XMPP From Zero to Production Book

A Complete Developer's Guide

Build real-time messaging that survives 2G — from your first stanza to a clustered Ejabberd deployment.

*Running example throughout: **MaasaiChat**, a chat app for Maasai communities across rural Tanzania and Kenya — 2G/3G links, low-end Android phones, connections that drop constantly. Every technical decision in this book is shaped by that reality.*

First Edition · 2026 Written for backend engineers, mobile developers, and architects who need to own their messaging stack.

No prior XMPP knowledge assumed. By the final chapter you will have deployed a production Ejabberd server, implemented every XEP MaasaiChat needs on Android and iOS, and hardened it for real traffic.

How to use this book

- **Part 1** teaches the protocol from absolute zero. Read it in order.
- **Part 2** is a reference — one chapter per XEP, all built on the same six-part template. Read the Core and Messaging chapters, then dip into the rest as you need them.
- **Part 3** is hands-on Ejabberd. Keep a terminal open.
- **Part 4** is the mobile SDK guide (Android Smack, iOS XMPPFramework).
- **Part 5** is production: security, database, scaling, network tuning, pitfalls.

Every XEP chapter answers the same six questions in the same order:

```
1. What problem does it solve?
2. The XML stanzas, explained line by line
3. A real MaasaiChat example
4. How Ejabberd handles it
5. What the mobile dev implements
```

```
6. What the backend dev implements
```

Every chapter ends with a **What you learned** box and a checkpoint before the next one.

Conventions: terminal commands are shown in fenced blocks, XML stanzas are complete and copy-pasteable, and the four MaasaiChat users appear in every example:

```
ole@maasaichat.com        Ole Saitoti   — elder
naserian@maasaichat.com   Naserian      — young warrior
enkiama@maasaichat.com    Enkiama       — village chief
nkeri@maasaichat.com      Nkeri         — cattle trader
```

Table of Contents

Part 1 — Foundation

Part 2 — XEPs Complete Reference

```
15  XEP-0184  Message Delivery Receipts
16  XEP-0333  Chat Markers
17  XEP-0308  Last Message Correction
18  XEP-0424  Message Retraction
19  XEP-0444  Message Reactions
20  XEP-0297  Stanza Forwarding
21  XEP-0461  Message Replies
22  XEP-0359  Stable & Unique Stanza IDs
23  XEP-0334  Message Processing Hints

    Group Chat
24  XEP-0045  Multi-User Chat (MUC)
25  XEP-0249  Direct MUC Invitations
26  XEP-0317  Hats
27  XEP-0425  Message Moderation
28  XEP-0490  Message Displayed Synchronization

    File & Media
29  XEP-0363  HTTP File Upload           (the one you'll actually use)
30  XEP-0065  SOCKS5 Bytestreams
31  XEP-0234  Jingle File Transfer
32  XEP-0264  Jingle Content Thumbnails

    Calls
33  XEP-0166  Jingle
34  XEP-0167  Jingle RTP Sessions
35  XEP-0176  Jingle ICE-UDP Transport
36  XEP-0177  Jingle Raw UDP Transport
37  XEP-0215  External Service Discovery (TURN/STUN credentials)
38  XEP-0320  DTLS-SRTP in Jingle

    Push & Notifications
39  XEP-0357  Push Notifications         (FCM on Android)

    Security & Encryption
40  XEP-0384  OMEMO Encryption
41  XEP-0388  Extensible SASL Profile
42  XEP-0440  SASL Channel Binding
```

```
   Presence & Roster
43 XEP-0054  vcard-temp
44 XEP-0153  vCard-Based Avatars
45 XEP-0292  vCard4 Over XMPP
46 XEP-0083  Nested Roster Groups
47 XEP-0144  Roster Item Exchange

   PubSub
48 XEP-0060  Publish-Subscribe
49 XEP-0163  Personal Eventing Protocol (PEP)

   History & Archive
50 XEP-0059  Result Set Management
51 XEP-0313  MAM in depth (querying, paging)
52 XEP-0430  Inbox

   Enterprise
53 XEP-0050  Ad-Hoc Commands
54 XEP-0004  Data Forms
55 XEP-0055  Jabber Search
56 XEP-0077  In-Band Registration
57 XEP-0133  Service Administration

   Federation
58 XEP-0220  Server Dialback
59 XEP-0288  Bidirectional Server-to-Server
```

Part 3 — Ejabberd in Practice

```
60 Architecture & the Erlang/BEAM Foundation
61 Docker Setup, Step by Step
62 ejabberd.yml — The Complete Guide
63 ejabberdctl — Every Command You'll Use
64 REST API — Complete Reference
65 OAuth Authentication
66 The Auth Bridge (HTTP auth → your backend as gatekeeper)
67 Clustering Two Nodes
68 MUC Administration
69 Monitoring & Logging
```

Part 4 — Mobile SDK Guide

```
70  Android with Smack — connect, auth, send/receive, all XEPs, custom stanzas
71  iOS with XMPPFramework — same coverage
```

Part 5 — Production

```
72  Security Hardening
73  PostgreSQL Backend
74  Scaling Path
75  East African Network Optimization
76  Common Pitfalls
77  Monitoring Setup
```

Part 1 — Foundation

Chapter 1 — What is XMPP?

1.1 The one-sentence answer

XMPP is an open protocol for sending small pieces of XML from one address to another, in real time, over a long-lived connection.

That's it. Everything else in this book is detail on top of that sentence. Each word was chosen deliberately:

- **Open protocol** — nobody owns it. It's an IETF standard (RFC 6120 for the core, RFC 6121 for instant messaging and presence). You don't ask permission or pay a license. Contrast this with WhatsApp's protocol, which is closed — you cannot legally build a server that speaks it.
- **Small pieces of XML** — the unit of communication is a *stanza*, a short XML fragment. A single chat message is a few hundred bytes.
- **From one address to another** — every user and service has an address called a **JID** (Jabber ID). Routing is built into the protocol.
- **In real time** — messages arrive the instant they're sent, not when the client next polls.

- **Over a long-lived connection** — the phone opens one TCP connection to the server and keeps it open for hours. Messages flow both ways over that single pipe.

XMPP originally stood for *eXtensible Messaging and Presence Protocol*. It started in 1999 under the name **Jabber**, created by Jeremie Miller. You'll still see "Jabber" everywhere — in the JID name, in library names, in old docs. Treat "Jabber" and "XMPP" as the same thing.

1.2 Why a long-lived connection matters (and why HTTP doesn't fit)

To understand XMPP, understand the problem it was built to avoid.

The web runs on HTTP, which is **request/response**. The client asks, the server answers, the connection closes. Perfect for loading a web page. Terrible for chat, because chat is *server-initiated* : the server needs to push a message to you the moment someone sends it, and it has no idea when that will be.

There are three ways to force chat onto HTTP, and all three are bad on a rural network:

```
Approach          How it works                       Problem on 2G/3G
-------------------------------------------------------------------------
Short polling     Ask "any messages?" every 3s       Wastes battery + data,
                                                      message lag up to 3s
Long polling      Ask, server holds request open     Constant reconnects,
                  until a message arrives, repeat     each carries TCP+TLS cost
WebSocket over    One socket, but you build the       You reinvent routing,
raw HTTP          entire chat protocol yourself       presence, offline, MUC...
```

XMPP solves this at the protocol level. The phone opens **one** connection, authenticates **once**, then both sides send stanzas whenever they want. No repeated handshakes. No polling. On a 2G link where a new TCP+TLS handshake costs several round-trips over 300–800 ms latency, "connect once and stay connected" is the difference between a usable app and an unusable one.

```
   MaasaiChat phone                        maasaichat.com server
   (Naserian's Tecno)                      (Ejabberd)
        |                                        |
        |------ open TCP connection ------------->|
        |<----- keep it open, both directions -----|
        |                                        |
        |  message to ole@maasaichat.com   ------->|   routes it
```

```
        |                                          |
        |<------  message from enkiama@... ---------|   pushed instantly
        |                                          |
        |            (connection stays open        |
        |             for hours)                   |
```

One pipe. Messages flow both directions. That's the core idea.

1.3 What actually travels down the pipe

Once connected, the phone and server exchange **stanzas**. There are exactly three kinds (Chapter 4 goes deep). For now, just see them.

