NexGate Chat System — Development Architecture Guide

QBIT SPARK | NexGate Platform *The complete reference for building the chat_system module inside nexgate_backend*

Table of Contents

---

1. Overview & Philosophy

What NexGate Chat Is

```
NexGate chat is NOT a standalone service.
It is a package inside nexgate_backend.

Three pillars of NexGate:
  VP Social   → social content
  VP Shop   → commerce
  VP Events → event management
  Chat      → the connective tissue between all three

Chat is deeply integrated with commerce.
Every conversation can become a transaction.
Every transaction has a conversation behind it.
```

The Key Principle

```
Ejabberd = the transport layer (moves stanzas)
Spring Boot = the business logic layer (decides what happens)
PostgreSQL = the persistence layer (stores everything)
RabbitMQ = the event bus (connects Ejabberd to Spring Boot)

Ejabberd knows NOTHING about:
  NexGate users
  Orders
  Shops
  Offers
  Commerce

Spring Boot knows EVERYTHING about:
  Who can talk to whom
```

```
  What messages mean
  Commerce context
  Offer lifecycle
  Notifications
```

Why Not a Separate Microservice

```
Chat is INSIDE nexgate_backend because:
  Shares user data constantly
    (display names, avatars, shop memberships)
  Shares order data
    (order confirmations, shipping updates)
  Shares product data
    (product cards, offer sessions)
  No distributed transactions needed
  Same team, same codebase
  Simple joins instead of API calls

Extract chat service later IF:
  Chat traffic overwhelms platform
  Separate team manages chat
  DB connections exhausted by chat
  NOT before — premature optimization
```

---

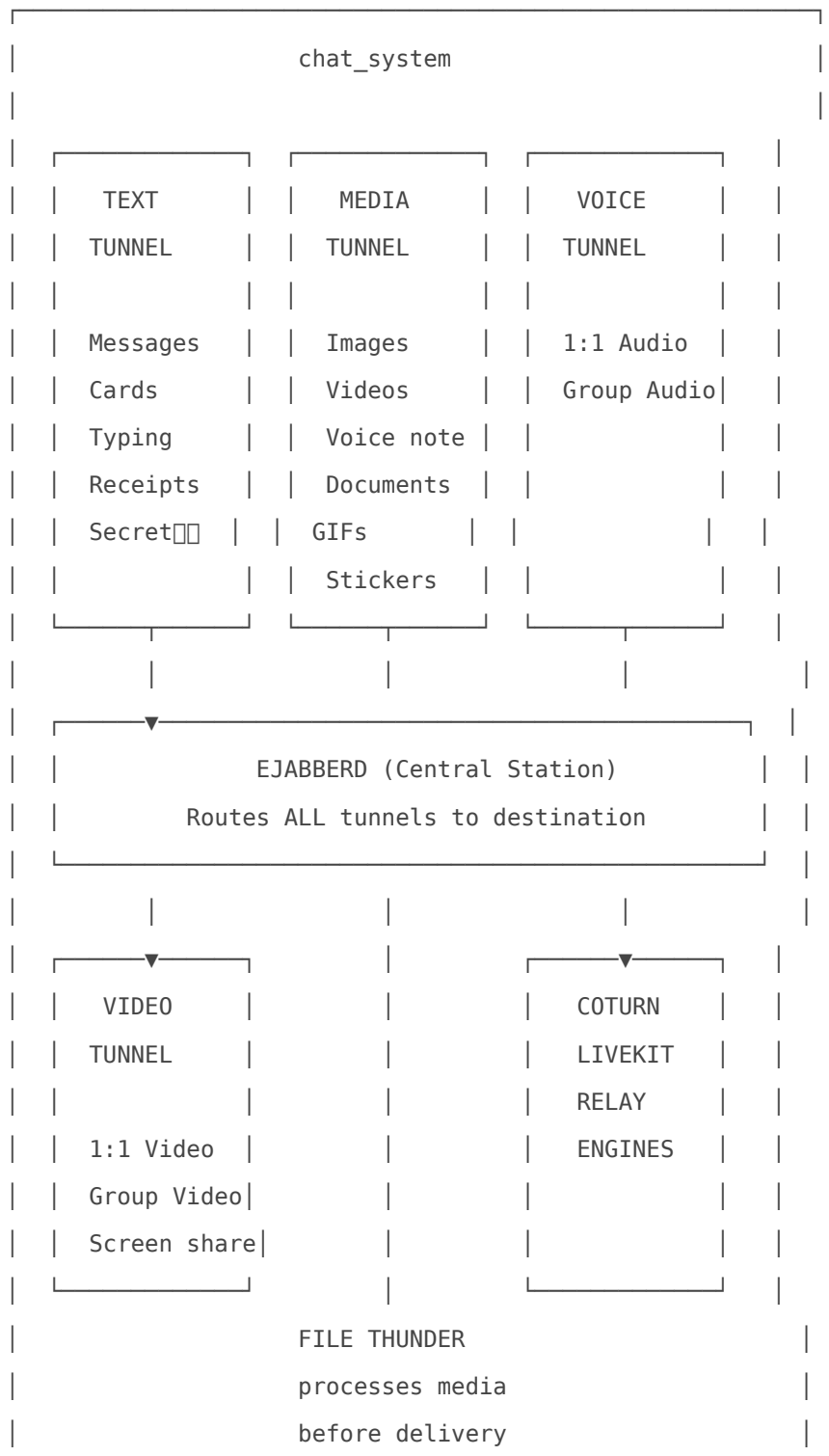
2. The Four Tunnels — How chat_system Works

Inspiration

```
Inspired by the Cu Chi tunnel system in Vietnam —
250km of specialized tunnels, each built for a
specific purpose, each carrying specific content,
all invisible to those above.

chat_system works the same way:
```

```
Four specialized tunnels
Each carries specific content type
Each powered by the right engine
All meet at the Central Station (Ejabberd)
All invisible to the network above
```

The Four Tunnels



```
└──────────────────────────────────────────────────────┘
```

Tunnel 1 — Text Tunnel

```
What it carries:
  Plain text messages
  Cards (all 6 categories)
  Typing indicators
  Delivery receipts (✓✓)
  Read receipts (□□)
  Reactions (□❤□□□)
  Message edits + deletes
  Presence (online/offline)
  Commerce stanzas
  Group invitations
  System notifications
  Secret Chat (encrypted) □□

Engine: Ejabberd (XMPP)
Protocol: XML stanzas over WebSocket/TCP
Port: 5222 with TLS

Security levels inside Text Tunnel:
  Level 1 — Normal: stored in DB (readable)
  Level 2 — Commerce: auditable, legal record
  Level 3 — Secret Chat: OMEMO encrypted
              nobody can read, not even NexGate
```

Tunnel 2 — Media Tunnel

```
What it carries:
  Images (JPEG, PNG, WebP)
  Videos (MP4, HLS)
  Voice notes (OGG/Opus)
  Documents (PDF, DOC etc)
  GIFs (MP4 loop)
  Stickers (WebP)
  Files (any type)
```

```
Engines:
  File Thunder → processes + stores + CDN
  Ejabberd → delivers the stanza with CDN URL

Flow:
  Sender uploads → File Thunder processes
  File Thunder → CDN URL
  Ejabberd stanza → delivers URL to recipient
  Recipient → downloads from CDN directly

  Ejabberd never touches the file itself
  Only delivers the URL stanza □
```

Tunnel 3 — Voice Tunnel

```
What it carries:
  1:1 voice calls
  Group voice calls (3+ people)

Engines:
  Ejabberd → Jingle signaling (call setup)
  WebRTC → audio capture + encoding (Opus)
  Coturn → relay when P2P blocked (EA NAT)
  LiveKit → group calls (3+ people only)

1:1 voice:
  Try P2P direct first
  If blocked (carrier NAT) → Coturn relay
  Engine: WebRTC + Coturn

Group voice (3+):
  Always LiveKit SFU
  No P2P attempt
  Each person uploads 1 stream
  LiveKit forwards to all others

Switching 1:1 → group mid-call:
  Spring Boot creates LiveKit room
```

```
  Sends join stanzas to all
  Old WebRTC terminates
  All join LiveKit seamlessly □
```

Tunnel 4 — Video Tunnel

```
What it carries:
  1:1 video calls
  Group video calls (3+ people)
  Screen sharing

Engines:
  Ejabberd → Jingle signaling
  WebRTC → video capture + H.264 encoding
  Coturn → relay when P2P blocked
  LiveKit → group calls (3+ people only)

1:1 video:
  Try WebRTC P2P first
  If blocked → Coturn relay
  Engine: WebRTC + Coturn

Group video (3+):
  Always LiveKit SFU
  Simulcast: 3 quality levels per stream
    Low (180p): 2G networks
    Medium (360p): 3G networks
    High (720p): 4G/WiFi
  Each receiver gets quality their network allows

Screen share:
  Android: MediaProjection API
  iOS: ReplayKit broadcast extension
  H.264 encoding, 5-15fps
  Works in both 1:1 and group calls

Switching 1:1 → group mid-call:
  Same as Voice Tunnel upgrade flow □
```

Why Not LiveKit for Everything?

```
Question: why not use LiveKit for 1:1 calls too?

WebRTC P2P (1:1):
  Alice → directly → Bob (no server)
  Zero extra latency □
  Zero server cost □
  Privacy: nobody in the middle □
  Faster connection □

LiveKit SFU (group):
  Alice → LiveKit server → Bob
  Extra server hop = more latency □
  Server cost per call □
  Server sees media □

Rule: P2P when 2 people (always faster + cheaper)
      SFU when 3+ people (P2P impossible on mobile)

Why P2P fails for groups:
  5 people P2P mesh:
    Each uploads 4 streams
    Each downloads 4 streams
    = 8 streams simultaneously
    = 8Mbps needed per person □
    EA 4G average = 5Mbps □
    Impossible on EA networks □

  5 people via LiveKit SFU:
    Each uploads 1 stream only □
    LiveKit forwards to others
    Feasible on EA 3G/4G □
```

The Engine Registry

```
Engine          Tunnel(s)             Role
_____________________________________________________
```

```
Ejabberd        ALL (signaling)        Central Station
                                       Routes everything
                                       XMPP stanzas

File Thunder    Media Tunnel           Process + store
                                       Virus scan
                                       CDN delivery

WebRTC          Voice + Video          Audio/video capture
                                       Encode/decode
                                       On-device processing

Coturn          Voice + Video          Relay station
                                       Bypasses EA NAT
                                       TURN protocol

LiveKit SFU     Voice + Video          Group calls (3+)
                (group only)           Simulcast engine
                                       One→many forwarding

OMEMO           Text Tunnel            Encryption engine
                (Secret Chat)          Keys on device only
                                       Level 3 security
```

Tunnel Security Levels

```
Inspired by Cu Chi tunnel depth levels:

Level 1 — Surface (Normal Chat):
  Regular conversations
  Stored in DB (Spring Boot can read)
  Used for: casual chat, group chat
  Engine: Ejabberd + PostgreSQL

Level 2 — Mid-depth (Commerce DMs):
  Business conversations
  Stored + auditable
  Legal record for disputes
  Staff audit log
```

```
  Used for: shop DMs, order updates
  Engine: Ejabberd + PostgreSQL (audit)

Level 3 — Deep (Secret Chat):
  End-to-end encrypted (OMEMO)
  Encrypted blob stored in DB
  NexGate CANNOT read content □
  Keys NEVER leave device □
  Used for: private personal conversations
  Engine: OMEMO + Ejabberd (transport only)
```

---

3. Inbox Model — Three Separate Inboxes

The Three Inboxes

```
NexGate has THREE completely separate chat inboxes:

  Inbox 1 — Personal:
    1:1 DMs with other users (@username)
    Group chats (friends, community)
    Accessed via: Chat → Personal tab

  Inbox 2 — Commerce:
    DMs with shops ($tag)
    Order updates from shops
    Offer sessions
    Accessed via: Chat → Commerce tab

  Inbox 3 — Secret □□
    End-to-end encrypted 1:1 conversations
    Completely separate from personal chats
    Handled like Telegram Secret Chat model:
      Long press contact → Start Secret Chat
      Opens parallel secret conversation
      Same contact can have both normal + secret
```

```
    Accessed via: Chat → Secret tab
    OR: lock icon □□entry point from contact profile

Why three separate?
  Personal: social conversations (friends, groups)
  Commerce: transactional (shops, orders)
  Secret: private E2EE (fully isolated)

  User never loses personal chats in orders □
  User never accidentally sends to wrong chat □
  Clean mental model □
  Telegram model for Secret = familiar □
```

Conversation Types

```
DIRECT:
  Between two @users
  Normal 1:1 conversation
  Inbox: Personal
  Security: Level 1

GROUP:
  Multiple @users
  Created explicitly
  Inbox: Personal
  Security: Level 1

DIRECT_SECRET:
  Between two @users
  End-to-end encrypted (OMEMO)
  Parallel to normal DIRECT (same contact)
  Inbox: Secret □□
  Security: Level 3
  Telegram model: contact has both DIRECT + DIRECT_SECRET

COMMERCE_DM:
  Between @user and $shop
  Initiated by either side
  Contains: product cards, offers, order updates
```

```
  Inbox: Commerce
  Security: Level 2 (auditable)

SYSTEM:
  Auto-generated per order
  No human on shop side (automated)
  Order confirmation, shipping updates
  Inbox: Commerce
  Security: Level 2 (immutable)

NOTE: Bei Ya Pamoja is NOT a conversation type
  It is a CARD (GROUP_BUY) that can be shared in:
    DIRECT conversations □
    GROUP conversations □
    COMMERCE_DM conversations □
    NOT in DIRECT_SECRET □
```

How Conversations Are Created

```
DIRECT — user taps on @username → start chat:
  POST /chat/conversations/direct
  { targetUserId }
  Spring Boot: find existing OR create new
  type = DIRECT

GROUP — user creates a group:
  POST /chat/groups/create
  { name, type: PRIVATE|PUBLIC }
  Spring Boot: create + Ejabberd room
  type = GROUP

DIRECT_SECRET — long press contact → Start Secret Chat:
  POST /chat/conversations/secret
  { targetUserId }
  Spring Boot: find existing OR create new secret thread
  type = DIRECT_SECRET
  App: initializes OMEMO session with target's public keys

COMMERCE_DM — buyer taps [Chat with Shop]:
```

```
  POST /chat/conversations/commerce
  { shopId }
  Spring Boot: find existing OR create new
  Ejabberd: uses shop JID (shop-456@nexgate.com)
  type = COMMERCE_DM

COMMERCE_DM — order placed (auto):
  Spring Boot order service triggers:
  CommerceDmService.getOrCreateCommerceThread(userId, shopId)
  First message auto-sent: "Your order ORD-789 confirmed"
  type = COMMERCE_DM

SYSTEM — one per order (auto):
  Spring Boot: createSystemConversation(orderId)
  Only system messages — user cannot reply
  type = SYSTEM
```

Inbox API — What Mobile Calls

```
Personal inbox:
  GET /chat/conversations?type=personal
  Returns: DIRECT + GROUP conversations
  Sorted by: last_message DESC

Commerce inbox:
  GET /chat/conversations?type=commerce
  Returns: COMMERCE_DM + SYSTEM conversations
  Sorted by: last_message DESC
  Grouped by: shop (all TechStore threads together)

Secret inbox:
  GET /chat/conversations?type=secret
  Returns: DIRECT_SECRET conversations
  Sorted by: last_message DESC
  Note: content previews NOT shown (encrypted)
        Shows: contact name + "Secret Chat □□ only

Unread counts:
  GET /chat/conversations/unread-counts
```

```
  Returns:
    { personal: 3, commerce: 7, secret: 1 }
  Source: Redis (never DB query)
  Used for: tab badges in UI
  Secret tab badge: shows count only (no preview)
```

Shop Side — Shop Inbox

```
Shop staff logs into NexGate shop dashboard:
  Sees: ALL conversations with customers
  Sorted by: unread first, then recent

Shop inbox API:
  GET /chat/shop/{shopId}/conversations
  Auth: requires shop staff JWT
  Returns: all COMMERCE_DM for this shop
  Shows: customer name, last message, unread

Staff replies:
  POST /chat/conversations/{convId}/messages
  { body, attachments }
  Sent as: shop-456@nexgate.com (no staff name) □

Shop cannot see:
  Personal conversations □
  Secret conversations □
  Other shops' conversations □
```

