Technical Design Document

Realtime Chat Application

A modular event-driven realtime chat system leveraging WebSockets, Kafka, Redis, Postgres, and SMTP for OTP-based authentication, deployed on k3s with Traefik and HTTPS.

System Architecture Overview

The system is a modular Spring Boot backend with a Vue-based frontend. WebSockets provide realtime messaging. Kafka brokers chat events to decouple persistence from delivery. Redis handles sessions and presence. Postgres stores persistent chat data. SMTP delivers OTPs. k3s hosts workloads with Traefik ingress and TLS.

Component Breakdown

Backend Modules:

- Auth module (OTP, sessions, rate limiting)
- WebSocket gateway (connections, presence, typing, read receipts)
- Messaging producer (publish to Kafka)
- Messaging consumer (persist to Postgres, dispatch to clients)
- Presence manager (Redis heartbeats)
- Retention scheduler (cleanup)
- SMTP client (OTP delivery)

Shared Components:

- Redis: sessions, presence, typing
- Kafka: event bus
- Postgres: persistent storage

Frontend:

- Auth screens
- Contact list
- Chat panel
- WebSocket client
- Pinia state store

Data Model

User: id, phoneNumber, createdAt
Session: Redis token, userId, expiry
Message: id, senderId, recipientId, body, timestamp, readAt
Presence: Redis TTL keys per user
Typing: Redis TTL keys per user-pair
OTP: Ephemeral codes with TTL and throttle metadata

API and Protocol Design

REST Endpoints

- POST /auth/request-otp { phone }
- POST /auth/verify-otp { phone, otp }
- GET /contacts
- GET /messages/{recipient}?before={ts}
- POST /read-receipt { messageIds }

WebSocket Events

- chat.send { to, body }
- chat.receive { from, body, timestamp }
- presence.update { userId, status }
- typing.start { to }
- typing.stop { to }
- read.update { messageIds }

Messaging Event Flow

1. Client emits `chat.send` over WebSocket.
2. Gateway validates session and publishes event to Kafka.
3. Kafka consumer persists message in Postgres.
4. Consumer dispatches message to gateway.
5. Gateway forwards `chat.receive` to target if online, otherwise marks unread.

Session & Security

OTP login verifies phone ownership. Server-side sessions stored in Redis keyed by secure cookie. WebSockets reuse session cookie. CSRF tokens protect REST. TLS enforced via Traefik. Rate limiting applied to OTP requests. Input sanitized to prevent injection.

Presence, Typing & Read Receipts

Presence via Redis heartbeat TTL ($\approx 20s$). Typing via TTL keys ($\approx 5s$). Read receipts update message state in Postgres and notify sender.

Retention & Cleanup

Weekly scheduler deletes messages older than seven days. No cascading complexity due to simple schema. Optionally vacuum for DB performance.

Deployment Topology

Single-node k3s cluster:

- Traefik ingress (LetsEncrypt TLS, WebSocket routing)
- Spring Boot service
- Vue app served via Nginx container
- Kafka broker (single node for learning)
- Redis (single node)
- Postgres (single node + PVC)

Secrets stored via Kubernetes Secrets. Volumes for Postgres.

Scalability

Kafka enables decoupled fanout. Redis centralizes presence and sessions. Horizontal scaling possible by splitting gateway and consumer pods. Single VPS sufficient for 100 users.

Failure Modes & Edge Cases

WebSocket reconnect with backoff. OTP spam rejection. Kafka failure causes degradation. Redis failure breaks sessions. Postgres failure breaks persistence. At-least-once semantics with idempotent DB writes.

Observability

Structured logs. Optional Prometheus/Grafana. Health via Actuator. Optional tracing via OpenTelemetry.

Tech Stack Justification

Spring Boot simplifies modular monolith design. Kafka decouples persistence. Redis accelerates ephemeral state. Postgres ensures durability. Traefik automates HTTPS. Vue + Vuetify streamline frontend. k3s gives platform learning at low cost.