EFMS Kafka Integration Architecture Documentation

1)Overview

This document describes how EFMS uses event streaming and real-time messaging to support background workflows and immediate user updates.

- Kafka is used as a reliable event channel between EFMS components. Instead of doing slow work inside API requests (such as sending email), EFMS publishes an event and processes it in the background.
- SignalR is used to send immediate updates to the user interface (example: forcing a user logout across all active sessions).

This design supports:

- Faster API responses (work is handled asynchronously)
- Reliable processing (events can be re-processed if required)
- Better user experience (real-time notifications and session control)

2)Architecture Goals and Design Principles

Goals:

- Responsiveness: Requests shouldn't wait for email/SMS delivery.
- Reliability: Workflow signals must survive service restarts and temporary failures.
- Scalability: Increase throughput by scaling consumers (instances / consumer groups).
- Traceability: Inspect topics and messages via Kafdrop for debugging/audit support.
- Immediate UI control: Critical actions must reflect instantly on connected clients.

Design Principles:

- Oracle DB is the system of record (OTP tokens, user status, approvals).
- Kafka carries workflow signals, not data ownership.
- SignalR = “update UI now”, Kafka = “process consistently in background.”

3)System Context (What exists in EFMS)

- Frontend (Web UI): Calls EFMS APIs and maintains a SignalR connection.
- Backend (.NET API): Business logic + DB access + Kafka producers/consumers + SignalR hub.
- Oracle Databases:
    - EFMS (auth, OTP, users, approvals, setup)
    - AITKEN (fund-related domain data)
- Kafka Cluster: durable event pipeline for background workflows.
- Kafdrop: operational visibility (topics, partitions, messages).