JID Mapping Per Conversation Type

```
DIRECT (alice → juma):
  Alice JID:  usr-alice@nexgate.com
  Juma JID:   usr-juma@nexgate.com
  Ejabberd:   direct XMPP stanza □

DIRECT_SECRET (alice → juma, encrypted):
  Alice JID:  usr-alice@nexgate.com
  Juma JID:   usr-juma@nexgate.com
```

```
  Ejabberd:   same routing as DIRECT □
              BUT: payload is OMEMO encrypted
              Ejabberd cannot read content □
  Separate conversation ID from DIRECT

GROUP (warriors group):
  Room JID:   conv-abc123@conference.nexgate.com
  Members:    usr-alice, usr-juma, usr-enkiama
  Ejabberd:   MUC room □

COMMERCE_DM (alice → TechStore):
  Alice JID:  usr-alice@nexgate.com
  Shop JID:   shop-456@nexgate.com
  Staff:      shop-456@nexgate.com/staff-1 (internal)
  Customer sees: shop-456@nexgate.com □

SYSTEM (order notification):
  From JID:   system@nexgate.com
  To JID:     usr-alice@nexgate.com
  type:       headline (no offline storage trigger)
  Spring Boot sends via Ejabberd REST
```

Conversation Routing Rules

```
Message arrives from: usr-juma@nexgate.com (normal):
  → Check: is this DIRECT or DIRECT_SECRET?
  → Spring Boot checks conversation metadata
  → Route to correct conversation type
  → Inbox: Personal (DIRECT) or Secret (DIRECT_SECRET)

Message arrives from: usr-juma@nexgate.com (OMEMO):
  → Has OMEMO encryption element □
  → Route to DIRECT_SECRET conversation
  → Inbox: Secret □□

Message arrives from: shop-456@nexgate.com:
  → Route to COMMERCE_DM □
  → Inbox: Commerce
```

```
Message arrives from: system@nexgate.com:
  → Route to SYSTEM conversation for that order
  → Inbox: Commerce

Message arrives from: conv-abc123@conference.nexgate.com:
  → Route to GROUP conversation □
  → Inbox: Personal
```

---

4. Cards — Rich Content Type

What Cards Are

```
Cards are rich interactive content blocks
that travel inside the Text Tunnel.

Like a specialized cargo train
running inside the text tunnel:
  Same tunnel (Ejabberd routes it) □
  Same transport (XMPP stanza) □
  Different cargo (rich structured data) □
  Different rendering (UI card) □

Every card:
  Has a custom xmlns (identifies it) □
  Has a type field (what kind exactly) □
  Has a <body> fallback (for basic clients) □
  Has action buttons (1-3 max) □
  Renders as visual card in ChatBox □
```

The Six Card Categories

```
Category 1 — Social Cards
  Source: VP Social
  xmlns: urn:nexgate:social:1
  Types:
    POST_CARD      → feed post shared in chat
```

```
    REEL_CARD      → short video shared
    LIVE_CARD      → live stream invite
    PROFILE_CARD   → user profile share

Category 2 — Commerce Cards
  Source: VP Shop
  xmlns: urn:nexgate:commerce:1
         urn:nexgate:offer:1
         urn:nexgate:groupbuy:1
         urn:nexgate:installment:1
  Types:
    PRODUCT_CARD       → single product
    SHOP_CARD          → shop profile
    FLASH_SALE_CARD    → time-limited sale
    INSTALLMENT_PLAN   → buy now pay later

    CUSTOM_PRICE_OFFER → private negotiated price
      item_type: PRODUCT → offer on a product
      item_type: TICKET  → offer on an event ticket
      (item_type field inside <item> determines rendering)

    GROUP_BUY          → Bei ya pamoja card
      item_type: PRODUCT → group buy of a product
      item_type: TICKET  → group buy of event tickets
      Can be shared: in 1:1 DM, group chat, commerce DM
      NOT a group feature — it is a CARD □

Category 3 — Event Cards
  Source: VP Events
  xmlns: urn:nexgate:event:1
  Types:
    EVENT_CARD         → event details
    TICKET_CARD        → purchased ticket
    EVENT_REMINDER     → reminder notification

Category 4 — System Cards
  Source: Spring Boot (automated, no human)
  xmlns: urn:nexgate:system:1
  Types:
```

```
    ORDER_CONFIRMATION    → order placed □
    ORDER_STATUS_UPDATE   → shipped/delivered
    PAYMENT_CONFIRMATION  → payment received
    REFUND_CARD           → refund processed
    DISPUTE_CARD          → dispute opened

Category 5 — Group Cards
  Source: chat_system
  xmlns: urn:nexgate:group:1
  Types:
    GROUP_INVITATION → join group request
                       user sees card, taps Accept/Decline
                       this is the ONLY group CARD □

Category 6 — Call Signals
  Source: chat_system (NOT cards — system signals)
  xmlns: urn:nexgate:call:1
  Note: These are NOT interactive cards
        They are system signals that trigger UI states
        (incoming call screen, missed call indicator etc)
  Types:
    CALL_INITIATED       → triggers incoming call screen
    CALL_ACCEPTED        → call connected
    CALL_DECLINED        → other side declined
    CALL_ENDED           → call finished
    CALL_MISSED          → nobody answered
    GROUP_CALL_JOIN_INFO → join info for group call
    PARTICIPANT_ADDED    → someone added mid-call
    HOST_MUTED_YOU       → system: you were muted
    HOST_REMOVED_YOU     → system: you were removed

Category 7 — API Template Cards (Future)
  Source: Third-party developers via API
  xmlns: urn:nexgate:template:1
  Types:
    TRANSACTIONAL  → order/delivery updates
    MARKETING      → promotions (opt-in only)
    AUTHENTICATION → OTP, verification
```

Card Anatomy

```
Every card follows this structure:


  ┌──────────────────────────────────────┐
  │ HEADER                               │
  │   Category icon + source label       │
  │   e.g. □□□VP Shop                    │
  │──────────────────────────────────────│
  │ BODY                                 │
  │   Card-specific content              │
  │   Image, title, price, progress etc  │
  │──────────────────────────────────────│
  │ FOOTER                               │
  │   Action buttons (1-3 max)           │
  │   Status indicator                   │
  └──────────────────────────────────────┘


Every card also carries a shareable flag:
  shareable=true  → Forward option shown in UI □
  shareable=false → Forward option hidden □
                    Cannot be forwarded via API □
  Set by: Spring Boot (backend decides)
  Enforced by: Mobile UI + Spring Boot API

Example — Product Card:
  ┌──────────────────────────────────────┐
  │ □□□VP Shop                           │
  │──────────────────────────────────────│
  │ [product image]                      │
  │ Samsung A15                          │
  │ TZS 450,000 · TechStore              │
  │ Stock: 12 units                      │
  │──────────────────────────────────────│
  │ [View Product]    [Chat with Shop]   │
  └──────────────────────────────────────┘


Example — Order Confirmation:
  ┌──────────────────────────────────────┐
```

```
  | ⚙ Order Update                    |
  |-----------------------------------|
  | ✓ Order Confirmed                 |
  | ORD-789 · Samsung A15             |
  | TZS 400,000 · M-PESA              |
  |-----------------------------------|
  | [Track Order]                     |
  └-----------------------------------┘
```

Example — Group Buy:

```
  ┌-----------------------------------┐
  | 👥Group Purchase                  |
  |-----------------------------------|
  | Samsung A15                       |
  | Public: TZS 450,000               |
  | Group: TZS 350,000 (10 people)    |
  | ▓▓▓▓▓▓▓▓▒▒  8 / 10 joined         |
  | Expires: 2h 30m                   |
  |-----------------------------------|
  | [Join Group Buy]                  |
  └-----------------------------------┘
```

Shareable Flag — Default Values Per Card

Card	shareable	Reason
PRODUCT_CARD	true ✅	share to discover
SHOP_CARD	true ✅	share to discover
FLASH_SALE_CARD	true ✅	share = more buyers
INSTALLMENT_PLAN	false ❌	personal finance
CUSTOM_PRICE_OFFER	false ❌	private negotiation
		NEVER shareable
GROUP_BUY	true ✅	share = more joiners
EVENT_CARD	true ✅	invite others
TICKET_CARD	false ❌	personal ticket
EVENT_REMINDER	false ❌	personal reminder
ORDER_CONFIRMATION	false ❌	personal order
ORDER_STATUS_UPDATE	false ❌	personal order
PAYMENT_CONFIRMATION	false ❌	financial record

```
REFUND_CARD                 false □     financial record
DISPUTE_CARD                false □     private dispute
GROUP_INVITATION            false □     personal invite
POST_CARD                   true □      social sharing
REEL_CARD                   true □      social sharing
LIVE_CARD                   true □      invite others
PROFILE_CARD                true □      share contact
TEMPLATE (TRANSACTIONAL)    false □     personal
TEMPLATE (MARKETING)        false □     personal
TEMPLATE (AUTH/OTP)         false □     NEVER share OTP
```

Card Stanza Structure

```
All cards use the unified <ng> stanza standard.
See Section 15 — NexGate Stanza Standard
for complete stanza examples per card type.

Key rule:
  Every card carries <ng xmlns="urn:nexgate:1">
  with <meta> + <payload>
  card_type field inside <payload> identifies the card
  shareable field inside <meta> controls forwarding
```

Mobile Card Decision Tree

```
Message arrives:
  1. Find <ng xmlns="urn:nexgate:1"> element
  2. Read <meta> → message_type = CARD
  3. Read <payload> → check card_type field:

  card_type = POST_CARD    → post preview + [View Post]
  card_type = LIVE_CARD    → live badge + [Watch Live]
  card_type = REEL_CARD    → reel preview + [Watch]
  card_type = PROFILE_CARD → profile card + [Follow]

  card_type = PRODUCT_CARD → product + [View] [Chat]
  card_type = SHOP_CARD    → shop profile + [Visit]
  card_type = FLASH_SALE_CARD → sale card + [Buy Now]
```

```
  card_type = CUSTOM_PRICE_OFFER:
    status=PENDING  → offer + timer + [Accept] [Decline]
    status=ACCEPTED → "Offer Accepted □"
    status=DECLINED → "Offer Declined"
    status=EXPIRED  → "Offer Expired □"

  card_type = GROUP_BUY:
    → progress bar + members preview + [Join]
    → check item_type: PRODUCT or TICKET

  card_type = EVENT_CARD   → event + [View] [Get Ticket]
  card_type = TICKET_CARD  → ticket QR code + [View]

  card_type = ORDER_CONFIRMATION  → order + [Track]
  card_type = ORDER_STATUS_UPDATE → status badge
  card_type = PAYMENT_CONFIRMATION → payment details

  card_type = GROUP_INVITATION → [Accept] [Decline]

  card_type = CALL_INITIATED → incoming call screen
  card_type = GROUP_CALL_JOIN_INFO → [Join] [Decline]
  card_type = CALL_MISSED → missed call indicator

  card_type = TRANSACTIONAL → template card
  card_type = MARKETING → template + unsubscribe

  No card_type match?
    → render <body> fallback as plain text □
```

---

5. Content Types & User Boxes

The Five Content Types

```
chat_system delivers exactly 5 content types:

Type 1 — Text Plain
  What: pure text messages
```

```
  Tunnel: Text Tunnel
  Engine: Ejabberd
  Example: "Hello Juma!"

Type 2 — Cards
  What: rich interactive content blocks
  Tunnel: Text Tunnel (same engine)
  Engine: Ejabberd
  6 categories: Social, Commerce, Event,
                System, Group, Template
  Example: Product card, Order update

Type 3 — Media Bundle
  What: files users share
  Tunnel: Media Tunnel
  Engine: File Thunder + Ejabberd
  Types: Image, Video, Voice note,
         Document, GIF, Sticker

Type 4 — Audio Stream
  What: real-time voice
  Tunnel: Voice Tunnel
  Engine: WebRTC + Coturn (1:1)
          LiveKit SFU (group)
  Includes: 1:1 voice, group voice

Type 5 — Video Stream
  What: real-time video
  Tunnel: Video Tunnel
  Engine: WebRTC + Coturn (1:1)
          LiveKit SFU (group)
  Includes: 1:1 video, group video, screen share
```

Three User-Facing Boxes

```
Box 1 — Regular ChatBox:
  What it shows: normal conversation thread
  Security: Level 1 (casual) + Level 2 (commerce)
  Delivers:
```

```
    Type 1: Text Plain → message bubbles
    Type 2: Cards → rich interactive blocks
    Type 3: Media Bundle → image/video/voice player
  Conversations: DMs, Groups, Commerce DMs

Box 2 — Secret ChatBox □□
  What it shows: E2EE conversation thread
  Security: Level 3 (OMEMO encrypted)
  Different UI treatment:
    Lock icon in header □□
    "End-to-end encrypted" banner
    No screenshots (Android FLAG_SECURE)
    Self-destruct timer (optional) □□
  Delivers:
    Type 1: Text Plain (encrypted)
    Type 3: Media Bundle (encrypted)
    NOT: Cards (commerce requires server-side logic)
    NOT: Call Signals (calls not E2EE via OMEMO)
  Conversations: Secret Chat 1:1 only

Box 3 — CallBox:
  What it shows: call screen
  Delivers:
    Type 4: Audio Stream → voice call UI
    Type 5: Video Stream → video call UI

  1:1 CallBox:
    Two participants
    Microphone, camera, hang up controls
    Audio ↔ Video switch (no hang up needed)

  Group CallBox:
    Grid of participant tiles
    Raise hand □
    Emoji reactions □□□□□
    Host controls (mute, remove)
    Pin participant
    Speaker/grid layout switch
```

Call Actions — Who Handles What

```
Action                  Handler
──────────────────────────────────────────────────────

Mute/unmute mic         Local WebRTC (device)
Camera on/off           Local WebRTC (device)
Switch front/back cam   Local WebRTC (device)
Speaker/earpiece        Local OS audio
Hang up                 Jingle session-terminate

Raise hand □            LiveKit metadata broadcast
Lower hand              LiveKit metadata broadcast
Emoji reaction □□      LiveKit sendData() broadcast
Pin participant         Local UI only (no server)
Switch grid/speaker     Local UI only (no server)
Disable incoming video  Local WebRTC (save data)

Mute participant (host) LiveKit SDK + Spring Boot auth
Remove participant      LiveKit SDK + Spring Boot auth
Add participant mid-call Spring Boot (new LiveKit room)
                        → sends join stanzas to all
                        → old WebRTC terminates
                        → all migrate to LiveKit

Audio → Video switch    Jingle content-add
Video → Audio switch    Jingle content-remove
Screen share start      Jingle content-add (screen)
Screen share stop       Jingle content-remove (screen)
```

1:1 ? Group Call Upgrade Flow

```
Alice + Bob in 1:1 WebRTC call
Alice taps "Add Participant" → selects Juma

Spring Boot:
  1. Creates new LiveKit room
  2. Generates tokens for Alice, Bob, Juma
  3. Sends ng stanza to Alice:
       message_type: CALL
```

```
        signal_type: GROUP_CALL_JOIN_INFO
        reason: UPGRADED_FROM_P2P
        livekit_token: eyJ-alice...
        See Section 15 Example 11 □
  4. Sends same to Bob (his token)
  5. Sends incoming call to Juma (his token)

Alice + Bob apps:
  Receive stanza
  Terminate WebRTC P2P
  Connect to LiveKit room
  Seamless — no hang up needed □

Juma's app:
  Incoming group call notification
  [Join] button
  Joins LiveKit room □

Result: 3-way group call on LiveKit □
```

---

6. Where Chat Lives in the Codebase

Package Structure

```
com.nexgate.backend
  ├── auth/              ← JWT, XMPP token issuance
  ├── feed/              ← VP Social
  ├── shop/              ← VP Shop
  ├── events/            ← VP Events
  ├── notification/      ← FCM, Textfy SMS
  │
  └── chat/              ← EVERYTHING CHAT RELATED
        ├── config/      ← Ejabberd config, RabbitMQ config
        ├── controller/  ← REST endpoints for mobile
        ├── service/     ← Business logic
        ├── consumer/    ← RabbitMQ event consumers
        ├── ejabberd/    ← Ejabberd REST client + auth bridge
```

```
        ├── stanza/      ← Custom XMPP stanza builders
        ├── commerce/    ← Commerce DM flows
        ├── call/        ← Call management (TURN credentials)
        └── model/       ← Chat domain models (NO DB entities here)
```

Database

```
nexgate_postgres (port 5432)
  ├── schema: core    ← nexgate_backend owns
  │     users, shops, products, orders
  │
  └── schema: chat    ← chat package owns
        conversations
        messages
        message_reactions
        message_receipts
        calls
        offer_sessions
        group_invite_links
        notification_log
```

