**Overview**

The **Scalable Chat System** project implements a real-time chat solution that supports communication between admins and users, with robust deployment and scalability in mind. The system utilizes modern backend technologies, WebSocket-based real-time messaging, and supports seamless scaling in distributed environments.

**Features Implemented**

1. **Admin and User Communication**:
   * Admins can communicate with users and other admins.
   * Users can communicate only with admins.
   * Messages are routed securely based on roles and connection status.
2. **Real-Time Communication**:
   * Implemented using **Socket.IO** for WebSocket-based bidirectional communication.
   * Admins and users receive messages in real-time.
3. **Scalability and Resilience**:
   * Redis Pub/Sub for message broadcasting across multiple instances.
   * Stateless backend supports horizontal scaling.
4. **Active Connection Management**:
   * Redis hash storage maintains active connections for users and admins.
   * Tracks online status and ensures messages are routed accurately.
5. **Server-Sent Events (SSE)**:
   * Real-time updates for features like posts and system updates.
   * SSE endpoints emit changes to connected clients.
6. **REST APIs**:
   * CRUD operations for posts and managing system resources.
   * Designed for both admin and user roles.
7. **Deployment Strategy**:
   * Dockerized services for portability.
   * Kubernetes deployment manifests provided, supporting:
     + Round-robin load balancing.
     + Auto-scaling with a minimum of 3 replicas for each app.

**Architecture**

**Technology Stack**

* **Backend Framework**: NestJS
* **Real-Time Communication**: Socket.IO, Redis Pub/Sub
* **Database**: Redis
* **Containerization**: Docker
* **Orchestration**: Kubernetes

**System Architecture**

1. **Client**:
   * Admin and User interfaces connect via WebSocket for real-time messaging.
   * REST APIs for data manipulation and retrieval.
2. **Backend Services**:
   * **Admin App**: Handles admin-related messaging and operations.
   * **User App**: Handles user-related messaging and operations.
   * **Shared Library**: Common utilities and data models shared across apps.
3. **Redis**:
   * Pub/Sub system for inter-service message broadcasting.
   * Storage for active connections and post data.
4. **Deployment Environment**:
   * **Docker**: Containerized all apps and Redis for local development.
   * **Kubernetes**:
     + Multiple replicas of each app.
     + Horizontal Pod Autoscaler (HPA) for scaling based on CPU/memory usage.
     + Load balancing across replicas.

**How It Works**

**Real-Time Messaging**

1. **Connection Establishment**:
   * Users and admins connect via WebSocket with unique identifiers.
   * Active connections are stored in Redis under separate namespaces (activeConnections:users, activeConnections:admins).
2. **Message Flow**:
   * A message from an admin or user is published to Redis using the chat channel.
   * All instances subscribe to this channel, ensuring messages are delivered irrespective of which instance the sender or recipient is connected to.
3. **Recipient Identification**:
   * The recipient's socket ID is fetched from Redis.
   * The message is forwarded to the recipient using Socket.IO.

**Server-Sent Events (SSE)**

* Changes in posts or system data trigger updates, which are pushed to connected clients using SSE.
* Keeps clients updated on changes in near real-time.

**Deployment Details**

**Local Development**

1. **Pre-requisites**:
   * Docker and Docker Compose installed.
2. **Steps**:
   * Clone the repository and navigate to the project directory.
   * Run the following command:

docker-compose up --build

* + Admin App runs on http://localhost:3000.
  + User App runs on http://localhost:3001.

**Production Deployment**

1. **Kubernetes Deployment**:
   * YAML manifests are provided for deploying the system to a Kubernetes cluster.
   * Features:
     + **3 replicas per service**: Ensures high availability.
     + **Load Balancing**: Round-robin balancing via Kubernetes service.
     + **Horizontal Scaling**: Autoscaler adjusts replicas based on load.
   * Deploy using:

kubectl apply -f kubernetes-manifests/

* + Check status:
    - kubectl get pods
    - kubectl get services

**Achievements**

1. **Scalable Architecture**:
   * Stateless apps enable scaling across multiple nodes.
   * Redis ensures cross-instance communication.
2. **Seamless Real-Time Messaging**:
   * Users and admins receive real-time updates with minimal latency.
3. **Robust Deployment Plan**:
   * Kubernetes deployment ensures fault tolerance and high availability.
4. **Comprehensive Testing**:
   * Functional tests for WebSocket, SSE, and REST APIs.
   * Load tests to validate scalability under high traffic.