A chat message from Naserian to the elder Ole:

```
<message from='naserian@maasaichat.com/tecno'
         to='ole@maasaichat.com'
         type='chat'>
  <body>Elder, the cattle are safe at the river.</body>
</message>
```

Naserian telling the server she is online:

```
<presence from='naserian@maasaichat.com/tecno'>
  <show>chat</show>
  <status>Herding near Ngorongoro</status>
</presence>
```

Naserian asking the server a question and expecting an answer:

```
<iq from='naserian@maasaichat.com/tecno'
    type='get'
    id='disco1'>
  <query xmlns='http://jabber.org/protocol/disco#info'/>
</iq>
```

Three stanza types — `message`, `presence`, `iq` — and everything XMPP does is one of those three, possibly with extra XML tucked inside. That extra XML is what a XEP is, and it's why XMPP can grow reactions, calls, and file upload without ever changing the core (Chapter 5).

1.4 XMPP is the rulebook, Ejabberd is the builder

Here is the single most common confusion for newcomers, and clearing it up now makes the rest of the book effortless: **XMPP and Ejabberd are not the same thing, and one is not "inside" the other.**

XMPP is a **specification** — a set of documents written by the XMPP Standards Foundation. It is pure guidance. It says things like "*a chat message must have a* `<body>` *element,*" "*addresses look like* `user@domain/resource`," "*a typing indicator uses this namespace.*" That's all it is. Rules on paper. The XMPP spec, by itself, cannot route a single message — in the same way a rulebook cannot play the game.

Ejabberd is a **builder that read those rules and wrote the code.** The team at ProcessOne read every relevant document and implemented it in Erlang, producing an actual running server that obeys the rules. Ejabberd is the thing that accepts connections, routes stanzas, and stores offline messages.

The cleanest way to hold this:

```
    Building code (the rules)          The builder (follows the rules)
    -------------------------          -------------------------------
    "walls must be 30cm thick"    -->  reads the code,
    "doors must be 2m high"            actually builds the house,
    "foundation must be concrete"      following every instruction

    The building code is just paper.   The house is real and you
    It builds nothing by itself.       can live in it.

    XMPP = the building code           Ejabberd = the builder
```

Developers already know this pattern from tools they use every day:

```
    The language / rules      The thing that speaks it
    --------------------      ------------------------
    HTTP                 -->  Nginx, Apache, Tomcat
    SQL                  -->  PostgreSQL, MySQL
    XMPP                 -->  Ejabberd, Prosody, Openfire
```

Nobody says "HTTP is inside Nginx." HTTP is the rulebook; Nginx is a program that follows it. XMPP and Ejabberd relate the exact same way.

The chain: who writes the rules, who builds

Because XMPP is *open* guidance, many different teams read the same documents and each build their own piece — and because they all follow the same rules, all the pieces interoperate:

```
XMPP Standards Foundation     writes the guidance (the XEPs)
     │                        "delivery receipts work like THIS"
     ▼
ProcessOne                    reads it, writes Erlang  → Ejabberd (server)
Gajim team                    reads it, writes Python  → Gajim  (desktop client)
Smack team                    reads it, writes Java    → Smack  (Android library)
sendxmpp author               reads it, writes Perl    → sendxmpp (CLI tool)

Different code. Same guidance. They all understand each other. 
```

This is why the Gajim desktop app on Ole's laptop, the Smack-powered MaasaiChat app on Naserian's Tecno, and a `sendxmpp` script on a server can all exchange messages through Ejabberd without anyone coordinating — they are all following the same rulebook.

And it's the reason **you write almost no protocol code yourself.** Ejabberd is already built. Smack is already built. You read the XEPs to *understand*, then use those implementations, and only write custom code for MaasaiChat's own extensions (its own namespaces for things the standard doesn't cover). The 25-year-old rulebook, and the battle-tested builders who followed it, do the rest.

```
You (MaasaiChat)     read XEPs to understand the rules
                     use Ejabberd  (server, already built)
                     use Smack     (Android, already built)
                     write custom code ONLY for your own
                       app-specific extensions 
```

> **Catch it in one line:** *XMPP is the language, Ejabberd speaks it, Gajim and Smack also speak it — everyone understands each other because everyone follows the same rulebook.*

1.5 Who uses XMPP today

XMPP is not a museum piece. It quietly runs a large chunk of the messaging world:

- **WhatsApp** was built on a heavily modified fork of Ejabberd — the exact server this book deploys. The protocol you're learning is the ancestor of the app two billion people use.
- **Nintendo Switch** uses XMPP/Ejabberd for online chat and presence.
- **Zoom** acquired the XMPP-based team-chat product now sold as Zoom Team Chat.
- **Google Talk** was XMPP for its entire life and even federated with outside XMPP servers.
- **Jitsi, HCL Sametime, and many carrier, ISP, and government messaging systems** run on XMPP today.

The common thread: when an organization needs *self-hosted, standards-based, massively scalable* real-time messaging that they fully control, XMPP keeps being the answer. That is exactly MaasaiChat's position — you cannot depend on someone else's servers for a community tool in rural Tanzania and Kenya, so you run your own.

1.6 The messaging landscape — XMPP and its alternatives

XMPP is not the only way to build a chat app. Before committing, you should know what else exists, who runs on it, and the honest trade-offs. There are six real families of choice.

1. XMPP (this book)

Open IETF standard. Servers: **Ejabberd** (what we use), Prosody, Openfire. Clients everywhere.

Pros	Cons
Open standard, no vendor lock-in	Learning curve — it's a real protocol with real depth
Self-hosted, you own the data	XML is verbose vs binary (mitigated by compression)
Routing, presence, offline, MUC, archive, push are built in	Some XEPs are optional/uneven across servers
Federation across servers	You assemble the client stack
Proven to millions of connections	
Free (Ejabberd Community)	

Who uses it: WhatsApp (originally), Nintendo Switch, Google Talk, Jitsi.

2. Matrix

The main modern open-standard rival. Server: **Synapse** (also Dendrite, Conduit). Client: **Element**. Instead of XMPP's live XML stream, Matrix syncs a replicated JSON event graph over HTTP — every message is an event, and history is a shared, eventually-consistent room state.

Pros	Cons
Open standard, federated	Heavier — Synapse is resource-hungry vs Ejabberd
Strong built-in E2E encryption	
JSON over HTTP — familiar to devs	Sync model uses more bandwidth, worse fit for strict 2G budgets
Great for team/community chat	
Rich ecosystem, bridges galore	Younger, protocol still evolving

Who uses it: the **French government** (Tchap), the **German armed forces** (BwMessenger), **Mozilla**, **KDE**, and many privacy-focused communities. It's the serious open alternative — but its "replay the room's event history" model is more bandwidth-hungry than XMPP's lean stanza stream, which matters when your users pay per megabyte on 2G. That single fact is a large part of why MaasaiChat chooses XMPP over Matrix.

3. MQTT

A lightweight publish/subscribe protocol from the IoT world. Broker: **Mosquitto**, EMQX, HiveMQ.

Pros	Cons
Extremely lightweight wire format	Not a chat protocol — no roster, presence, offline history, MUC
Tiny overhead, ideal for low bandwidth / battery	You build ALL chat semantics on top yourself
Great pub/sub fan-out	
Simple to reason about	No federation, no identity model

Who uses it: sensors, cars, smart devices — and famously **Facebook Messenger**, which used MQTT for years to get fast, low-overhead delivery on poor mobile networks. MQTT is a fantastic *transport*, but it gives you a pipe, not a chat system. You'd rebuild everything XMPP already provides. (Note: Ejabberd itself speaks MQTT natively, so you can even use both.)

4. Proprietary binary protocols

The big consumer apps mostly rolled their own closed protocols:

- **WhatsApp** — started on Ejabberd/XMPP, later moved to a custom binary protocol using the **Noise Protocol Framework** for its handshake, with the **Signal Protocol** for end-to-end encryption.
- **Signal** — the **Signal Protocol** over its own service; the gold standard for E2E encryption.
- **Telegram** — its own **MTProto** protocol.
- **Discord** — a custom **WebSocket gateway** protocol, backed by Elixir/Erlang (the same platform Ejabberd runs on).

Pros	Cons
Fully optimized for one app	You must design + maintain the entire protocol yourself
Smallest possible wire format	No standard, no federation
Total control	Years of engineering
	Wrong choice unless you're at massive scale with a big team

Who uses it: WhatsApp, Signal, Telegram, Discord. Great if you're a well-funded platform. Not a starting point for a community app.

5. Hosted / Backend-as-a-Service

Buy chat as an API. Firebase (Firestore + Cloud Messaging), **Stream**, **Sendbird**, **PubNub**, **Twilio Conversations**.

Pros	Cons
Fastest to ship	Pay per user / per message forever
No servers to run	You don't own the data
Handles scale for you	Vendor can change pricing or cut you off
Nice SDKs	Data lives outside your country
	Costs balloon as you grow

Who uses it: startups that want chat live this week. Wrong fit for a self-reliant community tool where every user is cost-sensitive and independence is the point — a pricing change in San Francisco should never be able to shut down messaging in Ngorongoro.

