NexGate Chat — Phase 1

Foundation & Local Experiments

NexGate / QBIT SPARK | Version 1.0 *Spring Boot WebSocket · Commerce DMs · Offline Delivery · Local Experiments*

Table of Contents

---

1. Phase 1 Goals

Phase 1 is about shipping fast and learning.

The goal is not to build the perfect infrastructure from day one. The goal is to get NexGate chat into users' hands as quickly as possible — while running local experiments in parallel that prepare for Phase 2.

```
Two tracks running simultaneously:

Track A — Production (ship it):        Track B — Experiments (learn it):
  Spring Boot WebSocket                  Ejabberd local Docker setup
  Text chat + voice notes                Spring Boot ↔ Ejabberd auth bridge
  Commerce DMs (both flows)              WebRTC voice call on emulators
  Offline delivery + notifications       Coturn TURN relay local test
  Message receipts                       MessagePack encoding test
  Isolated shop inbox
```

Phase 1 architecture is intentionally simpler than Phase 2. Everything designed here carries forward — the schema, the commerce logic, the notification system, the inbox model. Phase 2 only swaps the transport layer.

---

2. What Ships in Phase 1

Messaging Features

1:1 Personal DMs	text, voice notes, media cards
Group chats	up to 500 members
Broadcast channels	creator → fans (one-way)
Voice notes	Opus recorded, waveform rendered
Rich content cards	product, custom price, event, Bei ya pamoja, post, stream link

Commerce DM Features

Buyer initiates chat	from product page → product card auto-appears
Seller attaches product	WhatsApp-style from inside any conversation
Custom price offer	private to buyer, public price unchanged
Proceed to checkout	button in chat → redirects to checkout
Order updates in thread	confirmation, shipping, delivery

Inbox Features

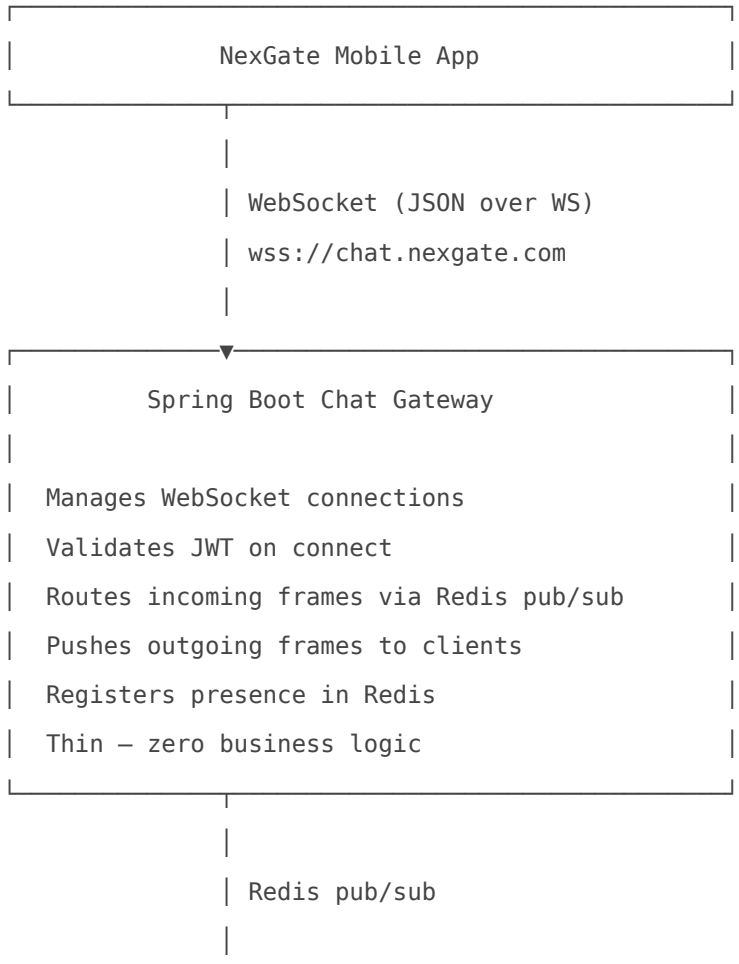
Isolated shop inbox	separate tab per shop
Personal inbox	always private, owner only
Shop inbox	shared with authorized staff (Pro tier)
Message receipts	sent / delivered / read ticks
Typing indicators	ephemeral, Redis TTL based
Online presence	shown in conversation header