---

7. The Three Communication Channels

Channel 1 — Spring Boot ? Ejabberd (REST API)

```
Direction:  Spring Boot calls Ejabberd
Protocol:   HTTP REST (port 5280)
Auth:       Admin credentials (stored in HashiCorp Vault)
Used for:   Sending stanzas, creating rooms, managing members

Examples:
  Send product card to buyer
  Create group room on event ticket purchase
  Kick user from shop group
```

```
  Send order confirmation stanza

Flow:
  Spring Boot
    → POST http://ejabberd:5280/api/send_message
    → Authorization: Basic admin@nexgate.com:adminpass
    → Body: { from, to, body + custom stanza }
    → Ejabberd routes to recipient □
```

Channel 2 — Ejabberd ? Spring Boot (RabbitMQ)

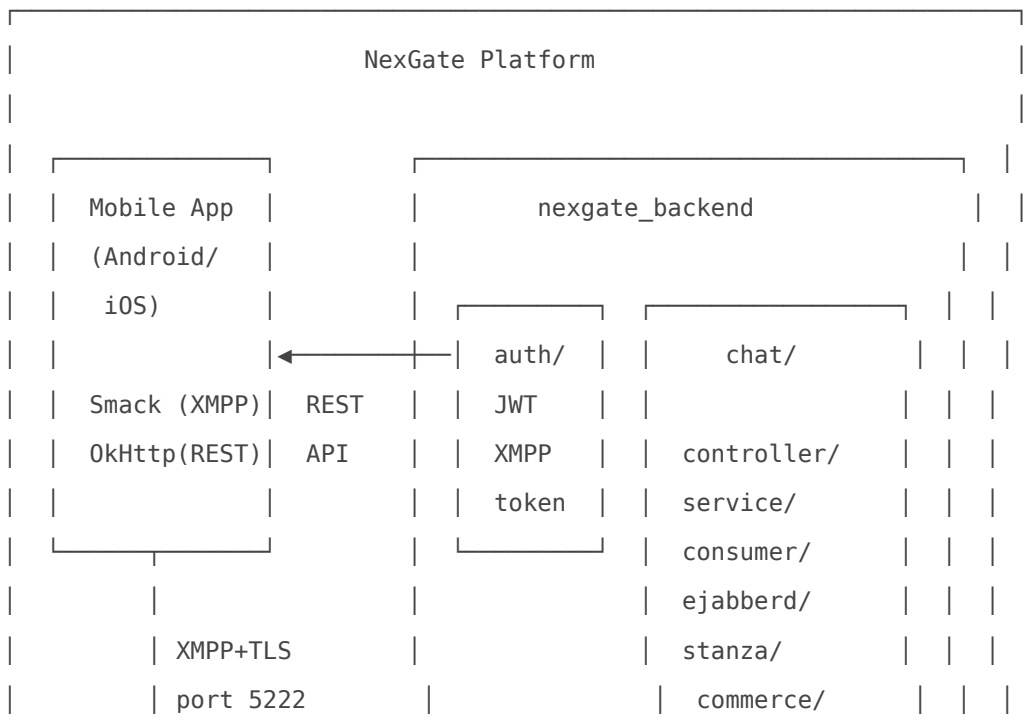
```
Direction:  Ejabberd fires events → Spring Boot consumes
Protocol:   RabbitMQ AMQP
Auth:       RabbitMQ credentials (stored in Vault)
Used for:   Knowing when messages arrive, presence changes, calls

Examples:
  Message delivered → Spring Boot persists to DB
  User goes offline → Spring Boot updates last seen
  Call started → Spring Boot creates call record
  Message read → Spring Boot updates receipt status

Flow:
  Ejabberd (mod_rabbitmq)
    → publishes to exchange: nexgate.chat
    → routing key: chat.message.inbound
    → Spring Boot consumer picks up
    → persists + triggers notifications □
```

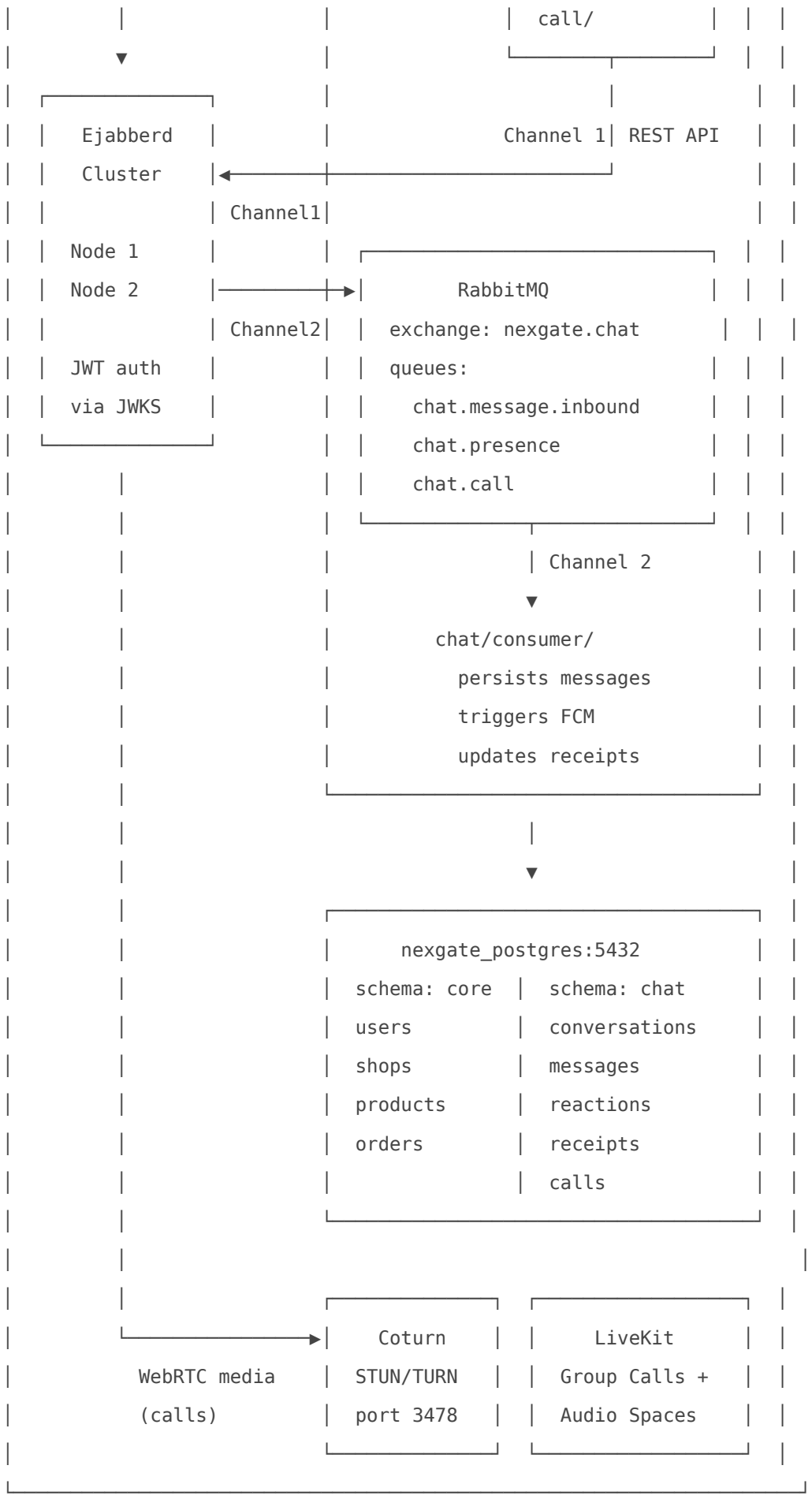
Channel 3 — Mobile ? Ejabberd ? Spring Boot (JWT)

```
Direction:  Mobile connects to Ejabberd
            Ejabberd validates via Spring Boot JWT
Protocol:   XMPP over WebSocket/TCP (port 5222) with TLS
Auth:       JWT token issued by Spring Boot on login
```

```
Used for:   User XMPP sessions (all chat activity)

Flow:
  1. Mobile → POST /auth/login (Spring Boot)
     Spring Boot validates credentials
     Issues TWO tokens:
       REST JWT  → for API calls (7 days)
       XMPP JWT  → for Ejabberd (24 hours)

  2. Mobile → Ejabberd port 5222
     Username: usr-kibuti@nexgate.com
     Password: <XMPP JWT>

  3. Ejabberd validates JWT locally
     Uses public key from JWKS endpoint:
       GET https://api.nexgate.com/auth/.well-known/jwks.json
     Verifies signature + expiry
     Allows connection □

  4. User is in — can send/receive stanzas
```

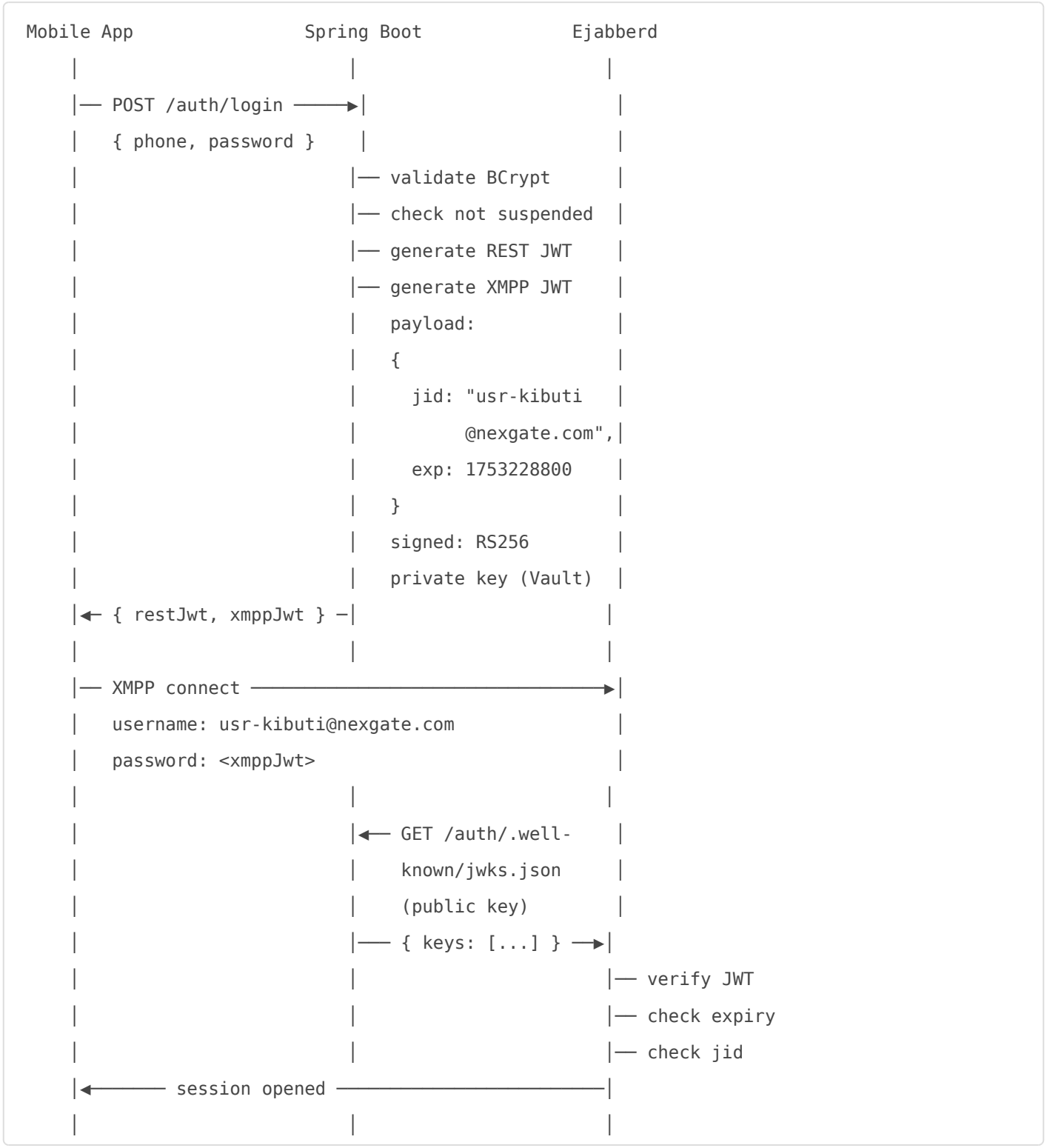
8. Full System Diagram




call/
Ejabberd
Cluster
Channel 1
REST API
Channel1
Node 1
Node 2
RabbitMQ
Channel2
JWT auth
via JWKS
exchange: nexgate.chat
queues:
chat.message.inbound
chat.presence
chat.call
Channel 2
chat/consumer/
persists messages
triggers FCM
updates receipts
nexgate_postgres:5432
schema: core
users
shops
products
orders
schema: chat
conversations
messages
reactions
receipts
calls
Coturn
STUN/TURN
port 3478
LiveKit
Group Calls +
Audio Spaces
WebRTC media
(calls)

9. Authentication Flow

Login + XMPP Token Issuance



JWT Key Management (Multi-Node)

```
Spring Boot:
  Private key → stored in HashiCorp Vault
  Used to SIGN XMPP JWTs
  Never shared

Ejabberd (all nodes):
  Public key → fetched from JWKS endpoint
  GET https://api.nexgate.com/auth/.well-known/jwks.json
  Used to VERIFY JWTs
  All nodes fetch same endpoint 

Key rotation:
  Spring Boot generates new key pair
  Adds both old + new to JWKS endpoint
  Issues new tokens with new key
  Old tokens valid until expiry (24h)
  After 24h → remove old key
  Zero downtime rotation 
```

---

10. Message Flow — 1:1 Chat

Mobile Sends Message

```
Mobile (Alice)           Ejabberd              Spring Boot
    |                       |                      |
    |── XMPP stanza ──────►|                      |
    |   <message            |                      |
    |     to="bob@nexgate.com"|                    |
    |     type="chat"       |                      |
    |     id="msg-001">     |                      |
    |     <body>Hello</body> |                     |
    |     <origin-id        |                      |
    |       xmlns="urn:xmpp:sid:0"                 |
    |       id="msg-001"/>  |                      |
```

```
    |     <request           |                      |
    |       xmlns="urn:xmpp:receipts"/>             |
    |     <markable          |                      |
    |       xmlns="urn:xmpp:chat-markers:0"/>       |
    |   </message>           |                      |
    |                        |                      |
    |                        |── route to Bob ─────►|(if offline)
    |                        |   (if online → deliver directly)
    |                        |                      |
    |                        |── RabbitMQ event ───►|
    |                        |   {                  |
    |                        |     event: "message",|
    |                        |     from: "alice",   |
    |                        |     to: "bob",       |
    |                        |     stanza_id: "...",|
    |                        |     body: "Hello",   |
    |                        |     timestamp: ...   |
    |                        |   }                  |
    |                        |                      |── persist to DB
    |                        |                      |── if Bob offline:
    |                        |                      |   send FCM
    |                        |                      |
    |◄── stream ACK ─────────|                      |
    |    <a h="N"            |                      |
    |      xmlns="urn:xmpp:sm:3"/>                  |
```

Link Handling (Before Send)

```
When message contains a URL:
  1. Client detects URL before sending
  2. Calls Link Safety Service (separate service):
       safe: true  → allow send □
       safe: false → block, warn user ⚠□
       "This link may be unsafe"
  3. NexGate internal links (nexgate.com/...):
       → converted to Card automatically
       → no safety check needed (we own these)
  4. External links → sent as plain text URL in body
       Client fetches Open Graph preview independently
```

```
      (client side — no server involvement)
      Sender sees preview before sending
      Receiver app fetches OG independently

  Link Safety Service documented separately.
```

What Spring Boot Does With Event

```
RabbitMQ consumer receives:
  1. Find or create conversation record
  2. Insert message into chat.messages
  3. Detect message type:
     - plain text → just persist
     - commerce stanza → trigger offer flow
     - system stanza → update order record
  4. Check if recipient online:
     - Online → delivery handled by Ejabberd
     - Offline → send FCM via notification/
  5. Update conversation.last_message
  6. Update conversation.updated_at
```