6. Raw WebSocket + your own protocol

Open a WebSocket, invent your own message format, build the rest by hand.

Pros	Cons
Total freedom	You reimplement routing, presence, offline, groups, receipts, archive, reconnection — for years
Simple to start ("just a socket")	No standard, no interoperability
Familiar to web devs	

```
                    Every bug is yours to discover
```

Who uses it: Slack and Discord built custom protocols *on top of* WebSocket — but with large engineering teams. For a solo or small team, this is the "reinvent XMPP, badly" path.

The verdict for MaasaiChat

	Own data & control	Low 2G bandwidth	Built-in chat feats	Free / cheap	Effort to ship
XMPP/Ejabberd	YES	Excellent	YES	YES	Medium
Matrix	YES	Fair	YES	Cheap-ish	Medium
MQTT	YES	Excellent	NO	YES	High
Proprietary	YES	Best	NO	No	Very high
Hosted (SaaS)	NO	Varies	YES	No	Low
Raw WebSocket	YES	Good	NO	YES	Very high

XMPP is the only row that is **yes** on ownership, **excellent** on bandwidth, **yes** on built-in chat features, and **free** — at merely *medium* effort. For a self-hosted community app on 2G in East Africa, no other option matches on all four. That is why the rest of this book is XMPP.

1.7 Why XMPP fits rural East Africa specifically

Every decision in this book is shaped by one reality: MaasaiChat users are on **2G/3G, low-end Android, with connections that drop constantly**. XMPP earns its place for concrete reasons.

Tiny messages. A stanza is a few hundred bytes. On a metered 2G plan where users pay per megabyte, that matters. No fat envelopes, no HTTP headers per message.

Built for connections that drop. Stream Management (XEP-0198, Chapter 10) is designed for exactly this. When Naserian rides out of coverage near Ngorongoro and back ten minutes later, her session **resumes** — messages that arrived while she was gone are delivered, and messages she sent that didn't quite make it are re-sent, with no full reconnect and re-authentication. This single extension is worth the whole protocol on a rural network.

Offline delivery is standard. If Ole's phone is off when Enkiama messages him, the server holds the message and delivers it when Ole reconnects. You don't build this.

One server, huge capacity. A single Ejabberd node has handled two million concurrent connections in production. MaasaiChat across two countries won't come close to stressing it — so it

runs on modest, affordable infrastructure.

You own it. Self-hosted, open source, no per-message fees, no vendor who can cut you off.

```
Rural constraint                 XMPP answer
-----------------------------------------------------------------
Expensive metered data     -->   Tiny stanzas, no polling
Connection drops constantly -->  Stream Management (resume, XEP-0198)
Phone often off/asleep     -->   Server-side offline storage + push
Low-end hardware           -->   Lightweight client, one connection
Must be self-run           -->   Open protocol, free Ejabberd
```

1.8 The mental model to carry forward

Before the next chapter, lock in this picture:

1. Each user has an **address** (JID): `naserian@maasaichat.com`.
2. Each device opens **one long-lived connection** to `maasaichat.com`.
3. Over it flow **stanzas** — small XML fragments.
4. There are exactly **three stanza types**: `message`, `presence`, `iq`.
5. New features are added as **extra XML inside stanzas**, defined by **XEPs**, never by changing the core.
6. The server (**Ejabberd**) handles routing, offline storage, presence, groups, and archive so you don't have to.

Everything from here builds on those six facts.

> “
>
> ## ? What you learned in this chapter
>
> - XMPP is an **open, standardized protocol** for exchanging small XML **stanzas** in real time over **one long-lived connection**.
> - It exists because **HTTP request/response can't push** server-initiated messages efficiently — fatal on metered, high-latency rural links.
> - The three stanza types are **`message`**, **`presence`**, **`iq`**; features are added via **XEPs** without changing the core.
> - **XMPP is the rulebook; Ejabberd is a builder that followed it** — like HTTP↔Nginx or SQL↔PostgreSQL. Gajim, Smack, and sendxmpp are other builders following the same rules, which is why they all interoperate.

> - Real, current users include **WhatsApp (built on Ejabberd), Nintendo Switch, Zoom, and Google Talk**.
> - The alternatives are **Matrix, MQTT, proprietary binary protocols, hosted SaaS, and raw WebSocket** — each with real trade-offs; XMPP uniquely wins on ownership + bandwidth + built-in features + cost for a self-hosted 2G community app.
> - XMPP fits **rural East Africa**: tiny messages, **Stream Management** for dropping connections, standard offline delivery, one high-capacity self-hosted server, and full ownership.
> - MaasaiChat runs its own **Ejabberd** on `maasaichat.com` — no dependence on outside messaging providers.

Ready for next chapter? (Chapter 2 — *The XMPP Vocabulary*: every key term you'll meet in this book — stanza, JID, resource, stream, namespace, roster, presence, MUC, MAM, SASL, and the rest — each defined plainly with a MaasaiChat example, so no word is ever a mystery in the chapters ahead.)

Chapter 2 — The XMPP Vocabulary

Key Terms & Definitions

XMPP has a lot of vocabulary, and the deep-dive chapters ahead assume you know it. So this chapter is a **dictionary you read once and refer back to forever.** Every term gets a plain definition, an analogy, and — where it helps — a MaasaiChat example. You'll meet each of these again in depth later; the goal here is that no word is ever a mystery.

The terms are grouped the way they actually relate, not alphabetically:

```
A. The absolute core        XMPP, stanza, stream, JID, resource
B. Addressing details       bare/full JID, message types
C. Connecting & login       TLS, SASL, bind, stream features
D. Staying connected        Stream Management, ping, keepalive
E. Messaging features       receipts, markers, chat states, IDs, carbons
F. Presence & contacts      presence, roster, subscription
G. Group chat               MUC, affiliation vs role, occupant
```

```
  H. Storage & history        offline messages, MAM, RSM, inbox
  I. Discovery & extensions   namespace, XEP, disco, caps, PubSub, PEP
  J. Media, calls, push, E2E  file upload, Jingle, push, OMEMO
  K. Federation & transports  s2s, dialback, BOSH, WebSocket
  L. Ejabberd-specific        mod_, ejabberdctl, Mnesia, vhost, ACL,
                              Erlang cookie, Erlang distribution
```

A one-page Quick Reference Card sits at the end of the chapter — tear it out (metaphorically) and keep it beside you.

A. The absolute core

XMPP

The rulebook. *Extensible Messaging and Presence Protocol*, an open IETF standard (RFC 6120/6121) created in 1999 as "Jabber." It defines how chat works: the shape of messages, the address format, how you log in, how features extend the core. It runs nothing by itself — a server like Ejabberd implements it. (See Chapter 1.)

Stanza

The basic unit of communication — one small XML fragment, like one sentence in a conversation. Everything you send or receive is a stanza. There are exactly three types: `<message>`, `<presence>`, `<iq>`.

```
<message to='ole@maasaichat.com' type='chat'>
  <body>The cattle arrived safely at the river.</body>
</message>
<!-- that whole thing = one stanza -->
```

Stream

The single long-lived connection between client and server. A tunnel that stays open; every stanza flows through it. Opening it is like starting a phone call; closing it is hanging up.

```
<!-- Naserian's phone opens the stream -->
<stream:stream to='maasaichat.com' xmlns='jabber:client' ...>
  ... all stanzas flow here for hours ...
```

```
</stream:stream>   <!-- closed when she leaves the app -->
```

JID (Jabber ID)

The address of every entity in XMPP — users, servers, group rooms. Like an email address, but for real-time chat. Format: `user@domain/resource`.

```
ole@maasaichat.com/android    a specific device
ole@maasaichat.com            the person (any device)
maasaichat.com                the server itself
warriors@conference.maasaichat.com   a group room
```

Resource

The device-identifier part of a JID, after the slash. It exists because one person logs in from several devices at once, and the server must tell them apart.

```
ole@maasaichat.com/phone    Ole's Tecno
ole@maasaichat.com/tablet   Ole's tablet
```

Send to the **bare** JID (`ole@maasaichat.com`) and Ejabberd chooses the best device; send to a **full** JID (`.../phone`) and it goes to that one device only.

B. Addressing details

Bare JID vs Full JID

- **Bare JID** — `ole@maasaichat.com`. Identifies the *person*. Used for offline messages, roster entries, MUC membership.
- **Full JID** — `ole@maasaichat.com/android`. Identifies a *specific connected device/session*. Used to reach exactly one device, e.g. during a call.