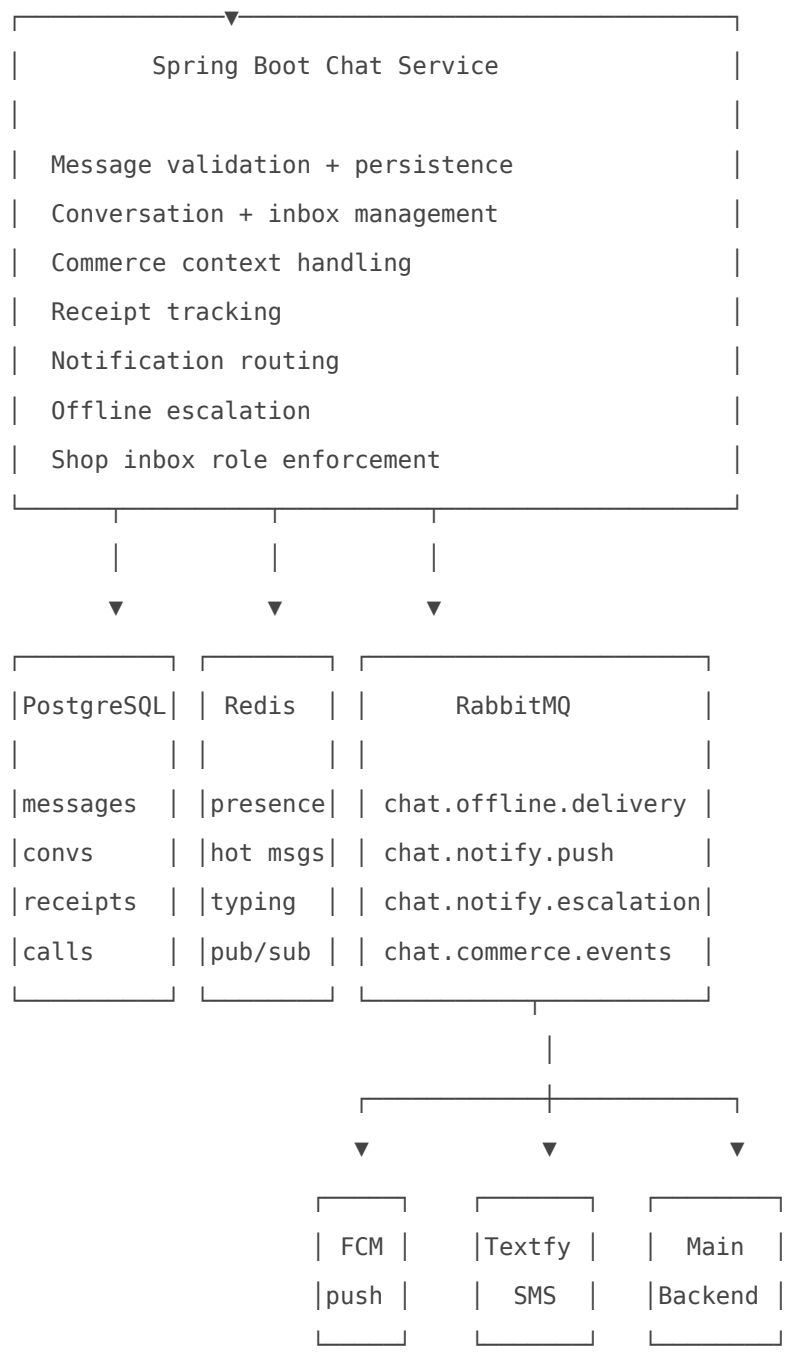
Notification Features

FCM push (Android + iOS)	HIGH priority, bypasses Doze mode
Textfy SMS escalation	CRITICAL and IMPORTANT messages
Offline queue	RabbitMQ holds messages until delivery
Catch-up banner	summary on reconnect after offline

