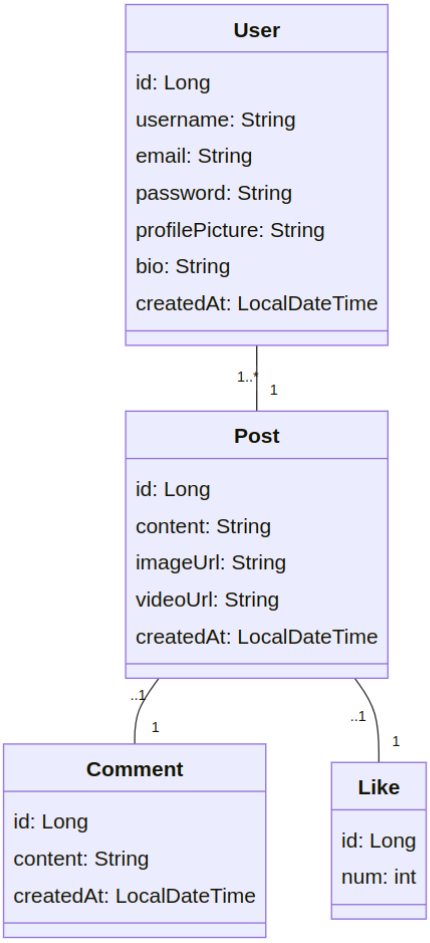
- Sends notifications to users for events such as new followers, likes on their posts, comments on their posts, etc.
- Handles push notifications, emails, or in-app notifications based on user preferences.

- **Communication:**
  - RESTful APIs: Each microservice exposes RESTful APIs for communication with other services and client applications. These APIs follow standard HTTP methods for CRUD operations.
  - Message Queues: Asynchronous communication between services can be achieved using message queues like RabbitMQ or Apache Kafka. This helps in decoupling services and ensures reliability and scalability.
  - API Gateway: An API gateway can be used to provide a unified entry point for client applications, managing authentication, rate limiting, and routing requests to the appropriate microservice.
- **Data Storage:**
  - Database per Microservice: Each microservice manages its database, using a database technology that best suits its requirements. For example, user service might use a relational database for user data, while the post service might use a NoSQL database for scalable storage of posts.
  - Data Replication: Data that needs to be shared across microservices can be replicated or synchronized using techniques like event sourcing or data replication.
- **Deployment:**
  - Containerization: Each microservice can be packaged as a container using Docker, ensuring consistency across different environments and simplifying deployment.
  - Orchestration: Container orchestration tools like Kubernetes can be used to manage and scale the deployment of microservices, providing features like auto-scaling, service discovery, and rolling updates.
- **Monitoring and Logging:**
  - Centralized Logging: Logs from all microservices can be aggregated and stored in a centralized logging system like ELK stack (Elasticsearch, Logstash, Kibana) or Splunk for easy monitoring and analysis.
  - Metrics and Monitoring: Metrics for each microservice can be collected using tools like Prometheus and Grafana, providing insights into performance, resource usage, and errors.

- **Security:**
  - Authentication and Authorization: Each microservice implements its authentication and authorization mechanisms, with JWT (JSON Web Tokens) or OAuth2 being common choices.
  - Encryption: Communication between microservices and with client applications should be encrypted using TLS/SSL to ensure data privacy and security.
  - Input Validation: Input data should be validated at each microservice boundary to prevent security vulnerabilities like SQL injection and XSS attacks.
- **Scalability:**
  - Horizontal Scaling: Microservices architecture inherently supports horizontal scaling, allowing individual services to be scaled independently based on demand.
  - Load Balancing: Load balancers can distribute incoming traffic across multiple instances of each microservice to ensure optimal resource utilization and high availability.

By adopting this microservices architecture, the social media platform application can achieve high scalability, flexibility, and maintainability, while also ensuring performance, security, and reliability.

4. UML Class Diagrams:



This class diagram illustrates the relationships between the User, Post, Comment, and Like classes in the social media platform application. Each class encapsulates specific attributes and behaviours related to users, posts, comments, and likes, forming the core data model of the application.