Rule of thumb: **person → bare, device → full.**

Message types

The `type` attribute on a `<message>` tells the server and client how to treat it:

```
chat        one-to-one conversation (Naserian → Ole)
groupchat   a MUC room message (to warriors@conference...)
normal      a single message, no ongoing chat (system notices)
headline    broadcast/alert, never stored offline (announcements)
error       something failed; carries an <error> child
```

Using the wrong type causes real bugs — e.g. a `headline` won't be saved for an offline user, so MaasaiChat uses `chat`/`groupchat` for anything that must survive a dropped connection.

C. Connecting & login

TLS / STARTTLS

Encryption of the stream. Before any password is sent, the client upgrades the plain TCP connection to an encrypted one (STARTTLS), or connects to an already-encrypted port (Direct TLS, 5223). No TLS = passwords and messages travel in the clear. MaasaiChat requires TLS always.

SASL

Simple Authentication and Security Layer — the login system, i.e. how you prove who you are. It supports several **mechanisms**:

```
PLAIN          sends the password directly (only safe inside TLS)
SCRAM-SHA-1    secure challenge/response, no password on the wire
SCRAM-SHA-256  stronger
SCRAM-SHA-512  strongest classic option (what Gajim used)
X-OAUTH2       log in with an OAuth token instead of a password
```

With SCRAM, Ole's phone never sends his password — it solves a cryptographic challenge that proves it knows the password. (See also §J, channel binding.)

Stream features

Right after the stream opens, the server sends a `<stream:features>` list — "here's what's available/required next": STARTTLS, which SASL mechanisms, resource binding, Stream Management, and so on. The client walks through them in order. It's the server announcing the login menu.

Bind (resource binding)

After SASL proves *who* you are, binding assigns *which session* — it hands you your full JID by attaching a resource.

```
<iq type='set'><bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
  <resource>android</resource>
</bind></iq>
<!-- server replies -->
<iq type='result'><bind>
  <jid>ole@maasaichat.com/android</jid>
</bind></iq>
```

Now Ole's full JID exists and messages can be routed to this exact session.

D. Staying connected (vital on 2G/3G)

Stream Management (SM) — XEP-0198

Makes delivery reliable on bad networks with an acknowledgement counter, and lets a dropped session **resume** instead of fully reconnecting.

```
<r/>          "please confirm how many of my stanzas you got"
<a h='32'/>   "confirmed — I have received up to stanza 32"
<resume/>     "I dropped and reconnected — resend from where we left off"
```

MaasaiChat example: Ole sends message #30 on 2G near the boma, the signal drops, he reconnects, the server sees he only acked #29, and re-sends #30. Zero loss. This is the single most important extension for rural networks.

Ping — XEP-0199

A tiny "are you still there?" `iq`. The client or server pings periodically to detect a silently-dead connection (common when a mobile network drops without closing the socket).

```
<iq type='get'><ping xmlns='urn:xmpp:ping'/></iq>
<iq type='result'/>   <!-- still alive -->
```

Whitespace keepalive

Even cheaper than a ping: the client sends a single space character down the stream now and then, just to keep NATs and carrier gateways from closing an "idle" connection. Common on mobile.

E. Messaging features

Delivery Receipt — XEP-0184

Proof a message *reached the recipient's device* — the single grey/blue tick "delivered."

```
<message to='naserian@maasaichat.com'>
  <body>Are the warriors ready?</body>
  <request xmlns='urn:xmpp:receipts'/>   <!-- please confirm delivery -->
</message>
```

Chat Markers — XEP-0333

Proof a message was *received and read* — the "read" tick. Distinct from a delivery receipt: delivery = it arrived on the device; marker (displayed) = the human actually saw it.

```
received     landed on the device
displayed    shown to the user (the "read" tick)
acknowledged app-level handled
```

Chat State Notifications — XEP-0085

The "typing…" experience. Tiny signals about what the other side is doing:

```
active     looking at the chat
composing  typing right now  ("Naserian is typing…")
paused     stopped typing but still there
inactive   tab/chat idle
gone       left the conversation
```

Stable & Unique Stanza IDs — XEP-0359

Gives every message two dependable IDs so all devices agree on identity: an **origin-id** set by the sender and a **stanza-id** assigned by the server/room. Essential for edits, reactions, replies, and de-duplication when the same message arrives via several paths.

Last Message Correction — XEP-0308

Editing a sent message. The new stanza points at the old one's ID with `<replace/>`, and clients swap the display in place (the "edited" label).

Message Retraction — XEP-0424

"Delete for everyone." A stanza that tells clients to remove a previously-sent message, referenced by its ID.

Message Reactions — XEP-0444

Emoji reactions attached to a message ID (□□ on Ole's cattle update), rather than a new separate message.

Message Carbons — XEP-0280

Multi-device sync for one-to-one chats. When Naserian messages Ole, a **copy** is delivered to *every* one of Ole's connected devices, and copies of what Ole *sends* also appear on his other devices — so phone and tablet stay identical (like WhatsApp Web mirroring your phone).

F. Presence & contacts

Presence

An announcement of availability, broadcast to those allowed to see it.

```
available    online (the default)
away         stepped away
xa           extended away (gone a while)
dnd          do not disturb
unavailable  offline
```

```
<presence><show>chat</show><status>Herding near Ngorongoro</status></presence>
```

When Ole closes the app, the server sends `unavailable` on his behalf, and his contacts see him go offline.

Presence priority

When Ole is online on several devices, each presence carries a numeric `<priority>`. Messages to his **bare** JID go to the highest-priority device. Negative priority means "never auto-deliver here."

Roster

Your contact list — stored on the **server**, synced to every device. Switch phones, log in, and all contacts reappear instantly. Each entry holds a JID, a display name, subscription state, and groups.

```
Ole's roster:
  naserian@maasaichat.com   (Warriors)
  enkiama@maasaichat.com    (Elders)
  nkeri@maasaichat.com      (Traders)
```

Presence Subscription

Permission to see someone's presence — like a mutual follow. You must ask and be approved.

```
subscribe     "I want to see your status"
subscribed    "granted — you may see mine"
unsubscribe   "stop seeing my status"
unsubscribed  "denied / revoked"
```

Ole sends `subscribe` to Naserian; she returns `subscribed`; now Ole sees when she's online.

Roster push

When your roster changes (you add Nkeri), the server *pushes* the update to all your logged-in devices automatically, so they stay in sync without asking.

G. Group chat

MUC (Multi-User Chat) — XEP-0045

Group chat rooms. Each room is itself a JID on the conference service:

```
warriors@conference.maasaichat.com
  ^room            ^MUC service domain
```

Send with `type='groupchat'` and every occupant receives it.

Occupant & nickname

Inside a room you appear under a **nickname**, and your in-room address is `room@service/nickname` (a full JID whose "resource" is your nick). Your real JID may be hidden depending on room settings.

Affiliation vs Role (the classic MUC confusion)

Two *different* permission systems in every MUC:

```
AFFILIATION  long-term membership status (persists across visits)
  owner    created the room, full control
  admin    can manage members/admins
  member   allowed into a members-only room
  outcast  banned
  none     no special standing

ROLE         what you can do RIGHT NOW, this visit
  moderator     can kick, grant voice, moderate
  participant   can speak
  visitor       can only read (no voice)
  none          not in the room
```

Affiliation is *who you are to the room over time*; role is *what you may do in this session*. Enkiama the chief might be an **owner** (affiliation) who is currently acting as **moderator** (role).

Message Moderation — XEP-0425

Lets a room moderator retract/hide someone else's message in the room (spam control), distinct from a user deleting their own.

H. Storage & history

Offline Messages — XEP-0160

If the recipient is offline, the server stores the message and delivers it on reconnect — voicemail for chat. Enkiama is out of signal in the bush; Ole's message waits on the server and lands when Enkiama's phone finds a tower.

MAM (Message Archive Management) — XEP-0313

The server keeps a searchable **archive** of conversations; clients fetch history on demand. Buy a new phone, log in, ask for "the last 7 days," and your full history appears. Without MAM, a new device starts empty.

RSM (Result Set Management) — XEP-0059

Paging for large result sets — "give me 20 messages before this point," then the next 20. MAM uses RSM so a phone on 2G loads history in small chunks instead of one huge download.

Inbox — XEP-0430

A server-built list of your conversations with the latest message and unread count for each — the "chat list" screen, computed server-side so it's instant and consistent across devices.