---

3. Architecture Overview





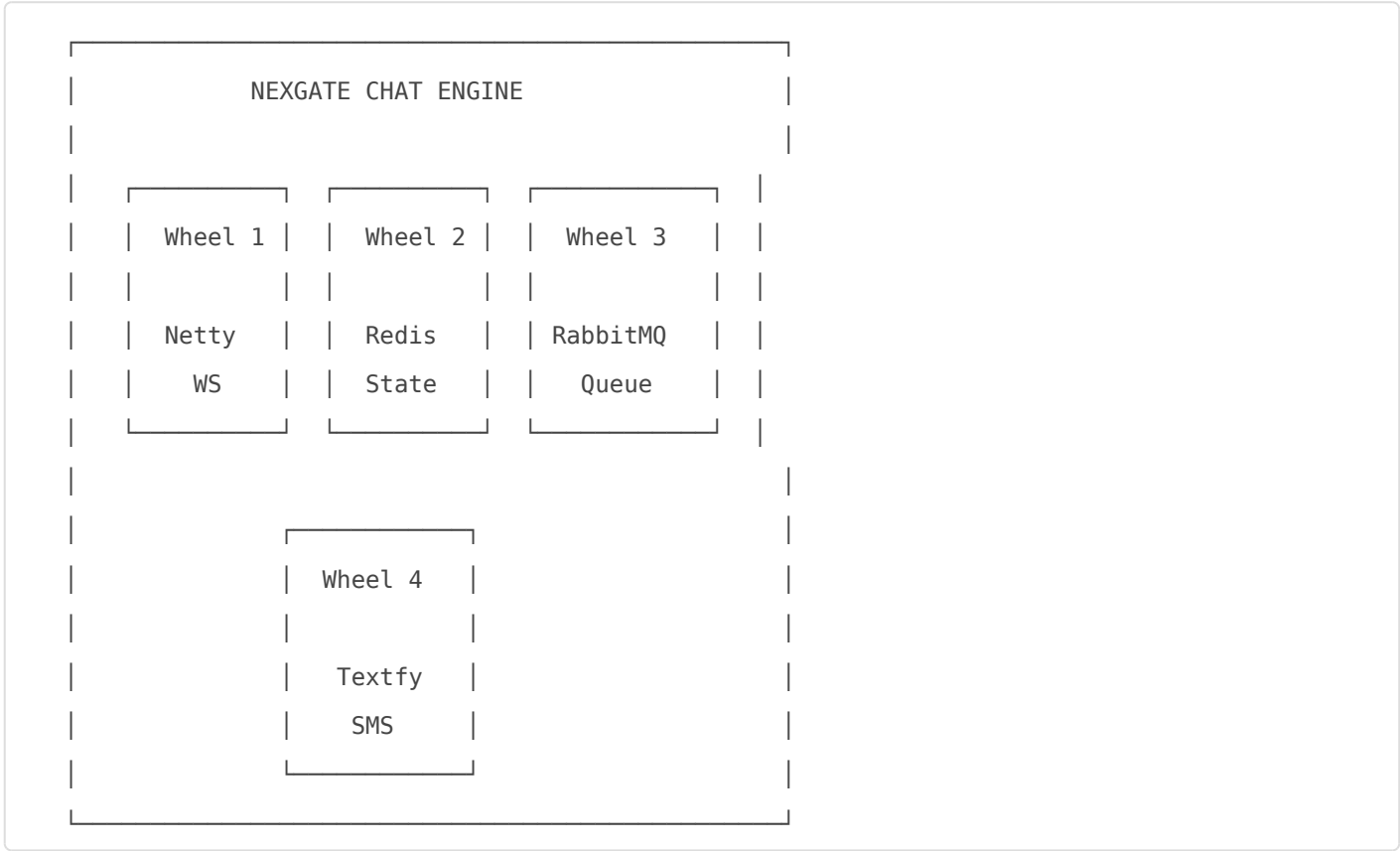
Key Rule

```
Gateway never touches business logic
Chat Service never manages WS connections
Both communicate only via Redis pub/sub

Gateway → Redis pub/sub → Chat Service   (inbound)
Chat Service → Redis pub/sub → Gateway   (outbound)
```

4. The Four Wheels

Just as File Thunder has four processing wheels, the Phase 1 chat engine has four foundational components:



Wheel 1 — Netty WebSocket Spring Boot uses Netty under the hood for WebSocket connections. Handles connect, disconnect, heartbeat, and frame routing. Scales to 50k+ concurrent connections per pod.

Wheel 2 — Redis State Tracks everything ephemeral and hot: online presence per user, typing indicators (5s TTL), last 50 messages per conversation (capped list), unread counts, cross-pod pub/sub routing, notification escalation timers.

Wheel 3 — RabbitMQ Queue Handles everything async: offline message delivery queue, delayed SMS escalation jobs, commerce event publishing to Main Backend, receipt acknowledgment processing.

Wheel 4 — Textfy SMS Critical and important message fallback. NexGate's own SMS platform — zero third-party cost. Swahili templates per message type. Deep links back to specific conversation. Full delivery audit log.

---

5. Service Design

Chat Gateway Responsibilities

```
On WebSocket connect:
  Validate JWT token
  Register user presence in Redis:
    presence:{userId} → TTL 30s (refreshed by heartbeat)
  Drain offline queue via RabbitMQ trigger

On frame received (inbound):
  Validate session
  Publish to Redis: chat:inbound
  ACK client immediately with temp_id

On Redis pub/sub message (outbound):
  Find client connection for recipient
  Push WS frame to device

On WebSocket disconnect:
  Remove presence from Redis
  Update last_seen_at via RabbitMQ event
```

Chat Service Responsibilities

```
On inbound message event (from Redis):
  Validate sender is conversation member
  Check conversation not blocked/archived
  Resolve message level (NORMAL / IMPORTANT / CRITICAL)
  Write to PostgreSQL
  Write to Redis hot cache
  Fan-out to recipients:
    Online  → Redis pub/sub → Gateway → WS push
    Offline → RabbitMQ queue + FCM + escalation timer

On commerce message:
  Attach product snapshot (frozen at send time)
```

```
  Emit commerce event to Main Backend via RabbitMQ

On receipt ack:
  Update message_receipts in PostgreSQL
  Notify sender (tick update) via Redis pub/sub

On presence event (user online):
  Drain RabbitMQ offline queue for user
  Send catch-up summary if messages missed
```