Message Interactions (Stanza Reference)

```
Edit (XEP-0308):
  <message id="edit-002">
    <body>corrected text</body>
    <replace xmlns="urn:xmpp:message-correct:0"
             id="original-msg-id"/>
  </message>
  Spring Boot: update messages.body, set edited_at

Delete (XEP-0424):
  <message id="retract-001">
    <apply-to xmlns="urn:xmpp:fasten:0"
              id="original-msg-id">
      <retract xmlns="urn:xmpp:message-retract:0"/>
    </apply-to>
    <body>/me retracted a message</body>
  </message>
```

```
  Spring Boot: set messages.deleted_at, body = null

React (XEP-0444):
  <message>
    <reactions xmlns="urn:xmpp:reactions:0"
               id="original-msg-id">
      <reaction>□□/reaction>
    </reactions>
  </message>
  Spring Boot: upsert message_reactions

Reply (XEP-0461):
  <message>
    <body>Thanks!</body>
    <reply xmlns="urn:xmpp:reply:0"
           to="alice@nexgate.com"
           id="original-msg-id"/>
  </message>
  Spring Boot: set messages.reply_to_id

Forward (XEP-0297):
  <message>
    <body>Check this</body>
    <ng xmlns="urn:nexgate:1">
      <meta>
        <message_type>TEXT</message_type>
        <shareable>true</shareable>
        ...
      </meta>
      <payload/>
    </ng>
    <forwarded xmlns="urn:xmpp:forward:0">
      <message from="..." to="...">
        <body>original message</body>
      </message>
    </forwarded>
  </message>
  Note: Spring Boot checks shareable=true before allowing
        Returns 403 if shareable=false □
```

11. Message Flow — Group Chat

Key Differences From 1:1

```
1:1 chat:                    Group chat:
  to = person JID              to = room JID
  type = "chat"                type = "groupchat"
  stanza-id by client          stanza-id by ROOM
  receipts per person          no per-person receipts
  reactions ref origin-id      reactions ref stanza-id

Group JID format:
  {conversationId}@conference.nexgate.com
  e.g. conv-abc123@conference.nexgate.com
```

Group Creation Flow

```
Mobile creates group:
  POST /chat/groups/create
  { name, type: PRIVATE|PUBLIC, description }

Spring Boot:
  1. Create conversation record (type=GROUP)
  2. Call Ejabberd REST:
     POST /api/create_room
     {
       name: "conv-abc123",
       service: "conference.nexgate.com",
       options: { persistent: true, members_only: true }
     }
  3. Creator auto-joined as OWNER
  4. Generate invite link token
  5. Return { conversationId, inviteLink }
```

Group Join — Two Mechanisms

```
Mechanism 1: Consent DM Invitation
  Admin selects contacts/followers
  Spring Boot sends invitation stanza via Ejabberd:
    <message to="juma@nexgate.com">
      <nexgate-group-invite xmlns="urn:nexgate:group:1">
        <group_id>conv-abc123</group_id>
        <group_name>Business Friends</group_name>
        <invited_by>Kibuti Mwangi</invited_by>
        <expires_at>2026-07-17T10:00:00Z</expires_at>
      </nexgate-group-invite>
    </message>

  Recipient taps Accept:
    POST /chat/groups/conv-abc123/join
    Spring Boot: ejabberd REST → add_member

  48h no response: auto-declined silently

Mechanism 2: Invite Link
  PRIVATE group: link → request → admin approves
  PUBLIC group:  link → instant join
  Admin controls: expiry, max joins, revoke
```

Fan-out

```
Group message fan-out:
  Member sends to room JID
  Ejabberd MUC broadcasts to ALL members
  Erlang handles fan-out natively
  No Redis pub/sub needed
  Spring Boot persists via RabbitMQ event

At 500 members:
  Still Ejabberd MUC fan-out □
  Erlang is built for this
```

---

12. Commerce DM Flow

Product Card (Seller Attaches in 1:1)

```
Seller taps [From My Shop] in 1:1 DM attach menu:

Spring Boot builds stanza using ng standard:
  card_type: PRODUCT_CARD
  sender_type: SHOP
  inbox: COMMERCE
  shareable: true
  See Section 15 Example 2 for full stanza □

Rules:
  1:1 DM only (never group) □
  Price = current price at snapshot time
  Buyer's app renders product card UI
  [View Product] [Chat with Shop] buttons
```

Custom Price Offer Lifecycle

```
States: PENDING → ACCEPTED|DECLINED|EXPIRED|COMPLETED

Seller creates offer (1:1 DM only):
  POST /chat/commerce/offer/create
  { conversationId, productId, offerPrice, validMinutes: 30 }

Spring Boot:
  1. Create offer_sessions record
  2. Build ng stanza:
       card_type: CUSTOM_PRICE_OFFER
       sender_type: SHOP
       inbox: COMMERCE
       shareable: false (ALWAYS)
       expires_at: now + validMinutes
       item_type: PRODUCT or TICKET
       See Section 15 Example 3 + 4 for full stanza □
  3. Send via Ejabberd REST
  4. Start expiry timer (Redis)
```

```
Buyer accepts:
  POST /chat/commerce/offer/accept
  Spring Boot: update status → ACCEPTED
  Trigger checkout flow

Offer expires (30 min):
  Spring Boot scheduler fires
  Update status → EXPIRED
  Send expiry ng stanza:
    card_type: OFFER_EXPIRED
    priority: SILENT
  Buyer's UI: "Offer Expired □"
```

Bei Ya Pamoja (Group Buy)

```
NOT a group type — it is a CARD □
Can be shared in: DIRECT, GROUP, COMMERCE_DM
Cannot be shared in: DIRECT_SECRET □

Spring Boot builds ng stanza:
  card_type: GROUP_BUY
  item_type: PRODUCT or TICKET
  shareable: true
  members_preview: max 3 avatars + more_count
  See Section 15 Example 5 + 6 for full stanza □

Spring Boot updates progress in real-time:
  When member joins:
    Sends GROUP_BUY_PROGRESS signal (priority: SILENT)
    References original card by message_id
    App updates card in place — no notification □
    See Section 15 Example 7 for progress stanza □
  When target reached → trigger group checkout
  When expired → send expired card stanza

Card states:
  Initial → In Progress → Joined → Target Reached → Expired
```

Order Updates (System Messages)

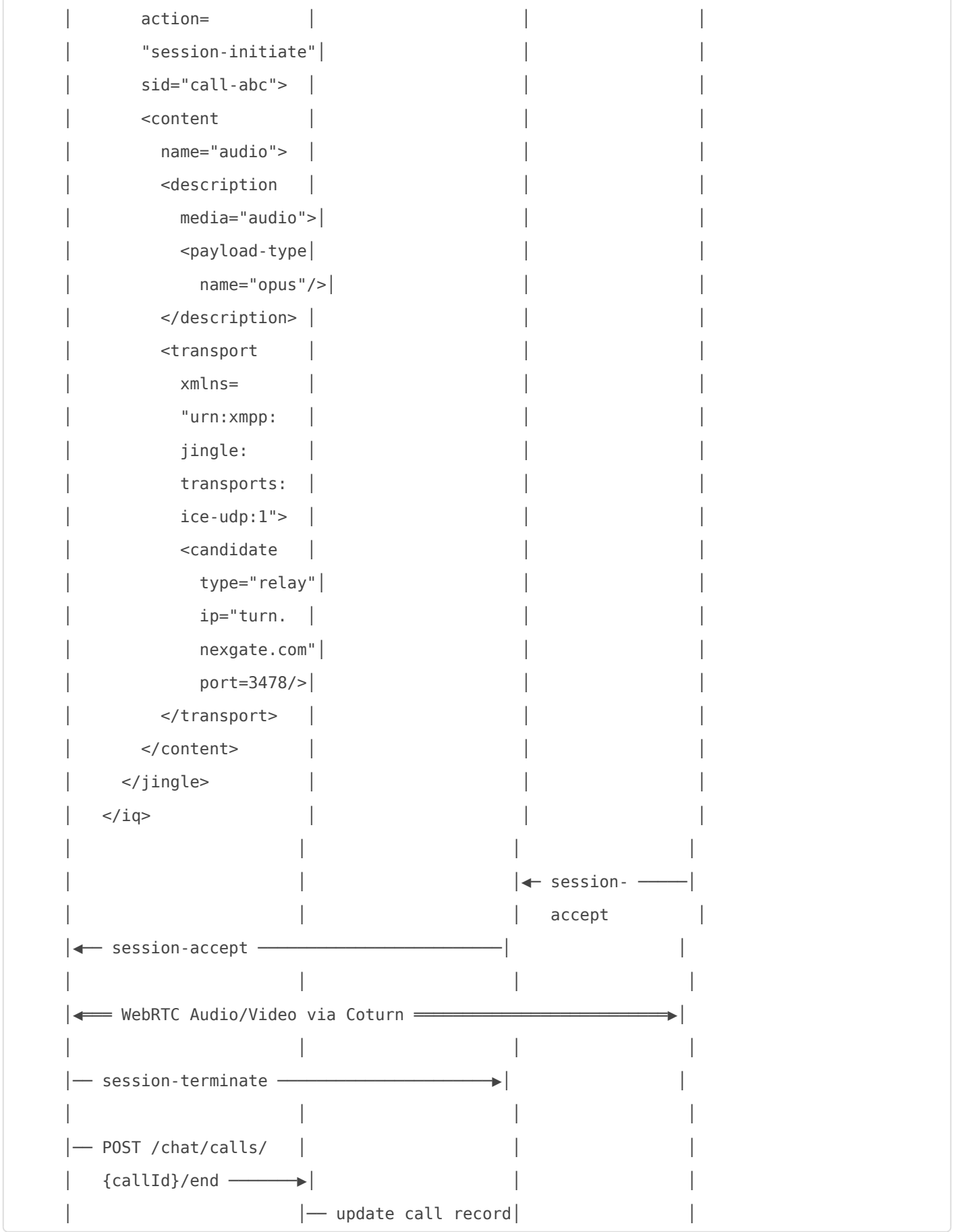
```
Auto-sent by Spring Boot (no manual action):
  Uses ng stanza standard:
    sender_type: SYSTEM
    message_type: CARD
    shareable: false
    deletable: false
    reactable: false
    inbox: COMMERCE
    card_type: ORDER_CONFIRMATION or ORDER_STATUS_UPDATE
  See Section 15 Example 12 for full stanza □

Rules:
  Goes directly to commerce thread □
  Cannot be deleted □ (deletable: false in meta)
  Cannot be reacted to □ (reactable: false in meta)
  Cannot be quoted □ (quotable: false in meta)
  Immutable for legal/audit □
```

---

13. Call Flow — Voice & Video

1:1 Call (WebRTC + Jingle)

```
Alice calls Bob:


Mobile (Alice)          Spring Boot            Ejabberd        Bob's Device
    |                       |                     |                |
    |── GET /chat/calls/    |                     |                |
    |   turn-credentials ──►|                     |                |
    |                       |── generate HMAC     |                |
    |                       |   TURN credentials  |                |
    |◄── { urls, username,  |                     |                |
    |      credential }     |                     |                |
    |                       |                     |                |
    |── Jingle              |                     |                |
    |   session-initiate ──────────────────────►|                |
    |   <iq type="set">     |                     |── route ──────►|
    |     <jingle           |                     |                |
```



Audio ? Video Switching (No Hang Up)

```
Mid-call: Alice enables camera:
  Alice → Jingle content-add:
    <jingle action="content-add" sid="call-abc">
      <content name="video">
        <description media="video">
          <payload-type name="H264"/>
        </description>
      </content>
    </jingle>

Bob accepts:
  Jingle content-accept
  Video starts — audio uninterrupted □

Switch back (video → audio):
  Jingle content-remove
  Audio continues □
```

TURN Credentials Generation

```
Spring Boot generates time-limited TURN credentials:
  username = timestamp:userId
  credential = HMAC-SHA256(sharedSecret, username)
  valid for: 1 hour

TURN server (Coturn) validates:
  Checks HMAC signature
  Checks timestamp not expired
  Allows relay □

Coturn config:
  use-auth-secret
  static-auth-secret = <shared-secret-from-vault>
  realm = nexgate.com
```

Group Call (LiveKit SFU)

```
3+ people call → use LiveKit (not P2P):

Spring Boot:
  1. Create LiveKit room: group-call-{callId}
  2. Generate token per participant
  3. Send join info via ng stanza:
       message_type: CALL
       signal_type: GROUP_CALL_JOIN_INFO
       priority: HIGH
       expires_at: now + 5 minutes
       shareable: false
       See Section 15 Example 11 for full stanza □

Each participant:
  Receives stanza → phone rings
  Joins → WebRTC to LiveKit
  LiveKit SFU forwards streams

EA network strategy:
  Simulcast (low/medium/high quality)
  Each receiver gets quality network allows
  Bad network → low quality (not dropped)
```

---

14. Secret Chat — End-to-End Encryption

What Secret Chat Is

```
Normal chat in NexGate:
  Alice sends message → Ejabberd routes → Bob
  Spring Boot stores message in DB (readable)
  NexGate can read messages if legally required
  Good for: commerce DMs, groups, order updates

Secret Chat:
  Alice encrypts ON DEVICE → Ejabberd routes → Bob decrypts ON DEVICE
```

```
  Spring Boot stores ENCRYPTED BLOB (unreadable)
  NexGate CANNOT read messages
  Keys NEVER leave the device
  Good for: personal private conversations

Secret Chat = optional feature
User explicitly chooses to start one
Cannot be forced or auto-converted
```

Technology — OMEMO (XEP-0384)

```
OMEMO = Encryption that scales to multiple devices
Based on Signal Protocol (same as WhatsApp)
Built for XMPP natively

Why OMEMO over other options:
  Signal Protocol = most secure available □
  Multi-device support built-in □
  Standard XEP = Ejabberd supports natively □
  Forward secrecy □
    (old messages safe even if key compromised)
  Deniability □
    (cannot prove who sent a message)
  Open standard □ (auditable)
```

How OMEMO Works (Simple)

```
Key Setup (happens once per device):
  Alice's phone generates:
    Identity Key pair (permanent)
    Signed PreKey pair (rotates every week)
    One-Time PreKeys (100 per batch)

  Alice publishes PUBLIC keys to Ejabberd:
    Via XMPP PubSub (XEP-0060)
    Ejabberd stores: alice's public keys
    Bob can fetch them anytime
```

```
  Private keys:
    NEVER leave Alice's device □
    Spring Boot NEVER sees them □
    Ejabberd NEVER sees them □

Message Encryption:
  Alice wants to send to Bob:
    1. Fetch Bob's public keys from Ejabberd
    2. Generate session key (Diffie-Hellman)
    3. Encrypt message with session key
    4. Encrypt session key with Bob's identity key
    5. Send encrypted blob via XMPP stanza

  Bob receives:
    1. Decrypt session key using his private key
    2. Decrypt message using session key
    3. Display plaintext to Bob □

  Ejabberd sees: encrypted blob only □
  Spring Boot stores: encrypted blob only □
  Nobody between Alice and Bob can read □
```

Multi-Device in Secret Chat

```
Alice has phone + tablet:
  OMEMO encrypts for EACH device separately

  Message encrypted for:
    Bob's phone key □
    Bob's tablet key □
    Alice's tablet key □ (carbon copy)

  Each device decrypts its own copy
  All devices see the message □

  If Alice adds new device:
    Must re-establish sessions
    Old messages NOT automatically available
    (forward secrecy — by design) □
```

Secret Chat Flow

```
Alice starts Secret Chat with Bob:

  Alice:
    Tap Bob's profile
    → "Start Secret Chat" □□
    → App checks: do I have Bob's OMEMO keys?
    → If no: fetch from Ejabberd PubSub
    → Establish encrypted session
    → UI shows: "□□Secret Chat"
    → "Messages are end-to-end encrypted"

  Alice sends message:
    App encrypts locally □
    Sends encrypted stanza:

    <message to="bob@nexgate.com" type="chat">
      <body>I can't read this</body>
      ← fallback for non-OMEMO clients

      <encrypted xmlns="eu.siacs.conversations.axolotl">
        <header sid="473229364">
          <key rid="987654321">
            BASE64_ENCRYPTED_SESSION_KEY_FOR_BOB
          </key>
          <key rid="111111111">
            BASE64_ENCRYPTED_SESSION_KEY_FOR_ALICE_TABLET
          </key>
          <iv>BASE64_IV</iv>
        </header>
        <payload>BASE64_ENCRYPTED_MESSAGE_BODY</payload>
      </encrypted>
    </message>

  Ejabberd:
    Routes stanza (cannot read payload) □

  Spring Boot (RabbitMQ consumer):
```

```
    Receives event
    Detects: has OMEMO encryption □
    Stores encrypted payload in DB:
      messages.body = null
      messages.encrypted_payload = BASE64_BLOB
      messages.is_e2ee = true
    Does NOT attempt to decrypt □

  Bob's device:
    Receives stanza
    Decrypts using private key
    Displays plaintext □
```