IQ (Info/Query)

The request/response stanza — XMPP's version of an HTTP GET/POST. Always in pairs, always with a matching `id`:

```
get     ask for information        result   success (may carry data)
set     do or change something     error    it failed
```

```
<iq type='get' id='r1'><query xmlns='jabber:iq:roster'/></iq>
<iq type='result' id='r1'><query> ...contacts... </query></iq>
```

Roster fetch, ping, disco, bind, MAM queries — all are IQs.

I. Discovery & extensions

Namespace (xmlns)

A unique string that says *which extension an element belongs to*. Without it, `<request>` is meaningless; with `xmlns='urn:xmpp:receipts'` everyone knows it's a delivery receipt. Think of it as the department stamp on a memo.

```
See  xmlns="urn:xmpp:something"  →  it's a XEP.
Search that string  →  you find the exact XEP instantly.
```

URN

*Uniform Resource **Name*** — the format most XMPP namespaces use. A permanent identifier that names *what* something is, as opposed to a URL, which locates *where* something is. `urn:xmpp:receipts` is never fetched from a server; it's simply an agreed string meaning "delivery receipts." Trailing digits are versions (`urn:xmpp:sm:3`), and `:0` marks an experimental XEP. (Full treatment in §3.5.)

XEP

XMPP Extension Protocol — a document that adds a feature on top of core XMPP, each with a number: XEP-0045 (group chat), XEP-0184 (receipts), XEP-0166 (calls). Base XMPP is a basic phone; XEPs bolt on the camera, caller-ID, and voicemail. (Full treatment in Chapter 6.)

Service Discovery (disco) — XEP-0030

How one entity asks another "what are you, and what can you do?" A client discos the server to learn which features and services (MUC, file upload, push) exist.

```
<iq type='get' to='maasaichat.com'>
  <query xmlns='http://jabber.org/protocol/disco#info'/>
</iq>
```

Entity Capabilities (caps) — XEP-0115

An optimization on top of disco: each client advertises a short hash of its feature set in presence, so others cache "a client with hash X supports these XEPs" instead of re-asking every time. Saves bandwidth — good on 2G.

PubSub (Publish-Subscribe) — XEP-0060

A general publish/subscribe system on the server: publishers post items to a **node**, subscribers get them. The backbone for many features (avatars, bookmarks, presence-like data).

PEP (Personal Eventing Protocol) — XEP-0163

A simplified PubSub attached to a user's own account — "my nodes." Used for things like your current avatar, mood, or bookmarks, auto-broadcast to contacts who care.

J. Media, calls, push, encryption

HTTP File Upload — XEP-0363

How you send photos/files. The client asks the server for an upload **slot** (a PUT URL + a GET URL), uploads the file over HTTPS, then sends the GET URL in a normal message. This is the file-transfer method MaasaiChat actually uses — it works fine over flaky mobile links, unlike peer-to-peer transfer.

Jingle — XEP-0166 (+0167/0176/0215/0320)

The signaling framework for voice/video calls. Jingle negotiates the call (who, what codec, network path) *inside XMPP*, while the actual audio/video flows over WebRTC. Related XEPs handle RTP media (0167), ICE network traversal (0176), TURN/STUN discovery (0215), and encryption (0320).

Push Notifications — XEP-0357

Wakes a sleeping phone. When a message arrives for an app that's backgrounded/disconnected, the server triggers a push (via FCM on Android) so the user gets notified without holding a socket open — essential for battery on low-end phones.

OMEMO — XEP-0384

Modern end-to-end encryption for XMPP (built on the Signal Protocol's ideas). Messages are encrypted per-device so that not even the server can read them. Optional; a design decision for later MaasaiChat phases.

SASL Channel Binding — XEP-0440 / Extensible SASL — XEP-0388

Hardening for login: channel binding ties the SASL authentication to the exact TLS connection, blocking a class of man-in-the-middle attacks. XEP-0388 modernizes how SASL mechanisms are negotiated.

K. Federation & transports

c2s and s2s

Two connection kinds Ejabberd listens for: **c2s** (client-to-server, port 5222 — phones connecting) and **s2s** (server-to-server, port 5269 — other XMPP servers connecting for federation).

Federation

Different XMPP servers talking to each other, so `naserian@maasaichat.com` could message `someone@another-server.org` — the same way email crosses providers. Enabled by s2s.

Server Dialback — XEP-0220

A verification handshake that lets a receiving server confirm a connecting server really owns the domain it claims, preventing spoofed federation. (Bidirectional s2s, XEP-0288, lets one connection carry traffic both ways.)

BOSH and WebSocket

Alternative transports for when a raw TCP stream isn't possible (e.g. a browser). **BOSH** tunnels XMPP over HTTP long-polling; **WebSocket** carries the XML stream over a WebSocket. Native mobile apps use raw TCP; web clients use WebSocket.

L. Ejabberd-specific terms

mod_ (module)

A plugin that adds a feature to Ejabberd, switched on in `ejabberd.yml` — like browser extensions.

```
mod_muc        group chat           mod_mam        message archive
mod_offline    offline storage      mod_http_api   REST API
```

```
mod_ping       keepalive pings       mod_mqtt       MQTT protocol
mod_push       push notifications    mod_admin_extra extra CLI commands
```

No `mod_muc` → no group chat. Add it → group chat works.

ejabberdctl

Ejabberd's command-line control tool, run inside the container — the `psql`/`redis-cli` equivalent for Ejabberd.

```
docker exec ejabberd ejabberdctl status
docker exec ejabberd ejabberdctl registered_users maasaichat.com
docker exec ejabberd ejabberdctl send_message chat ole@maasaichat.com \
  naserian@maasaichat.com "" "Meeting at the manyatta tonight"
```

vhost (virtual host)

One Ejabberd process can serve several domains at once (`maasaichat.com`, `staging.maasaichat.com`), each with its own users and settings — like Nginx server blocks.

ACL & Access Rules

Ejabberd's permission system: an **ACL** defines *who* (e.g. "admins = admin@maasaichat.com"), and an **access rule** grants those groups rights (e.g. who may use the REST API). Misconfigured ACLs are the classic cause of `Account does not have the right to perform the operation` when calling the API.

Mnesia

Erlang's built-in database, bundled with Ejabberd, used for its *internal* live state — who's connected where, room state, roster, short-term offline spool. It does **not** hold MaasaiChat's business data; that lives in PostgreSQL.

Erlang Cookie

A shared secret string all nodes in an Ejabberd cluster must have identical — the "password" that proves two nodes belong to the same cluster. Match → they trust each other; mismatch → the node is rejected.

Erlang Distribution (dist)

Erlang's native node-to-node networking. In a cluster, if Ole is on Node 1 and Naserian is on Node 2, Node 1 hands the stanza straight to Node 2 over Erlang distribution — no Redis pub/sub needed. This is why an Ejabberd cluster is simpler than a hand-built socket cluster.

Spool

The queue where offline messages wait for a disconnected user. When Enkiama reconnects, Ejabberd flushes his spool to his device.

Quick Reference Card

Term	Simple definition
XMPP	The language/rules of chat
Stanza	One unit of communication (message/presence/iq)
Stream	The single open connection (tunnel)
JID	Address: user@domain/resource
Resource	Device identifier (/phone /tablet)
Bare / Full JID	Person (bare) vs specific device/session (full)
Message type	chat / groupchat / normal / headline / error
TLS / STARTTLS	Encryption of the stream
SASL	Authentication system (SCRAM, PLAIN, OAuth)
Stream features	Server's "menu" of what to negotiate next
Bind	Claiming your resource → full JID
Stream Management	Reliable delivery + resume on bad networks (0198)
Ping	"Are you alive?" heartbeat (0199)
Keepalive	Whitespace to stop NAT/carrier timeouts
Delivery Receipt	"Delivered" tick (0184)
Chat Markers	"Read/displayed" tick (0333)
Chat States	typing / paused indicators (0085)
Stanza IDs	Stable message identity (0359)
Correction	Edit a sent message (0308)
Retraction	Delete for everyone (0424)
Reactions	Emoji on a message (0444)
Carbons	Multi-device 1:1 sync (0280)
Presence	Online/offline status
Priority	Which device gets bare-JID messages
Roster	Contact list (stored on server)