RabbitMQ Exchange Design

```
Exchange: nexgate.chat (topic)

Routing Key                    Flow
_____________________________________________________

chat.message.inbound           Gateway → Chat Service
chat.message.outbound          Chat Service → Gateway
chat.notify.push               Chat Service → FCM Worker
chat.notify.escalation         Chat Service → SMS Worker
chat.receipts.delivered        Gateway → Chat Service
chat.receipts.read             App → Chat Service
chat.commerce.initiated        Chat Service → Main Backend
chat.commerce.price.attached   Chat Service → Main Backend
chat.presence.online           Gateway → Chat Service
chat.presence.offline          Gateway → Chat Service
```

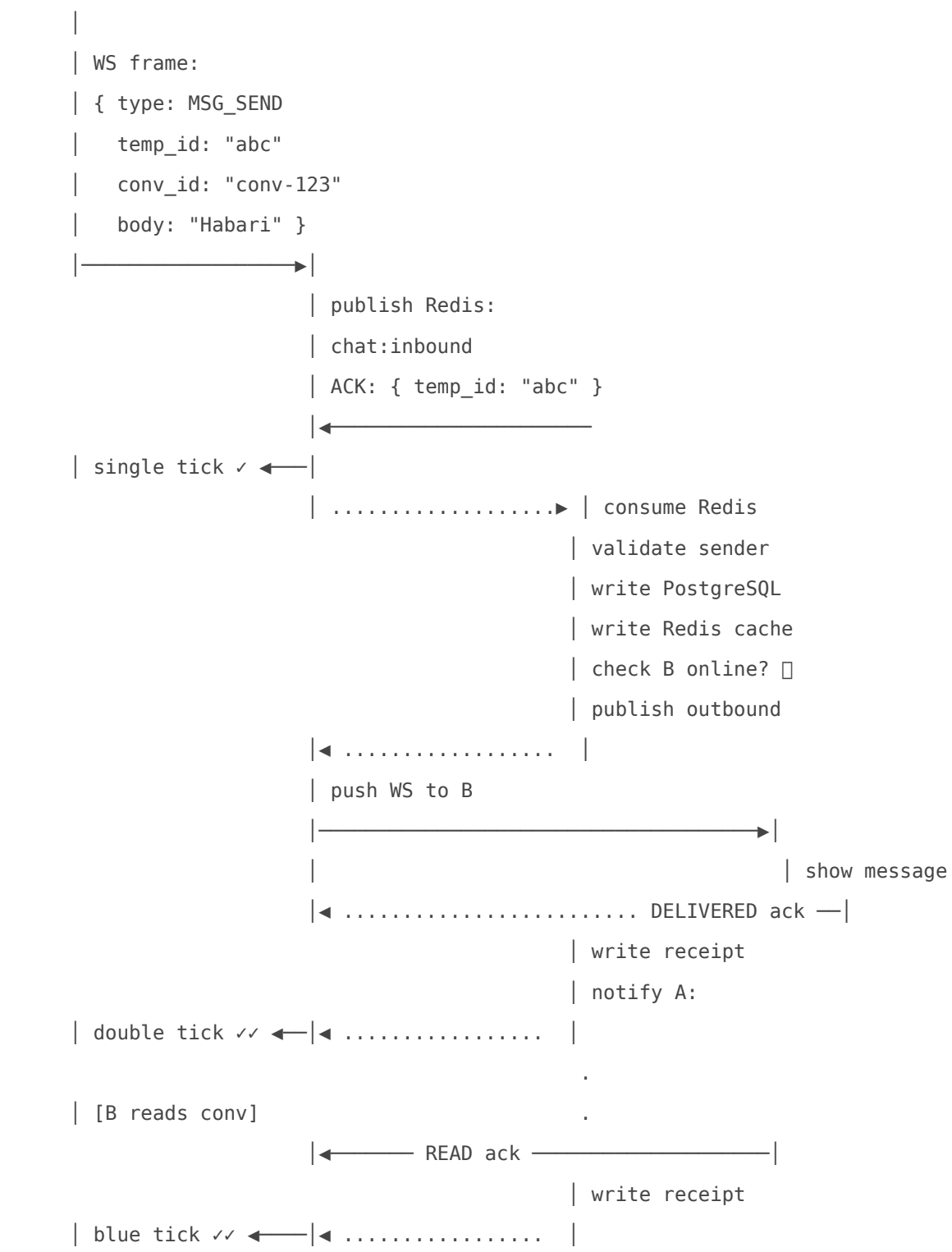
---

6. Data Flows

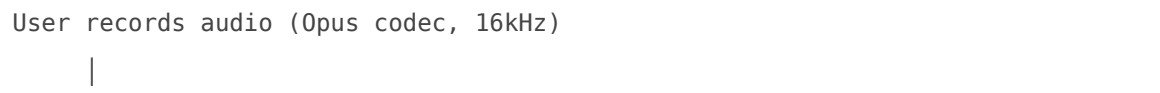
Text Message — Full Sequence

```
[Client A]              [Gateway]          [Chat Service]      [Client B]

type "Habari"
tap Send
show pending □
```