What Spring Boot Stores vs Doesn't

```
Normal message:
  messages.body = "Hello Bob!" ← readable
  messages.is_e2ee = false

Secret Chat message:
  messages.body = null ← empty □
  messages.encrypted_payload = "BASE64..." ← blob
  messages.is_e2ee = true
  messages.sender_id = usr-alice (known)
  messages.conversation_id = conv-abc (known)
  messages.created_at = timestamp (known)

Spring Boot knows:
  WHO sent □ (metadata)
  WHEN sent □ (metadata)
  TO WHOM □ (metadata)
  WHAT conversation □ (metadata)

Spring Boot does NOT know:
  WHAT was said □ (content encrypted)
  This is by design □
  Legal compliance: "we cannot read it" □
```

Secret Chat Rules

```
Can do in Secret Chat:
  □ Send text messages (encrypted)
  □ Send images (encrypted)
  □ Send voice notes (encrypted)
  □ Send files (encrypted)
  □ Set self-destruct timer
  □ Verify contact's identity (key fingerprint)

Cannot do in Secret Chat:
  □ Forward messages (breaks E2EE chain)
  □ Screenshot (Android FLAG_SECURE)
  □ Quote/reply across devices
  □ Search message content (encrypted in DB)
  □ Commerce stanzas (offer sessions need DB)
  □ Group Secret Chat (Phase 3 — complex)
  □ Web client (keys on device only)
```

Self-Destruct Timer

```
Optional feature in Secret Chat:
  Alice sets: "Delete after 5 minutes"
  Both devices delete locally after timer
  Server record remains (encrypted blob)
  BUT: server also deletes after timer □

Implementation:
  Uses ng stanza standard:
    message_type: TEXT
    encrypted: true
    inbox: SECRET
    expires_at: now + 300 seconds (self-destruct)
    OMEMO payload inside <payload> block
  See Section 15 Example 13 for full stanza □

  Self-destruct carried via expires_at in <meta>:
    <expires_at>2026-07-17T10:05:00Z</expires_at>
    Spring Boot: sets Redis TTL on message
    After TTL: deletes from DB □
    Device: shows countdown timer in UI □
```

```
Spring Boot:
  Receives event → sets Redis TTL on message:
    EXPIRE msg:msg-abc 300
  After 300 seconds:
    Spring Boot deletes from DB □
  Recipient device:
    Timer shown in UI (countdown)
    Local delete after timer □
```

Identity Verification

```
Users can verify each other's identity:
  Compare OMEMO key fingerprints
  Out-of-band (meet in person, voice call)
  "My fingerprint: A1B2 C3D4 E5F6..."
  "Your fingerprint matches □"

In UI:
  Contact profile → "Verify Security"
  Shows: "Your Safety Number with Alice"
  64-character fingerprint
  QR code option □
  If verified: show verified badge □□□

Why this matters:
  Protects against man-in-the-middle attacks
  "Is Ejabberd showing me Alice's real keys?"
  After verification: guaranteed □
```

Key Management in Spring Boot

```
Spring Boot role in OMEMO:
  Store public keys (not private) □
  Serve public keys via Ejabberd PubSub □
  Never store private keys □

Public key storage:
```

```
  Ejabberd PubSub handles automatically
  Node: eu.siacs.conversations.axolotl.bundles:{deviceId}
  Spring Boot does NOT manage this
  Ejabberd handles key distribution □

Key rotation:
  Device rotates Signed PreKey weekly
  Device publishes new PreKey to Ejabberd PubSub
  Spring Boot uninvolved □
  One-Time PreKeys replenished automatically
  When running low: device publishes more □

Spring Boot only knows:
  Which users have OMEMO enabled
  Which conversations are Secret Chats
  That encrypted blobs exist (not content)
```

UI Treatment

```
Secret Chat conversation:
  □□Lock icon in conversation header
  Dark/different color scheme (optional)
  "Messages are end-to-end encrypted" banner
  Timer icon if self-destruct enabled □□

Starting Secret Chat:
  Long press on contact OR
  Three dot menu → "Start Secret Chat"
  Separate conversation from normal chat
  Cannot accidentally send to wrong chat

Incoming Secret Chat:
  "Alice wants to start a Secret Chat"
  [Accept] [Decline]

Key fingerprint screen:
  Settings → Conversations → [name]
  → "View Security Code"
  Shows QR + text fingerprint
```

What's NOT E2EE (Important)

```
Secret Chat is 1:1 ONLY at launch:
  □ Group Secret Chat (Phase 3)
  □ Commerce DMs (need server-side logic)
  □ Shop conversations (staff audit needed)
  □ System messages (order notifications)
  □ Call content (WebRTC handles separately)

Call security (separate from OMEMO):
  WebRTC calls: DTLS-SRTP encrypted □
    (built into WebRTC standard)
    Keys negotiated per call
    Nobody can intercept call media □
  Call metadata: Spring Boot knows
    Who called whom □ (metadata)
    How long □ (metadata)
    But not WHAT was said □
```

What Secret Chat Supports at Launch

```
Phase 1 (Launch — all that is needed):
  □ 1:1 Secret Chat (OMEMO)
  □ Self-destruct timer
  □ Identity verification (fingerprint/QR)
  □ Image/file encryption
  □ Voice note encryption

Phase 1 is complete and sufficient.
No Phase 2 or Phase 3 planned.
```

---

15. NexGate Stanza Standard

The Element

```
Every NexGate message carries ONE unified element:

  <ng xmlns="urn:nexgate:1">

  ng            = NexGate (short name)
  urn:nexgate:1 = NexGate root namespace version 1

  Contains TWO children — ALWAYS:
    <meta>    → message metadata (always present, all fields)
    <payload> → message content (empty for TEXT)

  Ejabberd: routes the <message> wrapper — ignores <ng> 
  Mobile app: reads <ng> to render correctly 
  Spring Boot: reads <ng> to persist correctly 
  Basic XMPP clients: see <body> fallback only 
```

The Meta Block — All Fields

```
<meta>
  <sender_type>USER|SHOP|SYSTEM</sender_type>
  <sender_id>usr-kibuti</sender_id>

  <message_type>TEXT|CARD|MEDIA|CALL</message_type>

  <shareable>true|false</shareable>
  <deletable>true|false</deletable>
  <reactable>true|false</reactable>
  <quotable>true|false</quotable>

  <priority>NORMAL|HIGH|SILENT</priority>
  <inbox>PERSONAL|COMMERCE|SECRET</inbox>
  <encrypted>true|false</encrypted>
  <expires_at/>
</meta>
```

```
sender_type:
  USER   → regular user
  SHOP   → shop (staff hidden from customer)
  SYSTEM → Spring Boot auto (no human)
```

```
message_type:
  TEXT → plain text message
  CARD → rich interactive card
  MEDIA → image/video/voice/file/gif/sticker
  CALL → call lifecycle signal

shareable:
  true  → Forward option shown in UI □
  false → Forward hidden + API returns 403 □

deletable:
  true  → sender can delete for everyone
  false → immutable (system/payment/order)

reactable:
  true  → emoji reactions allowed
  false → no reactions (system/call signals)

quotable:
  true  → can be replied/quoted
  false → cannot quote (system/call signals)

priority:
  NORMAL → standard notification
  HIGH   → ring even in DND (calls)
  SILENT → update UI only, no notification
           (GROUP_BUY_PROGRESS, card updates)

inbox:
  PERSONAL → Personal tab
  COMMERCE → Commerce tab
  SECRET   → Secret tab □□

encrypted:
  true  → OMEMO Secret Chat
  false → normal message

expires_at:
```

```
  empty → lives forever (default)
  timestamp → auto-delete after this time
```

Meta Defaults Per Message Type

```
Field         TEXT    CARD    MEDIA   CALL    SYSTEM
──────────────────────────────────────────────────────
sender_type   USER    varies  USER    USER    SYSTEM
shareable     true    varies  true    false   false
deletable     true    varies  true    false   false
reactable     true    varies  true    false   false
quotable      true    varies  true    false   false
priority      NORMAL  NORMAL  NORMAL  HIGH    NORMAL
inbox         PERSONAL varies PERSONAL PERSONAL COMMERCE
encrypted     false   false   false   false   false
expires_at    empty   empty   empty   5min    empty
```

Complete Stanza Examples

1. Plain Text Message

```
<message to="bob@nexgate.com" type="chat" id="msg-001">
  <body>Hello Bob!</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>USER</sender_type>
      <sender_id>usr-alice</sender_id>
      <message_type>TEXT</message_type>
      <shareable>true</shareable>
      <deletable>true</deletable>
      <reactable>true</reactable>
      <quotable>true</quotable>
      <priority>NORMAL</priority>
      <inbox>PERSONAL</inbox>
      <encrypted>false</encrypted>
      <expires_at/>
    </meta>
    <payload/>
  </ng>
```

```
  <origin-id xmlns="urn:xmpp:sid:0" id="msg-001"/>
  <request xmlns="urn:xmpp:receipts"/>
  <markable xmlns="urn:xmpp:chat-markers:0"/>
</message>
```

2. Product Card

```
<message to="buyer@nexgate.com" type="chat" id="msg-002">
  <body>Check out this product</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>SHOP</sender_type>
      <sender_id>shop-456</sender_id>
      <message_type>CARD</message_type>
      <shareable>true</shareable>
      <deletable>false</deletable>
      <reactable>true</reactable>
      <quotable>false</quotable>
      <priority>NORMAL</priority>
      <inbox>COMMERCE</inbox>
      <encrypted>false</encrypted>
      <expires_at/>
    </meta>
    <payload>
      <card_type>PRODUCT_CARD</card_type>
      <product>
        <id>prod-123</id>
        <name>Samsung A15</name>
        <price>450000</price>
        <currency>TZS</currency>
        <image_url>https://cdn.nexgate.com/img.jpg</image_url>
        <shop_name>TechStore</shop_name>
        <shop_id>shop-456</shop_id>
        <stock>12</stock>
      </product>
    </payload>
  </ng>
  <origin-id xmlns="urn:xmpp:sid:0" id="msg-002"/>
  <request xmlns="urn:xmpp:receipts"/>
</message>
```

3. Custom Price Offer (Product)

```
<message to="buyer@nexgate.com" type="chat" id="msg-003">
  <body>Special price offer for you</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>SHOP</sender_type>
      <sender_id>shop-456</sender_id>
      <message_type>CARD</message_type>
      <shareable>false</shareable>
      <deletable>false</deletable>
      <reactable>false</reactable>
      <quotable>false</quotable>
      <priority>NORMAL</priority>
      <inbox>COMMERCE</inbox>
      <encrypted>false</encrypted>
      <expires_at>2026-07-17T11:30:00Z</expires_at>
    </meta>
    <payload>
      <card_type>CUSTOM_PRICE_OFFER</card_type>
      <offer_id>offer-abc-123</offer_id>
      <status>PENDING</status>
      <valid_minutes>30</valid_minutes>
      <item>
        <item_type>PRODUCT</item_type>
        <item_id>prod-123</item_id>
        <item_name>Samsung A15</item_name>
        <image_url>https://cdn.nexgate.com/img.jpg</image_url>
        <shop_name>TechStore</shop_name>
      </item>
      <pricing>
        <public_price>450000</public_price>
        <offer_price>400000</offer_price>
        <currency>TZS</currency>
        <discount_amount>50000</discount_amount>
        <discount_pct>11</discount_pct>
      </pricing>
    </payload>
  </ng>
  <origin-id xmlns="urn:xmpp:sid:0" id="msg-003"/>
```

```
</message>
```

4. Custom Price Offer (Ticket)

```
<message to="buyer@nexgate.com" type="chat" id="msg-004">
  <body>Special ticket price for you</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>SYSTEM</sender_type>
      <sender_id>org-123</sender_id>
      <message_type>CARD</message_type>
      <shareable>false</shareable>
      <deletable>false</deletable>
      <reactable>false</reactable>
      <quotable>false</quotable>
      <priority>NORMAL</priority>
      <inbox>COMMERCE</inbox>
      <encrypted>false</encrypted>
      <expires_at>2026-07-17T12:00:00Z</expires_at>
    </meta>
    <payload>
      <card_type>CUSTOM_PRICE_OFFER</card_type>
      <offer_id>offer-evt-456</offer_id>
      <status>PENDING</status>
      <valid_minutes>60</valid_minutes>
      <item>
        <item_type>TICKET</item_type>
        <item_id>evt-789</item_id>
        <item_name>Dar Tech Summit 2026</item_name>
        <image_url>https://cdn.nexgate.com/evt.jpg</image_url>
        <event_date>2026-08-15T09:00:00Z</event_date>
        <venue>Julius Nyerere ICC, Dar es Salaam</venue>
        <ticket_type>VIP</ticket_type>
        <quantity>2</quantity>
      </item>
      <pricing>
        <public_price>50000</public_price>
        <offer_price>35000</offer_price>
        <currency>TZS</currency>
        <discount_amount>15000</discount_amount>
```

```
        <discount_pct>30</discount_pct>
      </pricing>
    </payload>
  </ng>
  <origin-id xmlns="urn:xmpp:sid:0" id="msg-004"/>
</message>
```

5. Group Buy Card (Product) with Members Preview

```
<message to="conv-abc@conference.nexgate.com"
         type="groupchat" id="msg-005">
  <body>Group purchase: Samsung A15</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>USER</sender_type>
      <sender_id>usr-kibuti</sender_id>
      <message_type>CARD</message_type>
      <shareable>true</shareable>
      <deletable>false</deletable>
      <reactable>true</reactable>
      <quotable>false</quotable>
      <priority>NORMAL</priority>
      <inbox>PERSONAL</inbox>
      <encrypted>false</encrypted>
      <expires_at>2026-07-17T18:00:00Z</expires_at>
    </meta>
    <payload>
      <card_type>GROUP_BUY</card_type>
      <group_buy_id>gb-xyz-789</group_buy_id>
      <item>
        <item_type>PRODUCT</item_type>
        <item_id>prod-123</item_id>
        <item_name>Samsung A15</item_name>
        <image_url>https://cdn.nexgate.com/img.jpg</image_url>
        <shop_name>TechStore</shop_name>
      </item>
      <pricing>
        <public_price>450000</public_price>
        <group_price>350000</group_price>
        <currency>TZS</currency>
```

```
      </pricing>
      <progress>
        <current>8</current>
        <target>10</target>
        <pct>80</pct>
      </progress>
      <members_preview>
        <member>
          <id>usr-kibuti</id>
          <name>Kibuti</name>
          <avatar>https://cdn.nexgate.com/av1.jpg</avatar>
        </member>
        <member>
          <id>usr-juma</id>
          <name>Juma</name>
          <avatar>https://cdn.nexgate.com/av2.jpg</avatar>
        </member>
        <member>
          <id>usr-alice</id>
          <name>Alice</name>
          <avatar>https://cdn.nexgate.com/av3.jpg</avatar>
        </member>
        <more_count>5</more_count>
      </members_preview>
    </payload>
  </ng>
  <origin-id xmlns="urn:xmpp:sid:0" id="msg-005"/>
  <request xmlns="urn:xmpp:receipts"/>
</message>
```

6. Group Buy Card (Ticket)

```
<message to="bob@nexgate.com" type="chat" id="msg-006">
  <body>Group ticket purchase: Dar Tech Summit</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>USER</sender_type>
      <sender_id>usr-alice</sender_id>
      <message_type>CARD</message_type>
      <shareable>true</shareable>
```

```
    <deletable>false</deletable>
    <reactable>true</reactable>
    <quotable>false</quotable>
    <priority>NORMAL</priority>
    <inbox>PERSONAL</inbox>
    <encrypted>false</encrypted>
    <expires_at>2026-07-17T20:00:00Z</expires_at>
  </meta>
  <payload>
    <card_type>GROUP_BUY</card_type>
    <group_buy_id>gb-evt-001</group_buy_id>
    <item>
      <item_type>TICKET</item_type>
      <item_id>evt-789</item_id>
      <item_name>Dar Tech Summit 2026</item_name>
      <image_url>https://cdn.nexgate.com/evt.jpg</image_url>
      <event_date>2026-08-15T09:00:00Z</event_date>
      <venue>Julius Nyerere ICC</venue>
      <ticket_type>GENERAL</ticket_type>
    </item>
    <pricing>
      <public_price>25000</public_price>
      <group_price>18000</group_price>
      <currency>TZS</currency>
    </pricing>
    <progress>
      <current>15</current>
      <target>20</target>
      <pct>75</pct>
    </progress>
    <members_preview>
      <member>
        <id>usr-alice</id>
        <name>Alice</name>
        <avatar>https://cdn.nexgate.com/av1.jpg</avatar>
      </member>
      <member>
        <id>usr-bob</id>
        <name>Bob</name>
        <avatar>https://cdn.nexgate.com/av2.jpg</avatar>
```

```
        </member>
        <member>
          <id>usr-juma</id>
          <name>Juma</name>
          <avatar>https://cdn.nexgate.com/av3.jpg</avatar>
        </member>
        <more_count>12</more_count>
      </members_preview>
    </payload>
  </ng>
  <origin-id xmlns="urn:xmpp:sid:0" id="msg-006"/>
</message>
```