Subscription	Permission to see presence (mutual)
MUC	Group chat room (0045)
Occupant / nick	Your identity inside a room
Affiliation	Long-term membership (owner/admin/member/outcast)
Role	Current-session powers (moderator/participant/visitor)
Offline Messages	Stored when recipient offline (0160)
MAM	Server message archive/history (0313)
RSM	Paging for large results (0059)
Inbox	Server-built conversation list (0430)
IQ	Request/response stanza (get/set/result/error)
Namespace (xmlns)	Which XEP an element belongs to
URN	Permanent NAME (urn:xmpp:...), not a fetchable URL
XEP	A document adding a feature to XMPP
disco	Service discovery: "what can you do?" (0030)
caps	Cached capability hashes (0115)
PubSub / PEP	Publish-subscribe / personal eventing (0060/0163)
HTTP File Upload	Send files via upload slot (0363)
Jingle	Voice/video call signaling (0166)
Push	Wake a sleeping phone via FCM (0357)
OMEMO	End-to-end encryption (0384)
c2s / s2s	Client-to-server / server-to-server
Federation	Cross-server messaging (like email)
Dialback	Verifies a federating server (0220)
BOSH / WebSocket	XMPP over HTTP / over WebSocket (for browsers)
mod_	Ejabberd plugin/module
ejabberdctl	Ejabberd CLI management tool
vhost	One server hosting multiple domains
ACL / Access Rule	Ejabberd permission system
Mnesia	Ejabberd's internal live-state database
Erlang Cookie	Cluster membership password
Erlang dist	Direct node-to-node communication
Spool	Queue of waiting offline messages

> “
>
> ## ? What you learned in this chapter
>
> - **Stanza, stream, JID, resource** are the four words the whole protocol stands on — one XML unit, one open connection, one address, one

device tag.
- Addressing splits into **bare (person)** vs **full (device)** JIDs, and every message carries a **type** (chat/groupchat/normal/headline/error) that decides how it's handled.
- Login is a pipeline: **TLS → SASL → bind → stream features**, and staying connected on 2G relies on **Stream Management, ping, and keepalives**.
- Messaging polish is a stack of small XEPs — **receipts (0184), markers (0333), chat states (0085), stable IDs (0359), carbons (0280)** — plus edit/retract/react.
- **Presence + roster + subscription** run contacts and status; **MUC** runs groups, where **affiliation (long-term) and role (this session) are different things**.
- History is **offline messages → MAM → RSM → inbox**; discovery is **disco + caps**; extensions are identified by **namespace** and documented as **XEPs**.
- Ejabberd-specific: **mod_ modules, ejabberdctl, vhosts, ACLs, Mnesia, the Erlang cookie, and Erlang distribution** — the operational vocabulary for Part 3.

Ready for next chapter? (Chapter 3 — *XML Basics — Streams, Stanzas & Nonzas*: now that you know the words, we read the XML itself — the three layers, the container-vs-contents distinction, one complete annotated chat session from TCP handshake to stream close, what order is fixed and what is free, and the tools to type raw stanzas yourself.)

Chapter 3 — XML Basics: Streams, Stanzas & Nonzas

You know the vocabulary. Now we read the actual XML — slowly, once, all the way through a real session. By the end of this chapter you will be able to look at any raw XMPP exchange and know exactly what layer you're in, what's happening, and what should come next.

3.1 Is `<stream:stream>` a stanza?

No. This trips up almost everyone, so let's settle it immediately.

```
stream:stream  =  the CONTAINER
stanzas        =  what goes INSIDE the container

Like:
  stream  = the envelope
  stanzas = the letters inside it

Open the envelope once   (stream opens)
Put many letters inside  (stanzas, repeated)
Close it when done       (stream closes)
```

The stream is opened once per session and closed once per session. Stanzas flow inside it, many thousands of times.

Notice something odd about the stream tag: it is a single XML document whose root element **stays open for hours**. XMPP is not "send a document, get a document" — the entire session *is* one enormous XML document, streamed a piece at a time, and each stanza is a child element of its root. That's the trick that makes XMPP real-time.

3.2 The three layers

Always know which layer you're looking at:

```
┌──────────────────────────────────────────────┐
│ LAYER 3 — STANZAS (repeat forever)           │
│   <message>  <presence>  <iq>                │
├──────────────────────────────────────────────┤
│ LAYER 2 — XMPP STREAM (once per session)     │
│   <stream:stream> ......... </stream:stream> │
├──────────────────────────────────────────────┤
│ LAYER 1 — TCP + TLS (the wire)               │
│   raw bytes, port 5222, encrypted            │
└──────────────────────────────────────────────┘
```

Layer 1 is plumbing. Layer 2 is the envelope. Layer 3 is the conversation.

3.3 The third thing: nonzas

Here's a distinction that will make the rest of this book click. Inside the stream, **not everything is a stanza.**

Only three elements are stanzas: `<message>`, `<presence>`, `<iq>`. Everything else that flows inside the stream — `<stream:features>`, `<starttls/>`, `<auth/>`, `<challenge/>`, `<success/>`, `<enable/>`, `<r/>`, `<a/>` — is officially called a **nonza** (literally: "not a stanza").

```
    STANZAS                    NONZAS
    -----------------------    -----------------------------------
    <message>                  <stream:features>   <starttls/>
    <presence>                 <auth/> <challenge/> <response/>
    <iq>                       <success/> <failure/>
                               <enable/> <enabled/> <r/> <a/>


    Routable — they have       Not routable — they are between
    'to' and 'from', the       THIS client and THIS server only.
    server delivers them       Pure connection machinery.
    anywhere on the network.
```

The practical rule: **stanzas travel; nonzas negotiate.** A message can cross the planet to another server. An `<auth/>` never goes anywhere — it's a private word between your phone and Ejabberd.

This is why the SASL exchange and Stream Management acks look "different" from chat traffic. They're not stanzas at all.

3.4 Reading XMPP's XML: the four things to look for

You don't need to be an XML expert. You need four things:

1. Element — the tag name. It tells you the *kind* of thing: `<message>`, `<iq>`, `<body>`.

2. Attributes — the key facts on the tag itself: who, to whom, what kind, which id.

```
<message from='ole@maasaichat.com/android'
         to='naserian@maasaichat.com'
         type='chat'
         id='msg-001'>
```

3. Children — nested elements carrying the payload: `<body>`, `<show>`, `<query>`.

4. `xmlns` **(the namespace)** — *the most important attribute in XMPP.* It tells you which extension a child element belongs to. `<request/>` alone is meaningless; `<request xmlns='urn:xmpp:receipts'/>` is unambiguously a delivery-receipt request.

```
See an xmlns you don't recognize?
  → search that exact string
  → you land on the XEP that defines it
That single habit makes XMPP self-documenting.
```

3.5 Anatomy of a namespace: what `urn:` actually means

You just met `xmlns='urn:xmpp:receipts'`. Two questions always follow: *what is a URN*, and *why do some namespaces look like web addresses instead?*

URN = Uniform Resource **Name**

```
Uniform   consistent, standard format
Resource  anything that can be named
Name      a permanent identifier — NOT a location
```

The contrast with a URL is the whole point:

```
URL = Uniform Resource LOCATOR      URN = Uniform Resource NAME
Tells you WHERE something is        Tells you WHAT something is
Breaks if the server moves          Never changes, never breaks
Must be fetched to be useful        Nothing to fetch — it's just a name

https://xmpp.org/extensions/        urn:xmpp:receipts
  xep-0184.html                       permanent, forever
  (can 404 tomorrow)
```

Like a person versus their address: "*Ole Saitoti*" is a name that never changes; "*the manyatta by the river*" is a location that can. A URN is the name.

The format

```
urn : namespace : specific-name : version
 ↑        ↑            ↑             ↑
always   who owns   the thing      optional, and
"urn"    this space  being named   very common in XMPP

urn : xmpp       : receipts          delivery receipts
urn : xmpp       : sm : 3            stream management, version 3
urn : xmpp       : sid : 0           stable stanza IDs, version 0
urn : maasaichat : cattle : 1        OUR extension, version 1
```

That trailing number matters. `urn:xmpp:sm:3` is not decoration — it's version 3 of Stream Management, and a client that speaks `sm:2` cannot assume it understands `sm:3`. When you see `:0` (as in `urn:xmpp:sid:0`), the XEP is still experimental and the namespace *is expected to change* when it stabilizes. That single digit tells you how much to trust the feature.

Why XMPP chose URNs

XMPP needed identifiers that could never break or collide across two decades and thousands of implementations.

```
If namespaces were URLs:           Because they're URNs:
  xmpp.org restructures → broken     nothing to break
  server down → ambiguity            no server involved
  someone hijacks the domain         nobody can hijack a string
```

A namespace is not a resource to fetch. **Nothing ever requests `urn:xmpp:receipts` over the network.** It's a shared agreement: *when you see this exact string, you know it means delivery receipts.* It lives in developers' heads and in the `if` statements of every XMPP library on earth. That's all it needs to do.

So why do some namespaces look like URLs?