Voice Note — Send Flow



```
     ▼ Upload to File Thunder
GET /media/upload-request (DM_ATTACHMENT context)
→ presigned MinIO URL returned
Upload .ogg file directly to MinIO
POST /media/confirm { fileId }
     │
     ▼ File Thunder processes:
ClamAV scan
Waveform extraction via FFmpeg
  → amplitude array (50 values for UI bars)
  → waveform.webp thumbnail
Store in nexgate-private/messages/{convId}/{fileId}/
     │
     ▼ File Thunder returns: { fileId, waveformData[], durationSeconds }
     │
App sends WS frame:
{ type: MSG_SEND
  content_type: VOICE_NOTE
  media_ref: fileId
  duration_seconds: 15
  waveform: [0.2, 0.8, 0.6, ...] }  ← embedded for instant UI
     │
     ▼ Same path as text message
Recipient receives message with waveform data
Waveform bars render instantly (no extra request)
Tap play → GET /chat/media/{fileId}/url
         → signed URL (5 min TTL)
         → stream audio progressively
```

Offline Delivery Flow

```
Message arrives for offline user B
     │
Chat Service:
  Check Redis: B online? □
     │
     ├──► RabbitMQ: chat.offline.delivery
     │      { messageId, recipientId, level }
     │
```

```
      ├──► FCM HIGH priority push
      │      { type: NEW_MESSAGE
      │        convId, senderName, preview }
      │
      └──► RabbitMQ: chat.notify.escalation
             delay: CRITICAL=0min, IMPORTANT=10min
      .
      .  (time passes)
      .
Escalation consumer wakes:
  Check Redis: B online now? □ → cancel, done
                             □ → send Textfy SMS
      │
Textfy SMS:
  "NexGate: Ujumbe mpya kutoka Juma.
   nexgate.app/chat/conv-123"
      │
User turns on WiFi / opens app:
  WS reconnects → Gateway registers presence
  Chat Service drains RabbitMQ queue (priority order)
  CRITICAL first → IMPORTANT → NORMAL
  Show catch-up banner:
    "Umekosa: 2 maagizo, 5 ujumbe"
  DELIVERED receipts fire for all drained messages
```

---

7. Commerce DM Flows

Flow 1 — Buyer Initiates from Product Page

```
Buyer on product page
      │ taps "Chat with Seller"
      ▼
POST /chat/commerce/initiate
  { productId, shopId }
      │
Chat Service:
  Find or create DM conversation
```

```
    conversation.type = COMMERCE
    conversation.owner_type = SHOP
    conversation.owner_id = shopId
      |
  Fetch product snapshot from Main Backend:
    { name, price, images[0], stock, shopName }
    Snapshot frozen at this exact moment □
    Public price change later → does not affect this card
      |
  Create first message automatically:
    type: PRODUCT_CARD
    context_type: PRODUCT
    context_ref_id: productId
    snapshot_json: { frozen product data }
      |
     ▼ Seller receives in shop inbox tab
┌──────────────────────────────────┐
│ □□Samsung A15                    │
│ TZS 450,000                      │
│ In stock: 12 units               │
│ TechStore                        │
│                                  │
│ [Reply] [View Product]           │
└──────────────────────────────────┘
     |
Negotiation happens in thread
     | Agreement reached
     ▼
Seller attaches custom price offer:
┌──────────────────────────────────┐
│ □□Special Price Offer            │
│ Samsung A15                      │
│ TZS 400,000  (was 450,000)       │
│ Valid for you only               │
│                                  │
│ Quantity: [− 1 +]                │
│ [Proceed to Checkout →]          │
└──────────────────────────────────┘
     |
```

```
Buyer taps Proceed → redirected to checkout
Checkout outside inbox — at negotiated price
Public product price: TZS 450,000 unchanged □
     |
Order placed → confirmation back in thread:

┌──────────────────────────────────┐
| □ Order Confirmed                |
| Order #ORD-789                   |
| Samsung A15 × 1                  |
| TZS 400,000 paid                 |
└──────────────────────────────────┘
```

Flow 2 — Seller Attaches from Inside Chat

```
Seller inside any conversation
     | taps attach (+)
     ▼
┌──────────────────────────────────┐
| Attach                           |
|                                  |
| □□Image                          |
| □□Voice Note                     |
| □□File                           |
| □□From My Shop  ◄─── this one    |
| □□Event                          |
| □□Group Purchase                 |
└──────────────────────────────────┘
     | taps "From My Shop"
     ▼
Seller browses their shop products
Picks product
Sets custom price for this buyer (optional)
     |
     ▼ Sends as card in chat
┌──────────────────────────────────┐
| □□TechStore Offer                |
| Samsung A15                      |
| TZS 400,000                      |
|                                  |
```

```
| Quantity: [− 1 +]               |
| [Proceed to Checkout →]         |
└─────────────────────────────────┘

     |
Buyer taps Proceed → checkout outside inbox
```