❖ Relationships between classes:

User:

- One-to-many with Post (posts created by the user)
- One-to-many with Comment (comments made by the user)
- Many-to-many with User (users following each other)

Post:

- Many-to-one with User (user who created the post)
- One-to-many with Comment (comments on the post)
- One-to-many with Like (likes on the post)

Comment:

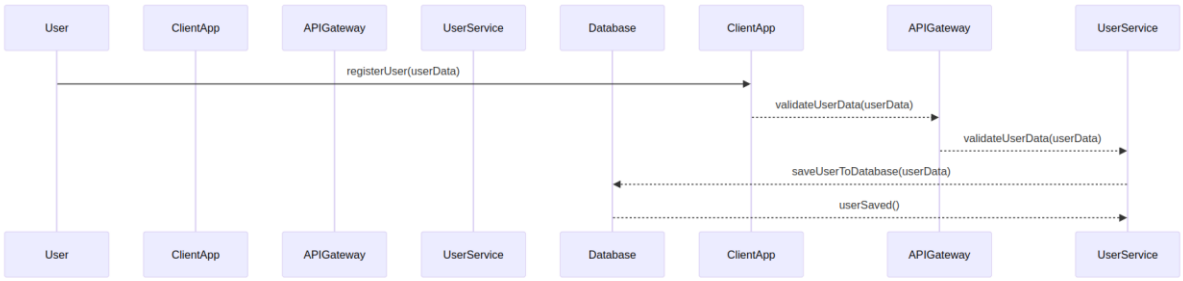
- Many-to-one with User (user who made the comment)
- Many-to-one with Post (post on which the comment was made)

Like:

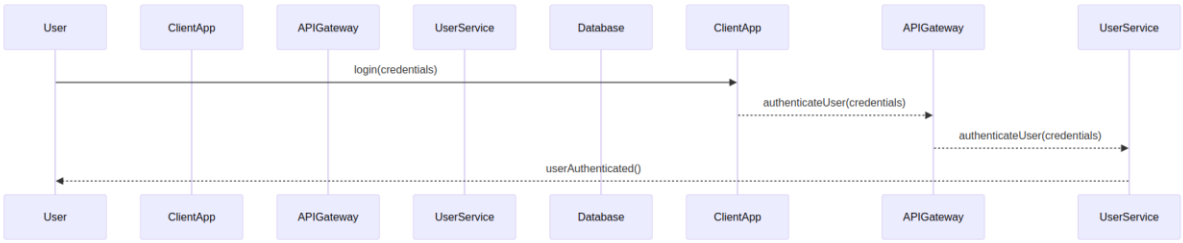
- Many-to-one with User (user who liked the post)
- Many-to-one with Post (post that was liked)

5. Sequence Diagrams:

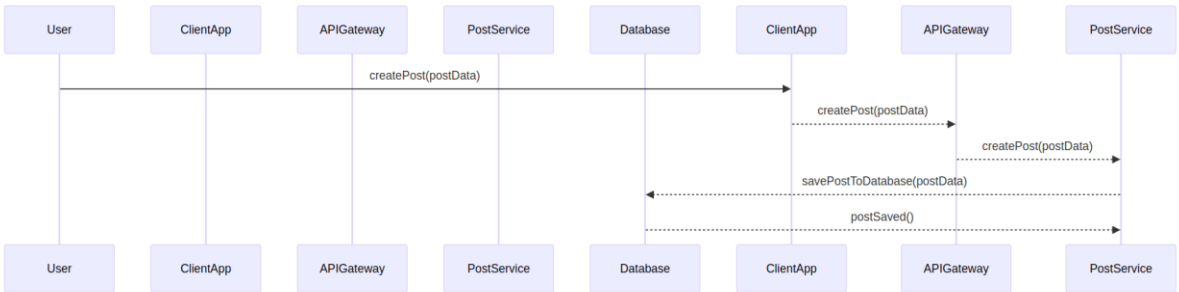
❖ User Registration:



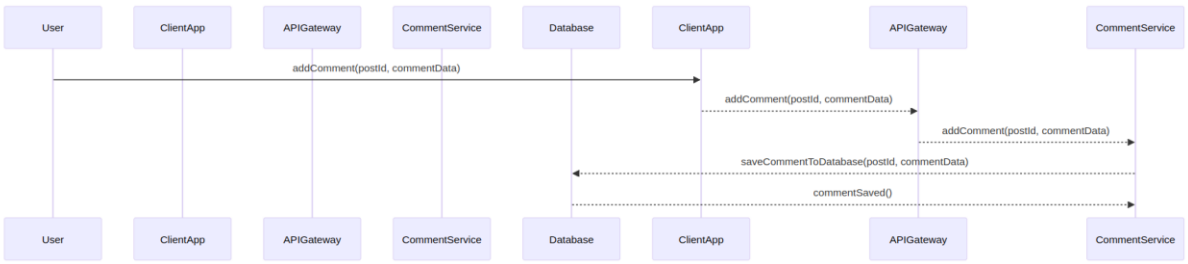
❖ User Login:



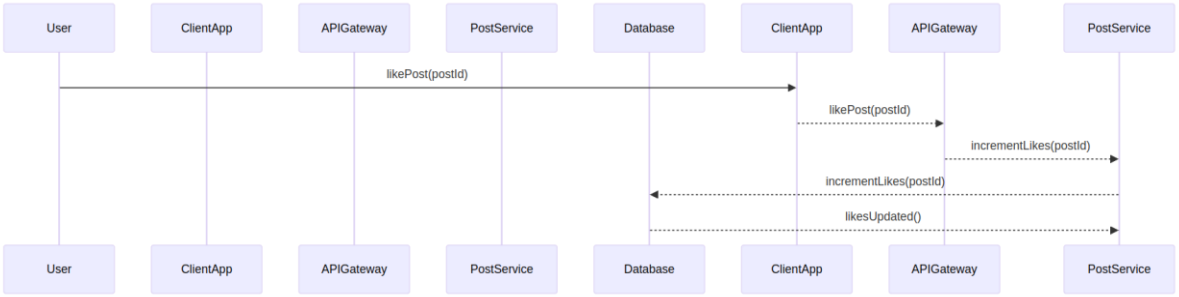
❖ Create Post:



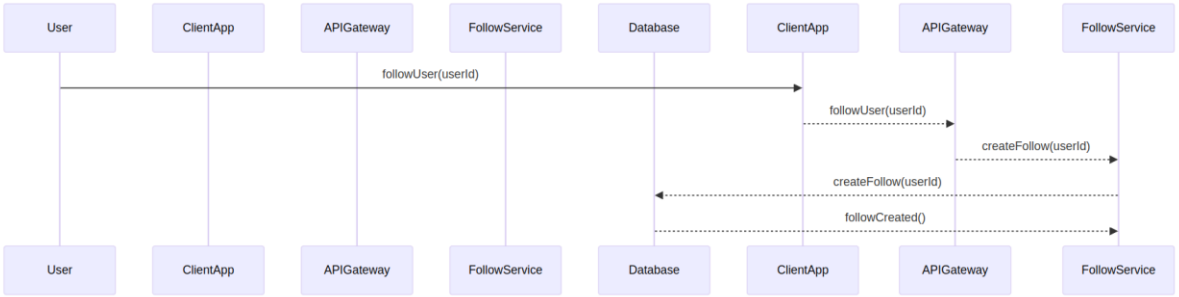
❖ Add Comment:



❖ Like Post:



❖ Follow User:



These sequence diagrams illustrate the interactions between various components during each specific use case in the social media platform application. Each diagram outlines the flow of messages and actions between the user, client application, API gateway, relevant service, and the database.

6. Component Design:

For the component design of the social media platform application, we can break down the project into several smaller components or modules. Here's a

proposed breakdown along with the responsibilities of each component and how they will interact with each other:

- **User Management Component:**
  - Responsibilities:
  - Handles user registration, login, authentication, and authorization.
  - Manages user profiles and settings.
  - Interaction:
  - Communicates with the User Service for user-related operations.
  - Integrates with the Security Service for authentication and authorization.
- **Post Management Component:**
  - Responsibilities:
  - Manages the creation, retrieval, updating, and deletion of posts.
  - Handles interactions related to posts such as likes and comments.
  - Interaction:
  - Communicates with the Post Service for post-related operations.
  - Interacts with the User Service to associate posts with users.
  - Coordinates with the Comment Service for managing comments on posts.
  - Collaborates with the Like Service for handling likes on posts.
- **Comment Management Component:**
  - Responsibilities:
  - Manages the creation, retrieval, updating, and deletion of comments on posts.
  - Interaction:
  - Communicates with the Comment Service for comment-related operations.
  - Interacts with the Post Service to associate comments with posts.
  - Collaborates with the User Service to link comments with users.
- **Like Management Component:**
  - Responsibilities:
  - Handles the liking/unliking of posts by users.
  - Interaction:
  - Communicates with the Like Service for like-related operations.
  - Interacts with the Post Service to associate likes with posts.
  - Collaborates with the User Service to link likes with users.

❖ Follow Management Component:

- Responsibilities:

- Manages the following/unfollowing of users by other users.