Because XMPP is 25 years old and has three generations of naming, all still in use:

```
Style                                Era        Example
-----------------------------------  --------   ----------------------------
jabber:xxx                           1999-      jabber:client
  the original short names                      jabber:iq:roster

http://jabber.org/protocol/xxx       ~2000s     http://jabber.org/protocol/
  older XEPs, URL-SHAPED but still                 chatstates
```

```
    just a name — never fetched!                  http://jabber.org/protocol/
                                                    disco#info

  urn:xmpp:xxx                          modern    urn:xmpp:receipts
    what all new XEPs use                         urn:xmpp:sm:3

  urn:ietf:params:xml:ns:xxx            IETF core urn:ietf:params:xml:ns:
    the base protocol itself (RFC 6120)             xmpp-sasl / xmpp-bind
```

The crucial point: `http://jabber.org/protocol/chatstates` **is not a URL that gets loaded.** It looks like one, but it is used exactly the same way as a URN — as an opaque, permanent identifier string. Nobody's phone has ever made an HTTP request to it. Early XMPP borrowed the URL *shape* (a common XML convention for guaranteeing uniqueness via a domain you own), then the community moved to `urn:xmpp:*` because the URL shape misled people into thinking it meant something fetchable.

So when a stanza mixes both styles — and real ones constantly do — nothing is inconsistent. You're just seeing XEPs from different decades:

```
<message to='naserian@maasaichat.com' type='chat' id='msg-001'>
  <body>The cattle arrived safely.</body>
  <active    xmlns='http://jabber.org/protocol/chatstates'/>  <!-- 2000s XEP -->
  <origin-id xmlns='urn:xmpp:sid:0' id='msg-001'/>            <!-- modern XEP -->
  <request   xmlns='urn:xmpp:receipts'/>                      <!-- modern XEP -->
</message>
```

Naming your own extensions

When MaasaiChat needs something the standard doesn't cover — say, attaching a cattle-market listing to a message — you mint your own namespace. Follow the URN convention and version it from day one:

```
  Good:  urn:maasaichat:cattle:1        a name, permanent, versioned
  Bad:   https://maasaichat.com/xmpp/cattle
           → looks fetchable, breaks the day the domain changes
```

```
<message to='cattle-market@conference.maasaichat.com' type='groupchat'>
  <body>12 head of cattle, Ngorongoro, ready Friday</body>
  <listing xmlns='urn:maasaichat:cattle:1' count='12' ready='2026-07-17'/>
</message>
```

Any client that doesn't know `urn:maasaichat:cattle:1` simply ignores the `<listing/>` child and still shows the `<body>` text — which is exactly why XMPP extends so gracefully. And that version digit means when the listing format changes, you bump to `:2` and old clients keep working instead of misreading new data.

> **Catch it in one line:** *A namespace is a permanent name, never a place — even the ones shaped like URLs are never fetched.*

3.6 One complete 1:1 chat session, annotated

This is the whole thing — Ole opens MaasaiChat on his phone, chats with Naserian, and closes the app. Every byte in order. Read it once now; it will make sense in pieces as the book continues.

```
<!-- ============================================================ -->
<!-- LAYER 1: TCP connection to maasaichat.com:5222               -->
<!-- (no XML yet — just the TCP handshake)                        -->
<!-- ============================================================ -->

<!-- ============================================================ -->
<!-- LAYER 2: STREAM OPENS                                        -->
<!-- ============================================================ -->

<!-- Ole's phone opens the stream -->
<?xml version='1.0'?>
<stream:stream
    to='maasaichat.com'
    version='1.0'
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'>

<!-- Server opens ITS stream back (two streams: one each direction) -->
<?xml version='1.0'?>
<stream:stream
    from='maasaichat.com'
    id='session-abc-123'
    version='1.0'
```

```
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'>

<!-- ---- STEP 1: SERVER OFFERS FEATURES (nonza) ---- -->
<stream:features>
  <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'>
    <required/>             <!-- MaasaiChat demands encryption -->
  </starttls>
</stream:features>

<!-- ---- STEP 2: ENCRYPT FIRST (nonzas) ---- -->
<starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
<proceed  xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
<!-- TLS handshake happens now. Everything below is encrypted.
     The stream RESTARTS from scratch after TLS. -->

<?xml version='1.0'?>
<stream:stream to='maasaichat.com' version='1.0'
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'>

<!-- Now the server offers login options -->
<stream:features>
  <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    <mechanism>SCRAM-SHA-512</mechanism>
    <mechanism>SCRAM-SHA-256</mechanism>
    <mechanism>PLAIN</mechanism>
  </mechanisms>
</stream:features>

<!-- ---- STEP 3: AUTHENTICATION (nonzas — NOT iq!) ---- -->
<auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
      mechanism='SCRAM-SHA-512'>BASE64_INITIAL</auth>

<challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>BASE64_CHALLENGE</challenge>

<response xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>BASE64_RESPONSE</response>

<success xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>BASE64_VERIFY</success>
<!-- Ole's password never crossed the wire — only proofs of knowing it -->
```

```
<!-- ---- STEP 4: STREAM REOPENS AFTER AUTH (required by spec) ---- -->
<?xml version='1.0'?>
<stream:stream to='maasaichat.com' version='1.0'
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'>

<!-- Post-auth features -->
<stream:features>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
  <sm   xmlns='urn:xmpp:sm:3'/>
</stream:features>

<!-- ============================================================ -->
<!-- LAYER 3: STANZAS BEGIN                                        -->
<!-- ============================================================ -->

<!-- ---- STEP 5: RESOURCE BINDING (a real iq stanza) ---- -->
<iq type='set' id='bind-001'>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <resource>android</resource>
  </bind>
</iq>

<iq type='result' id='bind-001'>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <jid>ole@maasaichat.com/android</jid>     <!-- full JID exists now -->
  </bind>
</iq>

<!-- ---- STEP 6: ENABLE STREAM MANAGEMENT (nonzas) ---- -->
<enable  xmlns='urn:xmpp:sm:3' resume='true'/>
<enabled xmlns='urn:xmpp:sm:3' id='sm-session-xyz' resume='true'/>
<!-- Now Ole can survive the 2G drops near Ngorongoro -->

<!-- ---- STEP 7: FETCH ROSTER (iq) ---- -->
<iq type='get' id='roster-001'>
  <query xmlns='jabber:iq:roster'/>
</iq>
```

```
<iq type='result' id='roster-001'>
  <query xmlns='jabber:iq:roster'>
    <item jid='naserian@maasaichat.com' name='Naserian' subscription='both'/>
    <item jid='enkiama@maasaichat.com'  name='Enkiama'  subscription='both'/>
    <item jid='nkeri@maasaichat.com'    name='Nkeri'    subscription='both'/>
  </query>
</iq>

<!-- ---- STEP 8: GO ONLINE (presence) ---- -->
<!-- NOTE: no type attribute = available. There is no <show>available</show> -->
<presence>
  <c xmlns='http://jabber.org/protocol/caps'
     hash='sha-1' node='https://maasaichat.com' ver='q07IKJEyjvHSyhy//CH0CxmKi8w='/>
  <!-- caps: "here's a hash of which XEPs I support" — saves re-asking -->
</presence>

<!-- ---- STEP 9: CONTACTS' PRESENCE ARRIVES ---- -->
<presence from='naserian@maasaichat.com/phone'>
  <show>chat</show>
  <status>Herding near Ngorongoro</status>
</presence>

<!-- ---- STEP 10: OLE SENDS THE MESSAGE ---- -->
<message from='ole@maasaichat.com/android'
         to='naserian@maasaichat.com'
         type='chat'
         id='msg-001'>
  <body>The cattle arrived safely at the river.</body>
  <origin-id xmlns='urn:xmpp:sid:0' id='msg-001'/>   <!-- XEP-0359 -->
  <request   xmlns='urn:xmpp:receipts'/>             <!-- XEP-0184: confirm delivery -->
  <markable  xmlns='urn:xmpp:chat-markers:0'/>       <!-- XEP-0333: allow read tick -->
</message>

<!-- Ole's client asks "did you get everything?" (nonza) -->
<r xmlns='urn:xmpp:sm:3'/>
<a xmlns='urn:xmpp:sm:3' h='5'/>    <!-- server: "I have 5 stanzas from you" -->

<!-- ---- STEP 11: DELIVERY RECEIPT (Naserian's app, automatic) ---- -->
<message from='naserian@maasaichat.com/phone' to='ole@maasaichat.com'>
  <received xmlns='urn:xmpp:receipts' id='msg-001'/>
```

```
</message>
<!-- Ole's UI: ✓ becomes ✓✓ -->

<!-- ---- STEP 12: NASERIAN IS TYPING ---- -->
<message from='naserian@maasaichat.com/phone'
         to='ole@maasaichat.com' type='chat'>
  <composing xmlns='http://jabber.org/protocol/chatstates'/>
  <!-- no <body> — this is a pure signal, costs ~100 bytes -->
</message>

<!-- ---- STEP 13: NASERIAN REPLIES ---- -->
<message from='naserian@maasaichat.com/phone'
         to='ole@maasaichat.com' type='chat' id='msg-002'>
  <body>Great news! How many?</body>
  <active    xmlns='http://jabber.org/protocol/chatstates'/>
  <origin-id xmlns='urn:xmpp:sid:0' id='msg-002'/>
  <request   xmlns='urn:xmpp:receipts'/>
  <markable  xmlns='urn:xmpp:chat-markers:0'/>
</message>

<!-- ---- STEP 14: OLE READS IT ---- -->
<message from='ole@maasaichat.com/android'
         to='naserian@maasaichat.com' type='chat'>
  <displayed xmlns='urn:xmpp:chat-markers:0' id='msg-002'/>
</message>
<!-- Naserian's UI: ✓✓ becomes the blue "read" tick -->

<!-- ---- STEP 15: KEEPALIVE (iq ping) ---- -->
<iq from='ole@maasaichat.com/android' to='maasaichat.com'
    type='get' id='ping-001'>
  <ping xmlns='urn:xmpp:ping'/>
</iq>
<iq from='maasaichat.com' to='ole@maasaichat.com/android'
    type='result' id='ping-001'/>

<!-- ---- STEP 16: OLE CLOSES THE APP ---- -->
<presence type='unavailable'/>

<!-- ---- STREAM CLOSES ---- -->
</stream:stream>
```

```
<!-- TCP connection closes -->
```