Commerce Message Types

```
message.type values for commerce:

PRODUCT_CARD          product shared in chat
CUSTOM_PRICE_OFFER    seller's private price for this buyer
EVENT_CARD            event shared in chat
GROUP_PURCHASE_CARD   Bei ya pamoja shared in chat
POST_CARD             VP Feed post shared in chat
ORDER_CONFIRMATION    system message after order placed
ORDER_STATUS_UPDATE   system message for shipping/delivery
PAYMENT_CONFIRMATION  system message after payment
```

---

8. Offline Delivery & Notification Escalation

Notification Levels

```
Message level resolved by Chat Service before fan-out:

CRITICAL:
  Order placed / payment received / payment failed
  Order status changed / delivery update
  → FCM HIGH + Textfy SMS simultaneously (no waiting)

IMPORTANT:
  Commerce DM from buyer
  Custom price offer received
  Bei ya pamoja threshold reached
```

```
  → FCM HIGH immediately
  → Textfy SMS after 10 minutes if no delivery ack

NORMAL:
  Regular DMs, group messages
  Social cards, post shares
  → FCM HIGH only
  → No SMS escalation
```

Textfy SMS Templates

```
Order notification (Swahili):
"NexGate: Agizo jipya kutoka [Buyer]!
 Kiasi: TZS [amount]. Kagua: nexgate.app/orders/[id]"

Payment received:
"NexGate: Malipo ya TZS [amount]
 yamepokelewa kutoka [Buyer].
 nexgate.app/wallet"

Commerce DM:
"NexGate: [Buyer] anakuuliza kuhusu
 [Product]. Jibu: nexgate.app/chat/[convId]"

Bei ya pamoja:
"NexGate: Watu [n]/[target] wamejiunga!
 nexgate.app/group-buy/[id]"
```

Notification Delivery Log

All notifications tracked for audit — critical for order disputes:

```
notification_log
─────────────────────────────────────
id                UUID
user_id           UUID
message_id        UUID
level             ENUM (NORMAL/IMPORTANT/CRITICAL)
fcm_status        ENUM (SENT/DELIVERED/FAILED)
```

```
sms_status        ENUM (SENT/DELIVERED/FAILED/SKIPPED)
sms_provider      TEXT
sent_at           TIMESTAMPTZ
delivered_at      TIMESTAMPTZ
opened_at         TIMESTAMPTZ
```

---

9. Message Status System

```
Client shows status via tick indicators:

 Pending      message on device, not yet sent
               (no connection)

✓  Sent         server received and persisted
               Gateway ACK returned with temp_id

✓✓ Delivered    recipient device received
               WS delivery ack received by Chat Service

✓✓ Read         recipient opened the conversation
               READ event sent by recipient app
               (blue ticks)

Flow:
[Send] → pending 
[Gateway ACK] → sent ✓
[Recipient WS ack] → delivered ✓✓
[Recipient opens conv] → read ✓✓ (blue)
```

Typing Indicators

```
Ephemeral — never persisted to PostgreSQL

User starts typing:
  App sends: { type: TYPING_START, convId }
  Chat Service sets Redis key:
    typing:{convId}:{userId} → TTL 5 seconds
```

```
  Redis pub/sub notifies conversation members
  Recipients see "Juma anaandika..."

User stops typing (or TTL expires):
  Key auto-expires after 5 seconds
  Chat Service notifies: typing stopped
  Indicator disappears
```

---

10. Database Schema

conversations

```
conversations
_____________________________________________

id                UUID          PK
type              ENUM          DM / GROUP / BROADCAST / COMMERCE
owner_type        ENUM          USER / SHOP
owner_id          UUID          userId or shopId
title             TEXT          for groups and broadcast channels
avatar_file_id    UUID          File Thunder fileId
status            ENUM          ACTIVE / ARCHIVED / BLOCKED
created_by        UUID          userId
created_at        TIMESTAMPTZ
```

conversation_members

```
conversation_members
_____________________________________________

conversation_id   UUID          FK → conversations
user_id           UUID
role              ENUM          MEMBER / ADMIN / OWNER
joined_at         TIMESTAMPTZ
last_read_at      TIMESTAMPTZ
last_read_seq     BIGINT        sequence ID (for gap detection)
is_muted          BOOLEAN
muted_until       TIMESTAMPTZ
```

messages

```
messages
──────────────────────────────────────────────────
id                UUID          PK
conversation_id   UUID          FK → conversations
sender_id         UUID
seq               BIGINT        monotonic per conversation
type              ENUM          TEXT / IMAGE / VIDEO / VOICE_NOTE /
                                FILE / PRODUCT_CARD / CUSTOM_PRICE_OFFER /
                                EVENT_CARD / GROUP_PURCHASE_CARD /
                                POST_CARD / ORDER_CONFIRMATION /
                                ORDER_STATUS_UPDATE / PAYMENT_CONFIRMATION /
                                SYSTEM
body              TEXT
media_ref         UUID          File Thunder fileId
context_type      ENUM          PRODUCT / ORDER / PAYMENT / EVENT / GROUP_PURCHASE
context_ref_id    UUID          ref to relevant entity
snapshot_json     JSONB         frozen context data at send time
reply_to_id       UUID          FK → messages (thread replies)
status            ENUM          SENT / DELIVERED / READ / FAILED
level             ENUM          NORMAL / IMPORTANT / CRITICAL
created_at        TIMESTAMPTZ
edited_at         TIMESTAMPTZ
deleted_at        TIMESTAMPTZ
```