7. Group Buy Progress Update (Silent)

```
<message to="conv-abc@conference.nexgate.com"
         type="groupchat" id="msg-007">
  <body>Group buy updated</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>SYSTEM</sender_type>
      <sender_id>system</sender_id>
      <message_type>CARD</message_type>
      <shareable>false</shareable>
      <deletable>false</deletable>
      <reactable>false</reactable>
      <quotable>false</quotable>
      <priority>SILENT</priority>
      <inbox>PERSONAL</inbox>
      <encrypted>false</encrypted>
      <expires_at/>
    </meta>
    <payload>
      <card_type>GROUP_BUY_PROGRESS</card_type>
      <group_buy_id>gb-xyz-789</group_buy_id>
      <original_message_id>msg-005</original_message_id>
      <progress>
        <current>9</current>
        <target>10</target>
        <pct>90</pct>
```

```
      </progress>
      <members_preview>
        <member>
          <id>usr-kibuti</id>
          <name>Kibuti</name>
          <avatar>https://cdn.nexgate.com/av1.jpg</avatar>
        </member>
        <member>
          <id>usr-juma</id>
          <name>Juma</name>
          <avatar>https://cdn.nexgate.com/av2.jpg</avatar>
        </member>
        <member>
          <id>usr-new</id>
          <name>Naserian</name>
          <avatar>https://cdn.nexgate.com/av4.jpg</avatar>
        </member>
        <more_count>6</more_count>
      </members_preview>
    </payload>
  </ng>
  <no-store xmlns="urn:xmpp:hints"/>
</message>
```

8. Image (Media)

```
<message to="bob@nexgate.com" type="chat" id="msg-008">
  <body>□□Photo</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>USER</sender_type>
      <sender_id>usr-alice</sender_id>
      <message_type>MEDIA</message_type>
      <shareable>true</shareable>
      <deletable>true</deletable>
      <reactable>true</reactable>
      <quotable>false</quotable>
      <priority>NORMAL</priority>
      <inbox>PERSONAL</inbox>
      <encrypted>false</encrypted>
```

```
      <expires_at/>
    </meta>
    <payload>
      <media_type>IMAGE</media_type>
      <file_id>file-abc-123</file_id>
      <url>https://cdn.nexgate.com/img.jpg</url>
      <thumbnail_url>https://cdn.nexgate.com/thumb.jpg</thumbnail_url>
      <mime_type>image/jpeg</mime_type>
      <size>245000</size>
      <width>1080</width>
      <height>720</height>
    </payload>
  </ng>
  <origin-id xmlns="urn:xmpp:sid:0" id="msg-008"/>
  <request xmlns="urn:xmpp:receipts"/>
  <markable xmlns="urn:xmpp:chat-markers:0"/>
</message>
```

9. Voice Note (Media)

```
<message to="bob@nexgate.com" type="chat" id="msg-009">
  <body>□□Voice note</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>USER</sender_type>
      <sender_id>usr-alice</sender_id>
      <message_type>MEDIA</message_type>
      <shareable>true</shareable>
      <deletable>true</deletable>
      <reactable>true</reactable>
      <quotable>false</quotable>
      <priority>NORMAL</priority>
      <inbox>PERSONAL</inbox>
      <encrypted>false</encrypted>
      <expires_at/>
    </meta>
    <payload>
      <media_type>VOICE_NOTE</media_type>
      <file_id>file-voice-456</file_id>
      <url>https://cdn.nexgate.com/voice/abc.ogg</url>
```

```
      <mime_type>audio/ogg</mime_type>
      <size>48000</size>
      <duration_seconds>15</duration_seconds>
      <waveform>0.1,0.4,0.8,0.6,0.3,0.9,0.2,0.5</waveform>
    </payload>
  </ng>
  <origin-id xmlns="urn:xmpp:sid:0" id="msg-009"/>
  <request xmlns="urn:xmpp:receipts"/>
</message>
```

10. Incoming Call Signal

```
<message to="bob@nexgate.com" type="chat" id="msg-010">
  <body>Incoming call from Alice</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>USER</sender_type>
      <sender_id>usr-alice</sender_id>
      <message_type>CALL</message_type>
      <shareable>false</shareable>
      <deletable>false</deletable>
      <reactable>false</reactable>
      <quotable>false</quotable>
      <priority>HIGH</priority>
      <inbox>PERSONAL</inbox>
      <encrypted>false</encrypted>
      <expires_at>2026-07-17T10:05:00Z</expires_at>
    </meta>
    <payload>
      <signal_type>CALL_INITIATED</signal_type>
      <call_id>call-abc-789</call_id>
      <call_type>VIDEO</call_type>
      <caller_name>Alice</caller_name>
      <caller_avatar>https://cdn.nexgate.com/av1.jpg</caller_avatar>
    </payload>
  </ng>
  <no-store xmlns="urn:xmpp:hints"/>
</message>
```

11. Group Call Join Info

```
<message to="bob@nexgate.com" type="chat" id="msg-011">
  <body>Join the group call</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>SYSTEM</sender_type>
      <sender_id>system</sender_id>
      <message_type>CALL</message_type>
      <shareable>false</shareable>
      <deletable>false</deletable>
      <reactable>false</reactable>
      <quotable>false</quotable>
      <priority>HIGH</priority>
      <inbox>PERSONAL</inbox>
      <encrypted>false</encrypted>
      <expires_at>2026-07-17T10:05:00Z</expires_at>
    </meta>
    <payload>
      <signal_type>GROUP_CALL_JOIN_INFO</signal_type>
      <call_id>call-group-xyz</call_id>
      <call_type>VOICE</call_type>
      <livekit_url>wss://livekit.nexgate.com</livekit_url>
      <livekit_token>eyJhbGciOiJIUzI1NiJ9...</livekit_token>
      <room_id>group-call-xyz</room_id>
      <initiated_by>Kibuti Mwangi</initiated_by>
    </payload>
  </ng>
  <no-store xmlns="urn:xmpp:hints"/>
</message>
```

12. Order Confirmation (System Card)

```
<message to="buyer@nexgate.com" type="chat" id="msg-012">
  <body>Your order has been confirmed</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>SYSTEM</sender_type>
      <sender_id>system</sender_id>
      <message_type>CARD</message_type>
      <shareable>false</shareable>
      <deletable>false</deletable>
```

```
      <reactable>false</reactable>
      <quotable>false</quotable>
      <priority>NORMAL</priority>
      <inbox>COMMERCE</inbox>
      <encrypted>false</encrypted>
      <expires_at/>
    </meta>
    <payload>
      <card_type>ORDER_CONFIRMATION</card_type>
      <order_id>ORD-789</order_id>
      <product_name>Samsung A15</product_name>
      <quantity>1</quantity>
      <amount_paid>400000</amount_paid>
      <currency>TZS</currency>
      <payment_method>M-PESA</payment_method>
      <status>CONFIRMED</status>
    </payload>
  </ng>
</message>
```

13. Secret Chat Message

```
<message to="bob@nexgate.com" type="chat" id="msg-013">
  <body>You have an encrypted message</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>USER</sender_type>
      <sender_id>usr-alice</sender_id>
      <message_type>TEXT</message_type>
      <shareable>false</shareable>
      <deletable>true</deletable>
      <reactable>false</reactable>
      <quotable>false</quotable>
      <priority>NORMAL</priority>
      <inbox>SECRET</inbox>
      <encrypted>true</encrypted>
      <expires_at>2026-07-17T10:35:00Z</expires_at>
    </meta>
    <payload>
      <encrypted xmlns="eu.siacs.conversations.axolotl">
```

```
        <header sid="473229364">
          <key rid="987654321">BASE64_KEY</key>
          <iv>BASE64_IV</iv>
        </header>
        <blob>BASE64_ENCRYPTED_BODY</blob>
      </encrypted>
    </payload>
  </ng>
  <origin-id xmlns="urn:xmpp:sid:0" id="msg-013"/>
</message>
```

14. Group Invitation Card

```
<message to="juma@nexgate.com" type="chat" id="msg-014">
  <body>You have been invited to join a group</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>USER</sender_type>
      <sender_id>usr-kibuti</sender_id>
      <message_type>CARD</message_type>
      <shareable>false</shareable>
      <deletable>false</deletable>
      <reactable>false</reactable>
      <quotable>false</quotable>
      <priority>NORMAL</priority>
      <inbox>PERSONAL</inbox>
      <encrypted>false</encrypted>
      <expires_at>2026-07-19T10:00:00Z</expires_at>
    </meta>
    <payload>
      <card_type>GROUP_INVITATION</card_type>
      <group_id>conv-abc123</group_id>
      <group_name>Business Friends</group_name>
      <group_type>PRIVATE</group_type>
      <member_count>47</member_count>
      <description>Dar founders discussion</description>
      <invited_by>Kibuti Mwangi</invited_by>
    </payload>
  </ng>
</message>
```

15. Post Card (Social)

```
<message to="bob@nexgate.com" type="chat" id="msg-015">
  <body>Check out this post</body>
  <ng xmlns="urn:nexgate:1">
    <meta>
      <sender_type>USER</sender_type>
      <sender_id>usr-alice</sender_id>
      <message_type>CARD</message_type>
      <shareable>true</shareable>
      <deletable>true</deletable>
      <reactable>true</reactable>
      <quotable>true</quotable>
      <priority>NORMAL</priority>
      <inbox>PERSONAL</inbox>
      <encrypted>false</encrypted>
      <expires_at/>
    </meta>
    <payload>
      <card_type>POST_CARD</card_type>
      <post_id>post-789</post_id>
      <author_name>Kibuti Mwangi</author_name>
      <author_username>kibuti</author_username>
      <author_avatar>https://cdn.nexgate.com/av.jpg</author_avatar>
      <caption>New Samsung models just arrived!</caption>
      <media_url>https://cdn.nexgate.com/post.jpg</media_url>
      <media_type>IMAGE</media_type>
      <like_count>245</like_count>
      <comment_count>12</comment_count>
    </payload>
  </ng>
  <origin-id xmlns="urn:xmpp:sid:0" id="msg-015"/>
  <request xmlns="urn:xmpp:receipts"/>
</message>
```

Standard XEPs — When to Include

```
XEP element                   When to include
_____________________________________________________________
```

```
<origin-id>                    All real messages □
                               NOT on CALL signals
                               NOT on SILENT updates

<request> (receipts)           TEXT + CARD + MEDIA □
                               NOT on CALL signals
                               NOT on SILENT updates
                               NOT on SYSTEM cards

<markable> (read markers)      TEXT + MEDIA □
                               NOT on CARD (no read tick)
                               NOT on CALL signals

<no-store> (hints)             CALL signals □
                               SILENT updates □
                               Typing indicators □
                               Anything ephemeral
```

The Complete Anatomy

```
<message>                    ← XMPP wrapper (Ejabberd routes)
  <body>fallback</body>      ← ALWAYS present

  <ng xmlns="urn:nexgate:1"> ← ALWAYS present
    <meta>                   ← ALWAYS all fields
      ...
    </meta>
    <payload>                ← empty for TEXT, content for others
      ...
    </payload>
  </ng>

  <!-- Standard XEPs below (as needed per table above) -->
  <origin-id/>
  <request/>
  <markable/>
  <no-store/>
</message>
```

16. NexGate Custom Namespaces

Complete Registry

```
All custom stanzas ALWAYS go inside <message>
Ejabberd routes without knowing what is inside
Frontend identifies by xmlns + type field

SOCIAL CARDS (VP Social):
  xmlns: urn:nexgate:social:1
  Types: POST_CARD, REEL_CARD, LIVE_CARD, PROFILE_CARD

COMMERCE CARDS (VP Shop):
  xmlns: urn:nexgate:commerce:1
  Types: PRODUCT_CARD, SHOP_CARD, FLASH_SALE_CARD

  xmlns: urn:nexgate:offer:1
  type: CUSTOM_PRICE_OFFER (always for offer cards)
        OFFER_RESPONSE, OFFER_EXPIRED
  item_type: PRODUCT | TICKET (for CUSTOM_PRICE_OFFER)

  xmlns: urn:nexgate:groupbuy:1
  type: GROUP_BUY (always)
  item_type: PRODUCT | TICKET
  Status types: GROUP_BUY_PROGRESS, GROUP_BUY_COMPLETED

  xmlns: urn:nexgate:installment:1
  Types: INSTALLMENT_PLAN_CARD

EVENT CARDS (VP Events):
  xmlns: urn:nexgate:event:1
  Types: EVENT_CARD, TICKET_CARD, EVENT_REMINDER, EVENT_GROUP_BUY

SYSTEM CARDS (Automated):
  xmlns: urn:nexgate:system:1
  Types: ORDER_CONFIRMATION, ORDER_STATUS_UPDATE,
         PAYMENT_CONFIRMATION, REFUND_CARD, DISPUTE_CARD
```

```
GROUP CARDS (chat_system):
  xmlns: urn:nexgate:group:1
  Types: GROUP_INVITATION (only card — user taps Accept/Decline)

CALL SIGNALS (chat_system — NOT cards, system signals):
  xmlns: urn:nexgate:call:1
  Types: CALL_INITIATED, CALL_ACCEPTED, CALL_DECLINED,
         CALL_ENDED, CALL_MISSED, GROUP_CALL_JOIN_INFO,
         PARTICIPANT_ADDED, HOST_MUTED_YOU, HOST_REMOVED_YOU

API TEMPLATE CARDS (Third-party Future):
  xmlns: urn:nexgate:template:1
  Types: TRANSACTIONAL, MARKETING, AUTHENTICATION

MEDIA:
  xmlns: urn:nexgate:media:1
  Types: IMAGE, VIDEO, VOICE_NOTE, FILE, STICKER, GIF

SECRET CHAT METADATA:
  xmlns: urn:nexgate:secret:1
  Types: SELF_DESTRUCT_TIMER, KEY_VERIFICATION

EPHEMERAL (always add no-store hint):
  xmlns: urn:nexgate:states    (recording state)
  xmlns: urn:nexgate:forward   (forward metadata)
```

Standard XEPs Used

```
XEP-0085   chatstates              Typing indicators
XEP-0184   urn:xmpp:receipts       Delivery ticks
XEP-0333   urn:xmpp:chat-markers:0 Read ticks
XEP-0308   message-correct:0       Edit message
XEP-0424   message-retract:0       Delete message
XEP-0444   urn:xmpp:reactions:0    Emoji reactions
XEP-0297   urn:xmpp:forward:0      Forward message
XEP-0461   urn:xmpp:reply:0        Reply/quote
XEP-0359   urn:xmpp:sid:0          Stable stanza IDs
XEP-0198   urn:xmpp:sm:3           Stream management
XEP-0199   urn:xmpp:ping           Keepalive ping
```

```
XEP-0334   urn:xmpp:hints          no-store hint
XEP-0384   axolotl (OMEMO)         E2EE encryption
XEP-0166   urn:xmpp:jingle:1       Call signaling
XEP-0045   muc protocol            Group chat
XEP-0280   urn:xmpp:carbons:2      Multi-device sync
XEP-0313   urn:xmpp:mam:2          Message archive
```