That's a complete XMPP session. Every chapter in Part 2 is a deep-dive into one line of what you just read.

Two corrections worth burning in

Two mistakes appear constantly in blog posts and even in some tutorials:

1. `<show>available</show>` **does not exist.** The legal `<show>` values are only `away`, `chat`, `dnd`, `xa`. "Available" is the *default* — signalled by a `<presence/>` with **no** `type` attribute at all. Offline is `<presence type='unavailable'/>`.
2. **SASL auth and Stream Management are not `iq`.** `<auth/>`, `<success/>`, `<enable/>`, `<r/>`, `<a/>` are **nonzas**. If you go looking for an `<iq>` wrapper around them you'll be confused forever.

3.7 What order is fixed, and what is free

```
FIXED — the setup handshake, always this order:
  1. TCP connect
  2. Stream open
  3. Features → STARTTLS → (stream restarts)
  4. Features → SASL auth → (stream restarts)
  5. Resource bind          → full JID assigned
  6. (optional) Enable Stream Management

FREE — once bound, anything, any time, both directions:
  Presence, roster fetch, messages, iq requests, pings...
```

Convention (not law) after binding: enable SM → fetch roster → send initial presence. Clients do it in that order because you want reliability on before traffic, and contacts loaded before you announce yourself.

After that, XMPP is **fully asynchronous and bidirectional**. Ole can send three messages without waiting for any reply; the server can push Naserian's presence in the middle of them. The one hard rule is that every `iq` you send with an `id` will get exactly one `result` or `error` back carrying that same `id` — that's how you match responses to requests on a pipe where anything can arrive at any moment.

```
   Stream setup   = a strict staircase, step by step
   After binding  = a busy two-way street
```

3.8 Tools — type stanzas yourself

Reading XML teaches you some of it. *Typing* raw stanzas at a live server and watching it answer teaches you all of it. Four tools, easiest first:

1. Gajim's XML Console (start here)

The best tool, and you likely already have it. `Gajim → Accounts → Advanced → XML Console`. It shows every stanza in and out **and** lets you type raw XML and send it.

Try this first — a ping:

```
<iq type='get' id='test-001' to='maasaichat.com'>
  <ping xmlns='urn:xmpp:ping'/>
</iq>
```

Watch `<iq type='result' id='test-001'/>` come back. You just spoke XMPP by hand.

2. Psi

```
sudo apt install psi
```

Another desktop client with a cleaner, more technical XML console. Some people prefer it purely for learning.

3. websocat — raw stream from the terminal

Talk to Ejabberd's WebSocket endpoint with nothing in between:

```
websocat --protocol xmpp ws://localhost:5280/ws
```

Then paste the stream opener and watch the server's raw reply:

```
<open xmlns='urn:ietf:params:xml:ns:xmpp-framing'
      to='maasaichat.com' version='1.0'/>
```

(The WebSocket binding uses `<open/>` instead of `<stream:stream>` — same idea, framed differently. Note the `--protocol xmpp` flag; Ejabberd requires that subprotocol header.)

4. A throwaway Python script

For scripted sequences, nothing beats a socket:

```
import socket, ssl

sock = socket.create_connection(("localhost", 5222))
sock.send(b"""<?xml version='1.0'?>
<stream:stream to='maasaichat.com' version='1.0'
  xmlns='jabber:client' xmlns:stream='http://etherx.jabber.org/streams'>""")
print(sock.recv(4096).decode())     # server's stream + features
```

You'll get the stream header and `<stream:features>` back — proof you're speaking the protocol with 6 lines of code. (To go further than STARTTLS you'd wrap the socket with `ssl`; easier to let a library handle it — see Part 4.)

5. Server-side: send a stanza as the admin

Ejabberd can inject a stanza for you from the CLI — useful for testing what a client *receives*:

```
docker exec ejabberd ejabberdctl send_stanza \
  'maasaichat.com' 'ole@maasaichat.com' \
  '<message type="chat"><body>Test from the server</body></message>'
```

```
   Best for visual learning:  Gajim XML console
   Best for seeing the truth: websocat / raw socket
   Best for automation:       ejabberdctl send_stanza
```

3.9 A practice loop that actually works

Four short sessions will give you real fluency:

```
   1. ECHO      Open Gajim's console. Just watch. Send a message
                in the UI and read the XML it produced.
   2. HAND-SEND Type a ping by hand. Then a message. Then a
                <composing/> chat state. Watch the other client react.
```

```
3. FROM SCRATCH  Pick a goal ("mark msg-002 as read"), write the
                 stanza with no reference, send it, see if it works.
4. DECODE        Grab any unknown stanza from the console and answer:
                 which type? which xmlns/XEP? what does it do?
                 what response should come back?
```

Step 4 is the one that matters. When you can look at a stanza, spot the `xmlns`, and say "that's XEP-0333, it's a read marker, no response expected" — you've stopped memorizing and started *reading* XMPP.

> “
>
> ## ? What you learned in this chapter
>
> - `<stream:stream>` **is not a stanza** — it's the container. The whole session is one long XML document whose root stays open for hours; stanzas are its children.
> - There are **three layers**: TCP+TLS (wire) → stream (envelope) → stanzas (conversation).
> - Only `<message>`, `<presence>`, `<iq>` are stanzas. Everything else inside the stream — features, STARTTLS, SASL, `<enable/>`, `<r/>`, `<a/>` — is a **nonza**. **Stanzas travel; nonzas negotiate.**
> - Reading XMPP means reading four things: **element, attributes, children, and** `xmlns` — and the `xmlns` is the pointer to the XEP that explains everything else.
> - A namespace is a **URN — a permanent *name*, never a *location***. Nothing is ever fetched from it; it's a shared agreement. The trailing digit is a **version** (`urn:xmpp:sm:3`), and `:0` means the XEP is still experimental.
> - Namespaces come in **three generations** — `jabber:*`, `http://jabber.org/protocol/*`, and modern `urn:xmpp:*` — all used the same way. **The URL-shaped ones are still never fetched.**
> - You walked one **complete session**, TCP handshake → STARTTLS → SASL → bind → SM → roster → presence → message → receipt → typing → read marker → ping → unavailable → close.
> - **Setup order is fixed** (connect → TLS → auth → bind); **everything after binding is free-form and asynchronous** — with the one rule that every `iq` gets exactly one matching `result`/`error` by `id`.
> - Two myths corrected: `<show>available</show>` **isn't a thing** (available = presence with no type), and **SASL/SM are not** `iq`.
> - Tools to speak XMPP by hand: **Gajim's XML console**, Psi, **websocat** (`--protocol xmpp`), a raw Python socket, and `ejabberdctl send_stanza`.

Ready for next chapter? (Chapter 4 — *The JID Address System*: bare vs full JIDs, how Ejabberd decides which of Ole's devices gets a message, priorities, resource conflicts, and the addressing rules that quietly break apps that get them wrong.)

Revision #3
Created 14 July 2026 17:13:30 by Admin Qbit
Updated 15 July 2026 15:38:41 by Admin Qbit