message_receipts

```
message_receipts
──────────────────────────────────────────────────
message_id        UUID          FK → messages
user_id           UUID
status            ENUM          DELIVERED / READ
device_id         TEXT
timestamp         TIMESTAMPTZ
```

calls

```
calls
_____________________________________________________

call_id           UUID          PK
caller_id         UUID
receiver_id       UUID
conversation_id   UUID          FK → conversations
type              ENUM          VOICE / VIDEO
status            ENUM          RINGING / CONNECTED / COMPLETED /
                                MISSED / DECLINED / FAILED
started_at        TIMESTAMPTZ
answered_at       TIMESTAMPTZ
ended_at          TIMESTAMPTZ
duration_seconds  INT
relay_used        BOOLEAN       was TURN relay used?
end_reason        ENUM          NORMAL / NETWORK / TIMEOUT / DECLINED
```

shop_conversation_access

```
shop_conversation_access
_____________________________________________________

shop_id           UUID
user_id           UUID          staff member
role              ENUM          MANAGER / SUPPORT_AGENT / READ_ONLY
granted_by        UUID          owner userId
granted_at        TIMESTAMPTZ
revoked_at        TIMESTAMPTZ
```

---

11. Inbox Model Implementation

Conversation Ownership

```
Personal DM:
  owner_type = USER
  owner_id   = usr-kibuti
  Access: only usr-kibuti
```

```
Shop DM:
  owner_type = SHOP
  owner_id   = shop-techstore
  Access: anyone with role in shop_conversation_access
          for shop-techstore
```

Tab Resolution (App Side)

```
App fetches inbox tabs on load:
GET /chat/inbox/tabs

Response:
[
  { type: PERSONAL, label: "Personal", unread: 3 },
  { type: SHOP, shopId: "shop-techstore",
    label: "TechStore", unread: 12 },
  { type: SHOP, shopId: "shop-clothinghub",
    label: "ClothingHub", unread: 0 }
]

Chat Service resolves tabs by:
  1. User's own personal conversations
  2. All shops where user has role in
     shop_conversation_access
```

Access Control Check

```
Every request to open a shop conversation:

Does requesting user own the shop?
  YES → allow
  NO  → check shop_conversation_access:
          user_id = requester
          shop_id = conversation.owner_id
          revoked_at IS NULL
        Found? → allow with their role
        Not found? → 403 Forbidden
```

12. Local Experiments

These run in parallel with Phase 1 production work. All throwaway code — not NexGate quality. Goal is understanding, not production output.

Experiment 1 — Ejabberd Local Docker

```
Goal: get Ejabberd running, send one message

Steps:
  docker run -d --name ejabberd \
    -p 5222:5222 \
    -p 5280:5280 \
    -p 5285:5285 \
    ghcr.io/processone/ejabberd

  Open: http://localhost:5280/admin
  Create two test users: alice, bob

  Install Conversations app on Android
  Connect to localhost:5222 as alice
  Send message to bob
  See it arrive

What you learn:
  How Ejabberd config works
  What XMPP stanzas look like in logs
  How the dashboard shows connections
  What errors look like and how to fix them

Success: message delivered between two test users
```

XMPP Stanzas — What They Look Like

When Alice sends "Habari" to Bob, this travels over the wire:

```
<!-- Alice sends message to Bob -->
<message from="alice@localhost"
         to="bob@localhost"
```

```
          type="chat"
          id="msg-001">
    <body>Habari</body>
    <active xmlns="http://jabber.org/protocol/chatstates"/>
  </message>

  <!-- Bob's typing indicator back -->
  <message from="bob@localhost"
          to="alice@localhost"
          type="chat">
    <composing xmlns="http://jabber.org/protocol/chatstates"/>
  </message>

  <!-- Presence — Alice comes online -->
  <presence from="alice@localhost/android">
    <show>available</show>
    <status>Ninafanya kazi</status>
  </presence>

  <!-- Message delivery receipt (XEP-0184) -->
  <message from="bob@localhost" to="alice@localhost">
    <received xmlns="urn:xmpp:receipts"
              id="msg-001"/>
  </message>
```

These stanzas are what Ejabberd routes. In Phase 2, NexGate wraps its own data inside custom XMPP stanzas.