- Interaction:

- Communicates with the Follow Service for follow-related operations.

- Interacts with the User Service to establish follower-followed relationships.

❖ Security Service:

- Responsibilities:

- Provides authentication and authorization functionalities.

- Manages user sessions and access control.

- Interaction:

- Integrated with the User Management Component for user authentication and authorization.

- Collaborates with other components to enforce access control policies.

❖ API Gateway:

- Responsibilities:

- Acts as a single entry point for client applications to interact with the backend services.

- Routes incoming requests to the appropriate microservices.

- Provides load balancing and routing functionalities.

- Interaction:

- Directs requests from client applications to the corresponding microservices.

- Communicates with the various microservices to fulfill client requests.

Each component encapsulates specific functionalities related to its domain, promoting modularity and separation of concerns. They interact with each other through well-defined interfaces and APIs, enabling loose coupling and facilitating easier maintenance and scalability of the system.

7. Error Handling and Exception Handling:

Error handling and exception handling are crucial aspects of any software application, including a social media platform. Here's a plan for how the application will handle errors and exceptions gracefully:

- **Define Error Codes and Messages:**
  - Establish a standardized set of error codes and messages to represent different types of errors and exceptions that may occur throughout the application.
  - Error codes should be unique and descriptive, enabling easy identification and debugging of issues.
  - Include meaningful error messages that provide relevant context and guidance for users and developers.
- **Categorize Errors:**
  - Categorize errors based on their severity and impact on the application and user experience.
  - Common categories may include:
  - Validation errors (e.g., invalid input data)
  - Authentication and authorization errors
  - Resource not found errors
  - Internal server errors
  - Each category may have its own set of error codes and messages.
- **3. Implement Robust Exception Handling:**
  - Use try-catch blocks and exception handling mechanisms to capture and manage exceptions at appropriate levels of the application's architecture.
  - Handle exceptions gracefully by providing fallback mechanisms or alternative paths whenever possible.
  - Log detailed error information, including stack traces, to facilitate troubleshooting and debugging.
- **HTTP Status Codes:**
  - Utilize appropriate HTTP status codes to convey the outcome of API requests to clients.
  - Common status codes include 200 (OK), 400 (Bad Request), 401 (Unauthorized), 404 (Not Found), 500 (Internal Server Error), etc.

- Map application-specific errors to corresponding HTTP status codes for consistency and clarity.

❖ **Error Response Formats:**

- Define a standardized format for error responses returned by the API endpoints.
- Include error code, message, and possibly additional details or metadata in the response payload.
- Ensure consistency in the structure of error responses across different endpoints to simplify client-side error handling.

❖ **Recovery Strategies:**

- Implement recovery strategies to gracefully recover from transient errors or failures, such as network timeouts or temporary service unavailability.
- Use retry mechanisms with exponential backoff to retry failed operations with increasing delays.
- Provide informative messages to users when errors occur, suggesting actions they can take to resolve the issue or seek assistance.

❖ **User-Friendly Error Messages:**

- Craft user-friendly error messages that are easy to understand and actionable.
- Avoid technical jargon or overly cryptic error messages that may confuse users.
- Provide clear instructions or suggestions on how users can resolve the issue or contact support for assistance.

By implementing robust error handling and exception handling strategies, the social media platform application can ensure a more resilient and user-friendly experience for its users while facilitating efficient troubleshooting and maintenance by developers.

8. Security design:

For the security design of the social media platform application, it's essential to incorporate various measures to protect user data, prevent unauthorized access, and ensure the integrity and confidentiality of information. Here's an outline of the security measures to be implemented:

- **Authentication:**
  - Implement a robust authentication mechanism to verify the identity of users accessing the application.
  - Use industry-standard protocols such as OAuth 2.0 or JWT (JSON Web Tokens) for user authentication.
  - Support multi-factor authentication (MFA) for an additional layer of security.
  - Enforce strong password policies, including password hashing and salting for storing user credentials securely.
- **Authorization:**
  - Implement fine-grained access control mechanisms to determine what actions users are allowed to perform within the application.
  - Define roles and permissions for different user types (e.g., regular users, admins) and restrict access to sensitive functionalities or data based on these roles.
  - Use role-based access control (RBAC) or attribute-based access control (ABAC) to enforce authorization policies.
- **Encryption:**
  - Encrypt sensitive data at rest and in transit to protect it from unauthorized access or tampering.
  - Utilize HTTPS (HTTP over SSL/TLS) for secure communication between clients and servers to prevent eavesdropping and man-in-the-middle attacks.
  - Implement encryption algorithms such as AES (Advanced Encryption Standard) for encrypting data stored in databases or files.
- **Input Validation:**
  - Implement thorough input validation to prevent common security vulnerabilities such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).
  - Validate and sanitize user input on both client and server sides to mitigate

the risk of malicious input.