Mobile App Decision Tree

```
Message arrives:

  Has urn:nexgate:social:1 ?
    POST_CARD  → post preview + [View Post]
    LIVE_CARD  → live badge + [Watch Live]

  Has urn:nexgate:commerce:1 ?
    PRODUCT_CARD → product card + [View] [Chat]

  Has urn:nexgate:offer:1 ?
    PENDING  → offer + timer + [Accept] [Decline]
    ACCEPTED → "Offer Accepted"
    EXPIRED  → "Offer Expired"

  Has urn:nexgate:groupbuy:1 ?
    GROUP_BUY_CARD → progress bar + [Join]

  Has urn:nexgate:event:1 ?
    EVENT_CARD  → event + [View] [Get Ticket]
    TICKET_CARD → ticket QR code

  Has urn:nexgate:system:1 ?
    ORDER_CONFIRMATION  → order + [Track]
    ORDER_STATUS_UPDATE → status badge
    PAYMENT_CONFIRMATION → payment details

  Has urn:nexgate:call:1 ?
    CALL_INITIATED      → incoming call screen
    GROUP_CALL_JOIN_INFO → [Join Call] [Decline]
    CALL_MISSED         → missed call indicator
```

```
    HOST_MUTED_YOU      → "You were muted by host"
    HOST_REMOVED_YOU    → "You were removed from call"

  Has urn:nexgate:group:1 ?
    GROUP_INVITATION → [Accept] [Decline]

  Has urn:nexgate:media:1 ?
    IMAGE      → show image
    VOICE_NOTE → waveform + play button
    STICKER    → full size, no bubble
    GIF        → auto-play loop
    VIDEO      → video player
    FILE       → download button

  Has urn:nexgate:template:1 ?
    TRANSACTIONAL → template card
    MARKETING     → card + unsubscribe

  No xmlns match ?
    Render as plain text (use body)

ALWAYS include body as fallback
  Basic XMPP clients see fallback text
  NexGate app renders rich card
```

---

17. RabbitMQ Event Pipeline

Exchange & Queue Setup

```
Exchange: nexgate.chat (topic exchange)

Queues:
  nexgate.chat.messages     ← all message events
  nexgate.chat.presence     ← online/offline events
  nexgate.chat.calls        ← call start/end events
  nexgate.chat.rooms        ← group create/destroy events
```

```
Routing keys:
  chat.message.inbound      → message arrived
  chat.message.edited       → message edited
  chat.message.retracted    → message deleted
  chat.message.reaction     → reaction added/removed
  chat.presence.online      → user came online
  chat.presence.offline     → user went offline
  chat.call.initiated       → call started
  chat.call.ended           → call ended
  chat.room.created         → group created
  chat.room.destroyed       → group deleted
```

Event Payload Structure

```
chat.message.inbound:
  {
    event_type: "message.inbound",
    from_jid: "alice@nexgate.com/android",
    to_jid: "bob@nexgate.com",
    stanza_id: "1784107799294256",
    origin_id: "msg-abc-123",
    conversation_type: "DIRECT|GROUP",
    room_jid: null | "conv-abc@conference.nexgate.com",
    body: "Hello Bob!",
    has_custom_namespace: true,
    namespace: "urn:nexgate:commerce:1",
    raw_stanza: "<message>...</message>",
    timestamp: 1721210400000
  }

chat.presence.offline:
  {
    event_type: "presence.offline",
    jid: "alice@nexgate.com/android",
    bare_jid: "alice@nexgate.com",
    timestamp: 1721210400000
  }

chat.call.ended:
```

```
  {
    event_type: "call.ended",
    call_id: "call-xyz",
    from_jid: "alice@nexgate.com",
    to_jid: "bob@nexgate.com",
    duration_seconds: 120,
    call_type: "VOICE|VIDEO",
    ended_reason: "normal|declined|missed|failed"
  }
```

Spring Boot Consumer Responsibilities

```
@RabbitListener(queues = "nexgate.chat.messages")
fun onMessage(event: ChatMessageEvent) {

  // 1. Parse event
  // 2. Find or create conversation
  // 3. Detect message type:
  if (event.namespace == "urn:nexgate:commerce:1") {
    // parse product card, link to product
  } else if (event.namespace == "urn:nexgate:offer:1") {
    // update offer session status
  } else {
    // plain message
  }

  // 4. Persist to chat.messages
  // 5. Update conversation last_message
  // 6. Check recipient online status
  //    If offline → send FCM via notification/
  // 7. For commerce stanzas → trigger business flows
}

@RabbitListener(queues = "nexgate.chat.presence")
fun onPresence(event: PresenceEvent) {
  // Update user last_seen in core.users
  // Update conversation online indicators
}
```

18. Shop Inbox & Staff Access

Shop XMPP Identity

```
Every shop has its own JID:
  $techstore → shop-456@nexgate.com

Staff members share this JID:
  Staff A connects as: shop-456@nexgate.com/staff-1
  Staff B connects as: shop-456@nexgate.com/staff-2

Customer always sees: TechStore (never staff name)
JID resource stripped before delivery □

Staff XMPP token:
  Issued by Spring Boot when staff logs in
  { jid: "shop-456@nexgate.com", exp: ... }
  Short-lived: 8 hours (staff shift)
```

Shop Inbox Routing

```
Customer messages shop:
  to="shop-456@nexgate.com"
  Ejabberd routes to available staff resource
  (round-robin or first available)

Staff replies:
  from="shop-456@nexgate.com/staff-1"
  Spring Boot strips /staff-1 resource
  Customer receives:
    from="shop-456@nexgate.com" □
  Customer never knows which staff replied □

Audit log:
  Spring Boot records which staff_id sent each message
  Internal only — never visible to customer
  Legal requirement for disputes
```

Shop Chat Initiation Rules

```
ALWAYS ALLOWED (transactional — auto):
  Order confirmed → system sends to buyer
  Order shipped → system sends to buyer
  Payment issue → system sends to buyer

ALLOWED (existing relationship):
  Previous buyer → shop can message from Orders section
  Active follower → shop can send from Customers section
  Goes to existing thread OR message request

NEVER ALLOWED:
  Cold message to random users □
  System blocks at API level
  "You can only message customers who
   interacted with your shop"
```

---

19. Offline & Push Notification Flow

```
Message arrives for Bob (offline):

Ejabberd:
  1. Detects Bob offline
  2. Stores in Ejabberd offline queue
  3. Fires RabbitMQ event: chat.message.inbound
     + is_recipient_offline: true

Spring Boot consumer:
  1. Persists message to DB
  2. Detects is_recipient_offline = true
  3. Calls notification/ package:
     - Has FCM token? → send FCM push □
     - No FCM token? → try Textfy SMS □
     - Priority: FCM first, SMS fallback

FCM payload:
```

```
  {
    to: "bob-fcm-token",
    notification: {
      title: "Alice",
      body: "Hello Bob!"
    },
    data: {
      conversation_id: "conv-abc",
      message_id: "msg-001",
      type: "chat"
    }
  }

Bob comes online:
  Ejabberd delivers offline queue
  Spring Boot receives delivery event
  Updates message status → DELIVERED
  Sends <received> receipt to Alice □
```

---

20. Multi-Device Support

The Problem

```
Kibuti has:
  Phone (android)    → alice@nexgate.com/android
  Tablet (tablet)    → alice@nexgate.com/tablet
  Web (browser)      → alice@nexgate.com/web

Both logged in simultaneously.
How do messages reach all devices?
How do sent messages sync across devices?
How does read status sync?
```

Solution — XEP-0280 Message Carbons

```
Message Carbons = automatic copy to all devices

Kibuti sends message from phone:
  Phone → Ejabberd → Bob 
  Ejabberd ALSO sends carbon copy to:
    Kibuti's tablet 
    Kibuti's web 

Result:
  All Kibuti's devices see sent messages 
  Conversation stays in sync 
  No extra code needed in Spring Boot
  Ejabberd handles it via mod_carbons

Enable in ejabberd.yml:
  modules:
    mod_carbons: {}
```

Resource Priority

```
Each device has a priority:
  android: priority 10 (highest — phone)
  tablet:  priority 5
  web:     priority 1 (lowest)

When Juma sends to Kibuti:
  Ejabberd delivers to HIGHEST priority device
  = android (phone) gets it first
  Carbon copies → all other devices

Priority set during session setup:
  <presence>
    <priority>10</priority>
  </presence>

NexGate mobile sets:
  Foreground app: priority 10
  Background app: priority 0
  Ejabberd routes to foreground device 
```

Read Status Sync Across Devices

```
Problem:
  Kibuti reads message on phone
  Tablet still shows unread badge

Solution — XEP-0333 Chat Markers sync via Carbons:
  Phone sends <displayed> to Juma
  Ejabberd carbons copy to tablet + web
  Tablet receives: "Kibuti read this on phone"
  Tablet clears unread badge □

This is automatic via mod_carbons □
```

XMPP JWT Per Device

```
Each device gets its OWN XMPP JWT:
  Phone connects → POST /auth/login → xmppJwt-1
  Tablet connects → POST /auth/refresh → xmppJwt-2
  Web connects → POST /auth/login → xmppJwt-3

Each JWT has same JID but different resource:
  { jid: "usr-kibuti@nexgate.com", exp: ... }

Ejabberd assigns resource automatically:
  usr-kibuti@nexgate.com/android  (phone)
  usr-kibuti@nexgate.com/tablet   (tablet)
  usr-kibuti@nexgate.com/web-abc  (browser)

On logout (one device):
  That device's JWT invalidated
  Other devices unaffected □
```

Active Sessions Tracking

```
Spring Boot tracks active sessions in Redis:
  Key: sessions:usr-kibuti
  Value: SET {
```

```
    "android:gajim.B3HKS5VE",
    "tablet:dino.36aa17c4"
  }
  TTL: 24 hours (JWT expiry)

When checking if user is online:
  Redis lookup → instant □
  No DB query needed □
  No Ejabberd REST call needed □

Updated by:
  PresenceConsumer → user online → add to SET
  PresenceConsumer → user offline → remove from SET
```

Multi-Device Message Archive (MAM)

```
XEP-0313 Message Archive Management:
  All messages stored in Ejabberd archive (PostgreSQL)
  New device login → fetches message history
  "Give me messages since last week"

New device flow:
  1. Kibuti logs in on new tablet
  2. Tablet connects to Ejabberd
  3. Tablet sends MAM query:
     <iq type="set">
       <query xmlns="urn:xmpp:mam:2">
         <x><field var="start">
           <value>2026-07-10T00:00:00Z</value>
         </field></x>
       </query>
     </iq>
  4. Ejabberd returns all messages since that date
  5. Tablet renders full conversation history □

Spring Boot also has messages in chat.messages:
  Mobile can fetch via REST: GET /chat/messages
  Paginated, sorted, filtered
  Two sources of truth:
```

```
    Ejabberd MAM: raw XMPP archive
    Spring Boot DB: structured business records
```

---

21. Redis — What to Cache & Why

The Golden Rule

```
Redis is for:
  Data that changes frequently
  Data read far more than written
  Data where 1-2 second staleness is OK
  Data that is expensive to compute

Redis is NOT for:
  Primary source of truth
  Data that must be 100% consistent
  Large objects (images, files)
  Data that needs complex queries
```

Cache 1 — Online Presence

```
Key:    presence:usr-kibuti
Value:  { status: "online", last_seen: timestamp, devices: [...] }
TTL:    5 minutes (refreshed by heartbeat)

Why Redis:
  Checked on EVERY message send
  "Is recipient online?" → Redis hit □
  Without Redis: Ejabberd REST call per message □
  At 7,000 msg/min: 7,000 Ejabberd REST calls/min □
  With Redis: 7,000 Redis reads/min (microseconds) □

Updated by:
  PresenceConsumer when user comes online
  PresenceConsumer when user goes offline
  Heartbeat pings every 60 seconds
```

Cache 2 — Active XMPP Sessions

```
Key:    sessions:usr-kibuti
Value:  SET of resource strings
TTL:    24 hours (JWT expiry)

Why Redis:
  Know which devices are connected
  Without Redis: Ejabberd REST call (user_resources)
  With Redis: instant SET lookup 

Updated by:
  PresenceConsumer on connect/disconnect
```

Cache 3 — Conversation Metadata

```
Key:    conv:conv-abc123
Value:  {
          name: "Business Friends",
          type: "GROUP",
          member_count: 47,
          last_message: "...",
          last_updated: timestamp
        }
TTL:    10 minutes

Why Redis:
  Inbox list loads conversation metadata for EVERY conversation
  User with 50 conversations → 50 DB reads 
  With Redis: 50 cache hits 
  Inbox load: <50ms vs 500ms 

Updated by:
  MessageConsumer when new message arrives
  GroupService when group changes
```

Cache 4 — Unread Count Per Conversation

```
Key:    unread:usr-kibuti:conv-abc123
Value:  integer (unread message count)
TTL:    none (persist until read)

Why Redis:
  Show unread badge on inbox
  Without Redis: COUNT query per conversation 
  With Redis: INCR / SET / GET (nanoseconds) 

Updated by:
  MessageConsumer: INCR when message arrives for user
  ReceiptConsumer: SET 0 when user sends <displayed>

Total unread across all conversations:
  Key: unread:total:usr-kibuti
  MessageConsumer: INCR
  Reset: SET 0 when inbox opened
```

Cache 5 — Offer Session State

```
Key:    offer:offer-abc123
Value:  {
          status: "PENDING",
          expires_at: timestamp,
          offer_price: 400000,
          public_price: 450000
        }
TTL:    30 minutes (offer validity)

Why Redis:
  Offer expiry check on every [Accept] tap
  Without Redis: DB read per check 
  With Redis: instant lookup 
  TTL = offer auto-expires at Redis level 

Updated by:
  OfferLifecycleService on state change
  Redis TTL handles expiry automatically
```

Cache 6 — Rate Limiting

```
Key:    ratelimit:msg:usr-kibuti
Value:  integer (message count in window)
TTL:    1 minute (sliding window)

Why Redis:
  Prevent message spam
  Without Redis: DB query per message 
  With Redis: INCR + TTL = instant 

Rules:
  Max 60 messages per minute per user
  Max 10 group invitations per day
  Max 5 concurrent call attempts

Implementation:
  INCR ratelimit:msg:usr-kibuti
  EXPIRE ratelimit:msg:usr-kibuti 60
  If value > 60: reject with 429
```

Cache 7 — TURN Credentials

```
Key:    turn:usr-kibuti
Value:  { username, credential, ttl, urls }
TTL:    1 hour

Why Redis:
  User may request TURN credentials multiple times
  (call fails, retry, reconnect)
  Without Redis: HMAC compute per request
  With Redis: cache for 1 hour 
  Same credentials valid for 1 hour anyway
```

Cache 8 — User Profile (For Chat Display)

```
Key:    profile:usr-kibuti
Value:  {
```

```
          display_name: "Kibuti Mwangi",
          avatar_url: "https://cdn.nexgate.com/...",
          username: "kibuti",
          is_verified: false
        }
TTL:    30 minutes

Why Redis:
  Chat inbox shows sender name + avatar
  50 conversations × profile = 50 DB reads 
  With Redis: 50 cache hits 

Updated by:
  User updates profile → invalidate cache
  Cache miss → read from core.users → cache
```

Cache 9 — Group Membership (Hot Groups)

```
Key:    group:members:conv-abc123
Value:  SET of user JIDs
TTL:    5 minutes

Why Redis:
  Fan-out notification: "who is in this group?"
  For push notifications when group member offline
  Without Redis: DB query per group message 
  With Redis: SMEMBERS in microseconds 

Only cache for ACTIVE groups (>10 msg/hour)
Inactive groups: read from DB directly
```

Redis Key Naming Convention

```
Pattern: {domain}:{entity}:{id}:{sub}

Examples:
  presence:usr-kibuti
  sessions:usr-kibuti
```

```
  conv:conv-abc123
  unread:usr-kibuti:conv-abc123
  unread:total:usr-kibuti
  offer:offer-abc123
  ratelimit:msg:usr-kibuti
  ratelimit:invite:usr-kibuti
  turn:usr-kibuti
  profile:usr-kibuti
  group:members:conv-abc123

NEVER store:
  Raw message content (use DB)
  Files or media (use MinIO)
  Financial data (use DB with audit)
  Auth tokens as plain text (use Vault)
```

Cache Invalidation Strategy

```
Write-through (update cache + DB together):
  Unread counts ← must be accurate
  Offer status ← critical for commerce

Cache-aside (read DB on miss, then cache):
  User profiles ← changes rarely
  Conversation metadata ← OK if slightly stale
  Group membership ← OK if 5 min stale

TTL-based eviction (let it expire):
  Presence ← Ejabberd handles truth
  TURN credentials ← expire with JWT
  Rate limits ← sliding window

Event-based invalidation:
  User updates profile → DEL profile:usr-{id}
  Group settings change → DEL conv:{id}
  Member joins group → SADD group:members:{id}
  Member leaves group → SREM group:members:{id}
```