Experiment 2 — Spring Boot Auth Bridge

```
  Goal: Ejabberd calls Spring Boot to validate users

  Setup:
    Simple Spring Boot app (H2 in-memory DB)
    One endpoint: POST /internal/ejabberd/auth
      receives: { username, token }
      returns: 200 (allow) or 401 (deny)

    ejabberd.yml:
```

```
    auth_method: http
    auth_opts:
      url: "http://host.docker.internal:8080/internal/ejabberd/auth"

Test:
  Connect Conversations app to Ejabberd
  Ejabberd calls Spring Boot for auth
  Spring Boot validates → returns 200
  Connection allowed

What you learn:
  Auth flow between Ejabberd and Spring Boot
  How fast Spring Boot must respond (< 200ms)
  What happens when auth fails
  How to structure the internal endpoint

Success: Ejabberd rejects unknown users,
         allows users Spring Boot approves
```

Experiment 3 — WebRTC Voice Call on Emulators

```
Goal: voice call between two Android emulators

Setup:
  Two Android Studio emulators running
  Simple Android app — just two buttons:
    [Call] and [Answer]
  WebRTC library: io.getstream:stream-webrtc-android
  No UI — just logcat output

What to test:
  Create PeerConnection on both
  Exchange SDP offer/answer manually (copy-paste between logs)
  Exchange ICE candidates
  Hear audio between emulators

What you learn:
  How WebRTC PeerConnection works in practice
```

```
  What SDP looks like
  What ICE candidates look like
  How long negotiation actually takes
  What errors appear and how to fix them

Success: audio heard between two emulators
```

SDP offer example (what WebRTC generates):

```
v=0
o=- 123456 2 IN IP4 127.0.0.1
s=-
t=0 0
a=group:BUNDLE 0
m=audio 9 UDP/TLS/RTP/SAVPF 111
c=IN IP4 0.0.0.0
a=rtcp:9 IN IP4 0.0.0.0
a=ice-ufrag:someRandomString
a=ice-pwd:anotherRandomString
a=fingerprint:sha-256 AA:BB:CC:...
a=rtpmap:111 opus/48000/2    ← Opus codec negotiated here
a=fmtp:111 minptime=10;useinbandfec=1
```

Experiment 4 — Coturn TURN Relay

```
Goal: force audio through TURN relay, confirm it works

Setup:
  docker run -d --network=host coturn/coturn \
    -n --log-file=stdout \
    --min-port=49152 --max-port=65535 \
    --lt-cred-mech \
    --user=test:password123 \
    --realm=nexgate.com

Test:
  Disable P2P in WebRTC config (force TURN only)
  Run voice call experiment
  Confirm audio still flows through relay
```

```
What you learn:
  How Coturn logs relay connections
  Bandwidth used per call (check with iftop)
  How to generate HMAC credentials (not plain text)
  What happens when Coturn is unavailable

Success: audio heard between emulators via TURN relay
         Coturn logs show relay traffic
```

Experiment 5 — MessagePack Encoding

```
Goal: compare JSON vs MessagePack on same message

Simple Spring Boot test:
  Serialize same chat message object
  Once as JSON
  Once as MessagePack

Measure:
  Byte size comparison
  Serialization speed
  Deserialization speed

Expected result:
  JSON:        ~180-200 bytes per message
  MessagePack: ~60-70 bytes per message
  ~65% size reduction confirmed

Success: numbers prove EA bandwidth saving is real
```

Experiment Success Criteria Summary

```
Experiment 1 — Ejabberd local:
  □ Message delivered between two users
  □ Dashboard shows live connections

Experiment 2 — Auth bridge:
  □ Ejabberd rejects unknown tokens
```

```
    □ Ejabberd allows Spring Boot approved users

  Experiment 3 — WebRTC emulators:
    □ Audio heard between two emulators
    □ SDP and ICE flow understood

  Experiment 4 — Coturn relay:
    □ Audio heard via forced TURN relay
    □ Bandwidth per call measured

  Experiment 5 — MessagePack:
    □ Size reduction confirmed
    □ Serialization speed measured

  All five done → ready to write Phase 2 doc
                → ready to build NexGate chat Phase 2
```

Summary

Phase 1 is the foundation. It ships real features — text chat, voice notes, commerce DMs, offline delivery, the isolated shop inbox — using a clean Spring Boot WebSocket architecture that runs on existing infrastructure.

Everything designed here carries directly into Phase 2. The schema stays. The commerce logic stays. The notification system stays. The inbox model stays. Only the transport layer swaps — Spring Boot WS gateway out, Ejabberd in.

The five local experiments running in parallel are not wasted time. They are the insurance policy that makes Phase 2 a confident execution rather than a risky exploration. Every surprise Ejabberd, WebRTC, and Coturn have in store — you want to find them in a throwaway experiment, not in your production chat system.

NexGate Chat Platform — Phase 1: Foundation & Local Experiments v1.0 QBIT SPARK | Spring Boot WebSocket · Commerce DMs · Offline Delivery

---

Revision #1
Created 13 July 2026 06:36:33 by Admin Qbit
Updated 13 July 2026 06:36:53 by Admin Qbit