- **Session Management:**
  - Manage user sessions securely to prevent session hijacking or fixation attacks.
  - Use secure, randomly generated session tokens with short expiration times.
  - Implement mechanisms to detect and invalidate inactive or expired sessions.
- **Security Headers:**
  - Set appropriate security headers in HTTP responses to mitigate common web security risks.
  - Include headers such as Content-Security-Policy (CSP), X-Content-TypeOptions, X-Frame-Options, and X-XSS-Protection to prevent XSS attacks, clickjacking, and MIME type sniffing.
- **API Security:**
  - Secure APIs against unauthorized access by implementing authentication and authorization mechanisms.
  - Use API keys, OAuth tokens, or JWT tokens to authenticate API requests.
  - Validate and sanitize input data received from API clients to prevent injection attacks.

- ***Security Auditing and Logging:***
  - Implement logging mechanisms to record security-relevant events and activities within the application.
  - Perform regular security audits and vulnerability assessments to identify and remediate potential security weaknesses.
  - Monitor system logs and audit trails for suspicious activities and security incidents.

By incorporating these security measures into the design and implementation of the social media platform application, you can create a more secure environment for users and protect their data from various threats and vulnerabilities. Regularly updating and patching the application to address emerging security risks is also essential to maintain its security posture over time.

9. Performance Optimization:

Performance optimization is crucial for ensuring that the social media platform application delivers a responsive and efficient user experience, especially as the user base grows. Here are some strategies to identify potential performance bottlenecks and optimize the application's performance:

- **Profiling and Monitoring:**
  - Use profiling tools and performance monitoring solutions to identify areas of the application that contribute to performance bottlenecks.
  - Monitor key performance metrics such as response times, CPU and memory usage, database queries, and network latency.
  - Set up alerts and notifications to detect and address performance issues proactively.
- **Database Optimization:**
  - Optimize database queries by indexing frequently accessed columns and optimizing complex queries.
  - Implement database caching mechanisms to reduce the number of database queries and improve response times.
  - Consider denormalizing data or using NoSQL databases for read-heavy workloads to improve database performance.
- **Caching Strategies:**
  - Implement caching at various levels of the application to reduce the need for expensive computations or database queries.
  - Utilize in-memory caching solutions such as Redis or Memcached to cache frequently accessed data or computed results.
  - Employ HTTP caching mechanisms for static assets and API responses to reduce server load and improve latency for client requests.
- **Load Balancing and Scaling:**
  - Use load balancers to distribute incoming traffic across multiple server instances to prevent overloading of individual servers.
  - Implement horizontal scaling by adding more server instances to handle increasing load and traffic spikes.
  - Use auto-scaling solutions to automatically adjust the number of servers instances based on demand.

- **Asynchronous Processing:**
  - Offload long-running or resource-intensive tasks to background processes or worker queues to avoid blocking the main application thread.
  - Use asynchronous processing techniques such as message queues or eventdriven architectures to handle asynchronous tasks efficiently.
- **Content Delivery Networks (CDNs):**
  - Utilize CDNs to cache and serve static assets such as images, CSS, and JavaScript files from edge servers located closer to the user's geographic location.
  - Offloading static asset delivery to CDNs can reduce latency and improve page load times for users across different regions.
- **Optimized Frontend Rendering:**
  - Minimize the size of frontend assets such as CSS and JavaScript files by minification and compression techniques.
  - Implement lazy loading of assets and images to defer loading until they are needed, reducing initial page load times.
  - Optimize client-side rendering performance by avoiding excessive DOM manipulation and optimizing JavaScript code.
- **Network Optimization:**
  - Reduce the number of HTTP requests by bundling and combining assets such as CSS and JavaScript files.
  - Use HTTP/2 or HTTP/3 protocols to enable multiplexing and reduce latency for concurrent requests.
  - Compress server responses using gzip or Brotli compression to reduce data transfer size and improve network performance.

By implementing these performance optimization strategies, the social media platform application can deliver a fast and responsive user experience, even under high traffic conditions, while efficiently utilizing resources and infrastructure. Regular performance testing and tuning are essential to ensure that the application maintains optimal performance over time.