---

22. Best Practices & Anti-Patterns

? DO — Message Persistence

```
DO: Persist via RabbitMQ consumer (async)
  Message arrives → Ejabberd delivers → fires event
  Consumer persists to DB asynchronously
  User experience not blocked by DB write □

DON'T: Persist via synchronous API call
  Message arrives → Spring Boot REST call → DB
  DB write blocks message delivery □
  If DB slow → messages delayed □
  If DB down → messages lost □
```

? DO — Online Status Checks

```
DO: Check Redis for online status
  MessageConsumer:
    val isOnline = redis.exists("presence:$userId")
    if (!isOnline) sendFCM()

DON'T: Call Ejabberd REST for every message
  val resources = ejabberd.getUserResources(userId)
  if (resources.isEmpty()) sendFCM()
  ← This is an HTTP call per message □
  ← At scale: thousands of HTTP calls/min □
```

? DO — Stanza Building

```
DO: Build stanzas in dedicated StanzaBuilder class
  val stanza = StanzaBuilder.productCard(product)
  ejabberdClient.sendStanza(from, to, stanza)

DON'T: Build XML strings manually scattered in code
  val xml = "<message><nexgate-commerce>..." □
  ← Error prone, hard to test, hard to maintain
```

? DO — Offer Expiry

```
DO: Use Redis TTL for expiry detection
  redis.set("offer:$offerId", offer, ttl = 30.minutes)
  // TTL fires → OfferExpiryListener → update DB

DON'T: Use scheduled DB queries
  @Scheduled(fixedRate = 60000)
  fun expireOffers() {
    db.query("SELECT * FROM offers WHERE expires_at < NOW()")
  }
  ← Hammers DB every minute □
  ← Redis TTL is instant and free □
```

? DO — Conversation Updates

```
DO: Update conversation in MessageConsumer
  MessageConsumer:
    1. Persist message
    2. Update conversation.last_message (DB)
    3. INCR unread count (Redis)
    4. Update conv metadata cache (Redis)

DON'T: Fetch conversation on every message render
  GET /chat/conversations → DB query every time □
  Serve from Redis cache □
```

? DO — JID Handling

```
DO: Always strip resource for storage
  Store: "alice@nexgate.com" (bare JID)
  Never: "alice@nexgate.com/android" (full JID)

  val bareJid = fullJid.substringBefore("/")

DON'T: Store full JID with resource
  Resource changes between sessions □
  Queries will break □
```

```
Exception: call routing (need specific device)
  Full JID used only for Jingle call signaling
  Not stored in DB
```

? DO — Always Include Fallback

```
DO: Include <body> fallback in EVERY message
  <message>
    <body>Check out this product</body>  ← fallback □
    <ng xmlns="urn:nexgate:1">
      <meta>...</meta>
      <payload>...</payload>
    </ng>
  </message>

DON'T: Send message without body
  <message>
    <ng xmlns="urn:nexgate:1">
      ...
    </ng>
  </message>
  ← Basic XMPP clients show nothing □
  ← Debugging impossible □
  ← Violates NexGate stanza standard □
```

? DO — Group Message Stanza ID

```
DO: Use stanza-id (room-assigned) for group references
  // Reacting to a group message:
  val stanzaId = message.getStanzaId()  // by="room@..."
  buildReaction(stanzaId, emoji)

DON'T: Use origin-id for group message references
  val originId = message.getOriginId()  // client-generated
  buildReaction(originId, emoji)  ← WRONG for groups □

Rule:
```

```
  1:1 chat → reactions/edits reference origin-id
  Group chat → reactions/edits reference stanza-id
```

? DO — Shop Staff Privacy

```
DO: Always strip staff resource from JID
  Outbound to customer:
    from = "shop-456@nexgate.com"  □
    (resource stripped by StanzaBuilder)

DON'T: Let staff JID leak to customer
    from = "shop-456@nexgate.com/staff-john"  □
    Customer now knows staff name □
    Privacy violation □

Implementation:
  StanzaBuilder always strips resource
  for any stanza from a shop account
```

? DO — Rate Limiting

```
DO: Rate limit at Redis level before processing
  fun sendMessage(userId: String, ...) {
    val key = "ratelimit:msg:$userId"
    val count = redis.incr(key)
    if (count == 1L) redis.expire(key, 60)
    if (count > 60) throw RateLimitException()
    // proceed with message
  }

DON'T: Rate limit at DB level
  val recentCount = db.query(
    "SELECT COUNT(*) FROM messages
     WHERE user_id = ? AND created_at > NOW() - INTERVAL '1 minute'"
  )
  ← DB query per message □
  ← Slow and expensive □
```

? DO — Typing Indicators

```
DO: Add no-store hint to typing stanzas
  <message type="chat">
    <composing xmlns="http://jabber.org/protocol/chatstates"/>
    <no-store xmlns="urn:xmpp:hints"/>  ← ALWAYS include
  </message>

DON'T: Let typing indicators get stored in MAM
  Without no-store: Ejabberd archives the typing indicator
  New device fetches history: sees "alice is typing..."
  from 3 weeks ago ← makes no sense □
```

? DO — Offline Message Handling

```
DO: Let Ejabberd handle offline queuing + Redis for FCM decision
  Ejabberd stores offline messages natively
  Spring Boot checks Redis for online status
  If offline → FCM via notification/

DON'T: Build your own offline queue in DB
  val offlineMessages = db.save(message)  □
  Duplicates Ejabberd's built-in offline storage □
  Two sources of truth □
```

? DO — Gzip Compression

```
DO: Enable Gzip on ALL REST responses
  # application.properties
  server.compression.enabled=true
  server.compression.min-response-size=1024
  server.compression.mime-types=application/json,application/xml

Result:
  Chat API responses: 25KB → 6KB □
  Inbox load on 2G: 1.25s → 0.3s □
  Zero code change required □
```

```
DON'T: Send uncompressed JSON to mobile on 2G
  25KB per inbox load × 10 loads/day
  = 250KB/day just for inbox
  On Tanzania 2G data plans: expensive □
```

Anti-Pattern — DB Poll for Events

```
NEVER DO THIS:
  @Scheduled(fixedRate = 1000)
  fun checkForNewMessages() {
    val messages = db.query(
      "SELECT * FROM messages WHERE delivered_at IS NULL"
    )
    messages.forEach { deliver(it) }
  }
  ← Polls DB every second □
  ← At scale: crushes DB □
  ← High latency (up to 1 second delay) □

DO THIS INSTEAD:
  RabbitMQ consumer:
    Ejabberd fires event → consumer reacts instantly □
    Zero polling □
    Sub-millisecond reaction time □
```

Anti-Pattern — N+1 Queries in Inbox

```
NEVER DO THIS:
  val conversations = db.getConversations(userId)
  conversations.forEach { conv ->
    conv.lastMessage = db.getLastMessage(conv.id)  // N queries □
    conv.unreadCount = db.getUnreadCount(conv.id)  // N queries □
    conv.memberNames = db.getMemberNames(conv.id)  // N queries □
  }
  50 conversations = 150 DB queries □

DO THIS INSTEAD:
  // Redis for unread counts (O(1) per key)
```

```
  val unreadCounts = redis.mget(convIds.map { "unread:$userId:$it" })

  // Single JOIN query for last messages
  val conversations = db.query("""
    SELECT c.*, m.body as last_body, m.created_at as last_at
    FROM chat.conversations c
    LEFT JOIN chat.messages m ON m.id = c.last_message_id
    WHERE c.user_id = ?
    ORDER BY c.updated_at DESC
    LIMIT 50
  """)

  // Profiles from Redis
  val profiles = redis.mget(userIds.map { "profile:$it" })
```

Anti-Pattern — Blocking on Ejabberd REST

```
NEVER DO THIS (in request path):
  fun sendMessage(req: MessageRequest): Response {
    val result = ejabberdClient.sendMessage(...)  // HTTP call
    db.saveMessage(...)
    return Response.ok()
  }
  ← HTTP call in request path 
  ← If Ejabberd slow → user waits 
  ← If Ejabberd down → request fails 

DO THIS INSTEAD:
  fun sendMessage(req: MessageRequest): Response {
    // Validate + queue immediately
    val messageId = db.saveOutgoing(req)
    rabbitMQ.publish("chat.outbound", req)
    return Response.ok(messageId)  // instant 
  }

  // Async consumer sends to Ejabberd
  @RabbitListener(queues = "chat.outbound")
  fun sendToEjabberd(req: MessageRequest) {
    ejabberdClient.sendMessage(...)  // in background 
```

```
}
```

23. Package Structure

Full chat/ Package

```
chat/
  ├── config/
  │     EjabberdConfig.java        ← Ejabberd connection settings
  │     RabbitMQChatConfig.java    ← queues, exchanges, bindings
  │     LiveKitConfig.java         ← LiveKit server settings
  │     CoturnConfig.java          ← TURN server settings
  │
  ├── controller/
  │     ConversationController.java  ← GET /chat/conversations
  │     MessageController.java       ← GET /chat/messages
  │     GroupController.java         ← POST /chat/groups/create
  │     CallController.java          ← GET /chat/calls/turn-credentials
  │     ReceiptController.java       ← POST /chat/messages/{id}/receipt
  │
  ├── service/
  │     ConversationService.java     ← conversation lifecycle
  │     MessageService.java          ← persist, query messages
  │     GroupService.java            ← group management
  │     OfferSessionService.java     ← offer lifecycle
  │     CallService.java             ← call records, TURN creds
  │     PresenceService.java         ← online status tracking
  │
  ├── consumer/
  │     MessageConsumer.java         ← handles chat.message.*
  │     PresenceConsumer.java        ← handles chat.presence.*
  │     CallConsumer.java            ← handles chat.call.*
  │     RoomConsumer.java            ← handles chat.room.*
  │
  ├── ejabberd/
  │     EjabberdRestClient.java      ← HTTP client for Ejabberd API
  │     EjabberdAuthController.java  ← POST /internal/ejabberd/auth
```

```
  │     JwksController.java          ← GET /auth/.well-known/jwks.json
  │     XmppTokenService.java        ← generate/validate XMPP JWT
  │
  ├── stanza/
  │     StanzaBuilder.java           ← builds XMPP stanzas
  │     CommerceStanza.java          ← product card, offer stanzas
  │     GroupBuyStanza.java          ← bei ya pamoja stanzas
  │     SystemStanza.java            ← order update stanzas
  │     CallStanza.java              ← group call join stanzas
  │     MediaStanza.java             ← file/image/voice stanzas
  │
  ├── commerce/
  │     CommerceDmService.java       ← initiate commerce DM
  │     OfferLifecycleService.java   ← offer state machine
  │     GroupBuyService.java         ← bei ya pamoja flows
  │     ShopInboxService.java        ← shop routing, staff access
  │
  └── call/
        TurnCredentialService.java   ← HMAC TURN credentials
        LiveKitService.java          ← group call room management
        CallRecordService.java       ← persist call records
```

---

24. Infrastructure Stack

Docker Containers (Local Development)

Container	Port(s)	Purpose
ejabberd	5222, 5280	XMPP server
coturn	3478	STUN/TURN relay
livekit	7880, 7881	Group calls + Audio Spaces
nexgate_postgres	5432	Main platform DB
ft_postgres	5433	File Thunder DB
ejabberd_postgres	5434	Ejabberd DB (production)
rabbitmq	5672, 15672	Message queue + management
redis	6379	Cache, sessions, rate limits
minio	9000, 9001	Object storage

```
ft_clamav             3310             Virus scanning

Local processes:
  nexgate_backend     8080             Spring Boot (all-in-one)
  file_thunder        8084             Media engine
```

Ejabberd Configuration Summary

```
ejabberd.yml key settings:
  hosts: ["nexgate.com"]
  auth_method: jwt
  jwt_key: https://api.nexgate.com/auth/.well-known/jwks.json

  listen:
    port 5222: ejabberd_c2s (XMPP + TLS)
    port 5280: ejabberd_http (/api, /admin, /oauth, /ws)

  modules:
    mod_muc: group chats
    mod_mam: message archive
    mod_offline: offline message storage
    mod_ping: keepalive
    mod_stream_mgmt: XEP-0198 reliability
    mod_rabbitmq: event publishing

  default_db: sql (production)
  sql_type: pgsql
  sql_server: "ejabberd_postgres"
  sql_database: "ejabberd"
```

Ejabberd ? Spring Boot Security

```
Channel 1 (Spring Boot → Ejabberd REST):
  Admin credentials in HashiCorp Vault
  Internal Docker network only
  Traefik blocks external access to 5280

Channel 2 (Ejabberd → Spring Boot via RabbitMQ):
  RabbitMQ credentials in Vault
```

```
  Internal Docker network only
  AMQP not exposed externally

Channel 3 (Mobile → Ejabberd JWT):
  JWT signed RS256 (private key in Vault)
  Public key via JWKS endpoint (HTTPS only)
  XMPP port 5222 with TLS
  JWT expires 24 hours
```

---

25. Build Order

Week 1 — Foundation

```
Day 1-2: JWT Auth Infrastructure
  EjabberdAuthController.java
    POST /internal/ejabberd/auth
    Validate XMPP JWT, return 200/401

  JwksController.java
    GET /auth/.well-known/jwks.json
    Return RSA public key

  XmppTokenService.java
    Generate XMPP JWT on login
    Add to auth/LoginResponse

  Configure Ejabberd:
    auth_method: jwt
    jwt_key: JWKS URL

  Test: Dino/Gajim connects via JWT □

Day 3-4: RabbitMQ Event Pipeline
  RabbitMQChatConfig.java
    Declare exchange, queues, bindings

  MessageConsumer.java
```

```
    Listen to chat.message.inbound
    Persist to chat.messages

  PresenceConsumer.java
    Listen to chat.presence.*
    Update last_seen

  Test: Send message in Gajim → appears in DB □

Day 5: Conversation Management
  ConversationService.java
    Find or create conversation
    Update last_message

  ConversationController.java
    GET /chat/conversations (inbox)
    GET /chat/conversations/{id}/messages
```

Week 2 — Core Messaging

```
Day 1-2: Message Interactions
  Handle edit stanzas (XEP-0308)
  Handle retract stanzas (XEP-0424)
  Handle reaction stanzas (XEP-0444)
  Handle reply stanzas (XEP-0461)
  MessageService.java for each

Day 3: Group Chat
  GroupController.java
    POST /chat/groups/create
    POST /chat/groups/{id}/join
    POST /chat/groups/{id}/decline
    DELETE /chat/groups/{id}/leave

  GroupService.java
    Ejabberd REST: create_room, add_member
    Invite link management

Day 4-5: Receipts & Notifications
```

```
  ReceiptController.java
    POST /chat/messages/{id}/receipt
    { status: DELIVERED|READ }

  Offline push:
    MessageConsumer → detect offline
    Call notification/ package
    FCM push □
```

Week 3 — Commerce DMs

```
Day 1-2: Commerce Stanza Sending
  EjabberdRestClient.java
    sendStanza(from, to, body, customElement)

  StanzaBuilder.java
    buildProductCard(product)
    buildOfferCard(offer)
    buildSystemMessage(order)

  CommerceDmService.java
    initiateCommerceDm(productId, shopId, buyerId)

Day 3-4: Offer Session Lifecycle
  OfferLifecycleService.java
    createOffer(...)
    acceptOffer(offerId)
    declineOffer(offerId)
    expireOffers() ← scheduled job

Day 5: Shop Inbox
  ShopInboxService.java
    Route to available staff
    Strip staff resource from JID
    Audit log staff actions
```

Week 4 — Calls

```
Day 1-2: TURN Credentials
  TurnCredentialService.java
    Generate HMAC-SHA256 credentials
    Time-limited (1 hour)

  CallController.java
    GET /chat/calls/turn-credentials

Day 3-4: Group Calls (LiveKit)
  LiveKitService.java
    createRoom(callId)
    generateToken(userId, roomId)

  CallStanza.java
    buildGroupCallJoinInfo(callId, token, url)

  Spring Boot sends join info stanza
  Members receive → join LiveKit room

Day 5: Call Records
  CallRecordService.java
    onCallInitiated(event)
    onCallEnded(event)
    getCallHistory(userId)
```

Week 5 — Polish & Testing

```
  Message search (MAM queries to Ejabberd)
  Read count for group messages
  Block/unblock users
  Message request system
  Integration testing with Gajim/Dino
  Load testing with sendxmpp
  Documentation for mobile dev team
```

NexGate Chat System — Development Architecture Guide v9.0 QBIT SPARK | chat_system · Spring Boot · Ejabberd · RabbitMQ · Redis · WebRTC · Coturn · LiveKit · OMEMO

Revision #9
Created 17 July 2026 09:53:35 by Admin Qbit
Updated 19 July 2026 09:24:13 by Admin Qbit