System Context Diagram

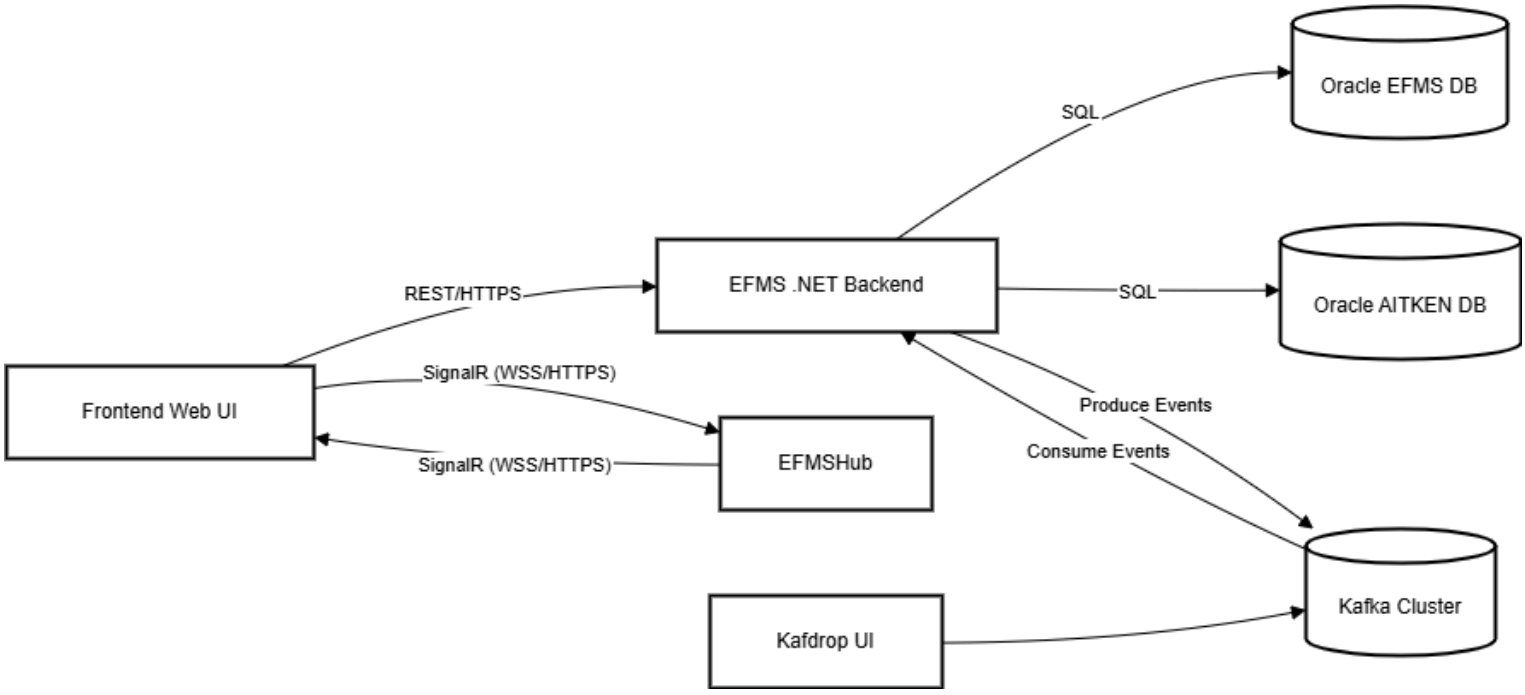


Image 1

4) Kafka Infrastructure (Docker Cluster)

4.1 Cluster Type

Your docker-compose creates a 3-broker Kafka cluster using KRaft (brokers are also controllers). This improves simplicity (no ZooKeeper) and supports replication and fault tolerance.

4.2 Listeners (Why there are two ports per broker)

You configured dual listeners to support both:

- Host machine apps (your EFMS backend running on localhost)
- Docker network apps (kafdrop + any containerized services)

Per broker:

- **SASL_PLAINTEXT://localhost:9092/9093/9094** → used by EFMS backend running on host
- **PLAINTEXT://kafkaX:19092/19093/19094** → used by Docker services (Kafdrop)
- **CONTROLLER://...** → internal KRaft controller quorum traffic

4.3 SASL/PLAIN authentication

Your server_jaas.conf defines Kafka users:

- producer / prod-secret
- consumer / cons-secret
- broker internal user

Efms uses:

```
SecurityProtocol = SASL_PLAINTEXT
SaslMechanism = PLAIN
SaslUsername = KafkaAuth.Username  // "producer"
SaslPassword = KafkaAuth.Password  // "prod-secret"
```

Kafka Deployment Diagram

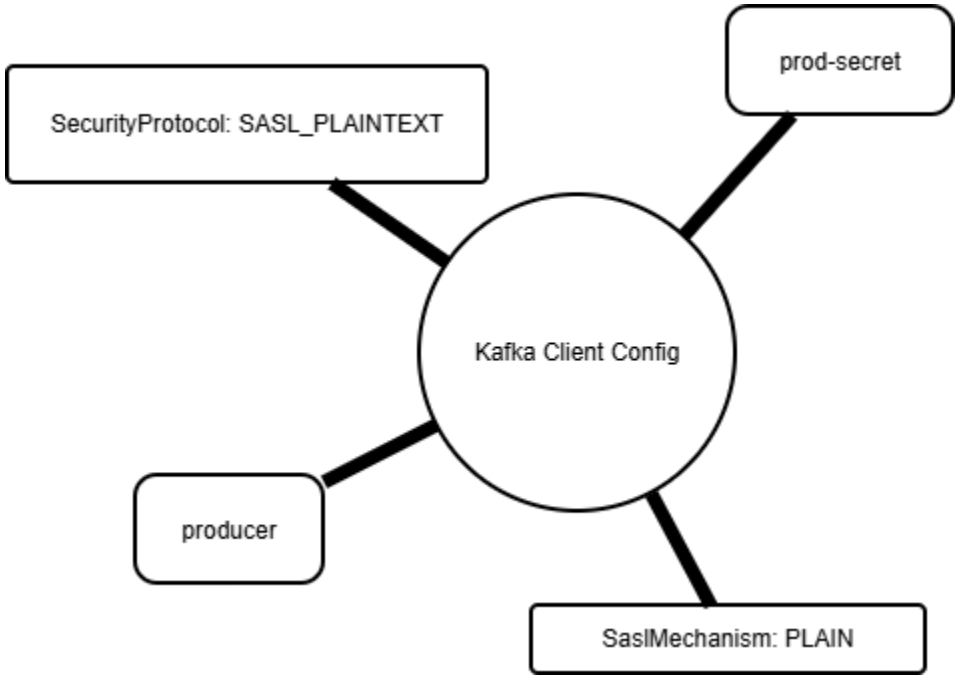


Image 2

5) Kafka in EFMS Backend (How your code uses Kafka)

5.1 Configuration (in-memory, currently)

Your backend config sets:

BootstrapServers = localhost:9092,localhost:9093,localhost:9094

GroupId = efms-session-expiry-group (with suffixes per consumer)

Topics:

- otp-service
- signup-rejection
- session-expiry

5.2 Producers (publish workflow events)

In the Code :

- **KafkaProducerService** → publishes OTP events to otp-service
- **SignupDecisionProducer** → publishes approval/rejection decision to signup-rejection
- **SessionExpiryProducer** → publishes logout/session-expiry signal to session-expiry

5.3 Consumers (background workers)

Hosted services in your backend:

- **EmailOtpConsumer** (consumer group: ...-email)
  Reads otp-service → sends email OTP
- **SmsOtpConsumer** (consumer group: ...-sms)
  Reads otp-service → placeholder for SMS provider integration
- **EmailSignupDecision**Consumer (consumer group: ...-signup)
  Reads signup-rejection → sends approval/rejection email
- **SessionExpiryWorker** (group: efms-session-expiry-group)
  Reads session-expiry → triggers ExpiryPushService.Schedule(userId, expiresAt)

Consumer Groups Diagram

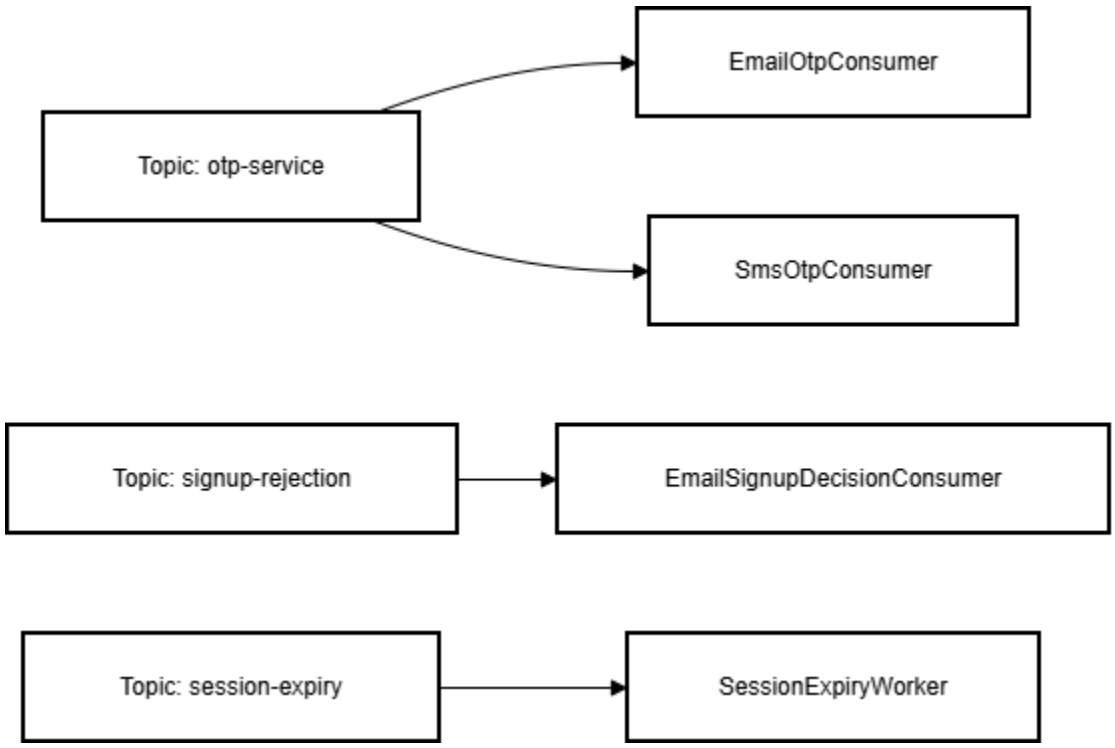


Image 3

6) SignalR in EFMS (Real-time UI control)

6.1 EFMSHub responsibilities

On connect:

- identifies user via JWT (token from query string or cookie)
- adds connection to a company group: company_{companyId}
- tracks connections via UserPresenceTracker

On disconnect:

- removes connection from tracker

6.2 UserPresenceTracker

This is how EFMS knows all active connections for a user so it can:

- send targeted events (only that user)
- force logout across all tabs/devices immediately

7) Key Workflows (End-to-end)

7.1 OTP Generation + Email Delivery (DB + Kafka)

What happens

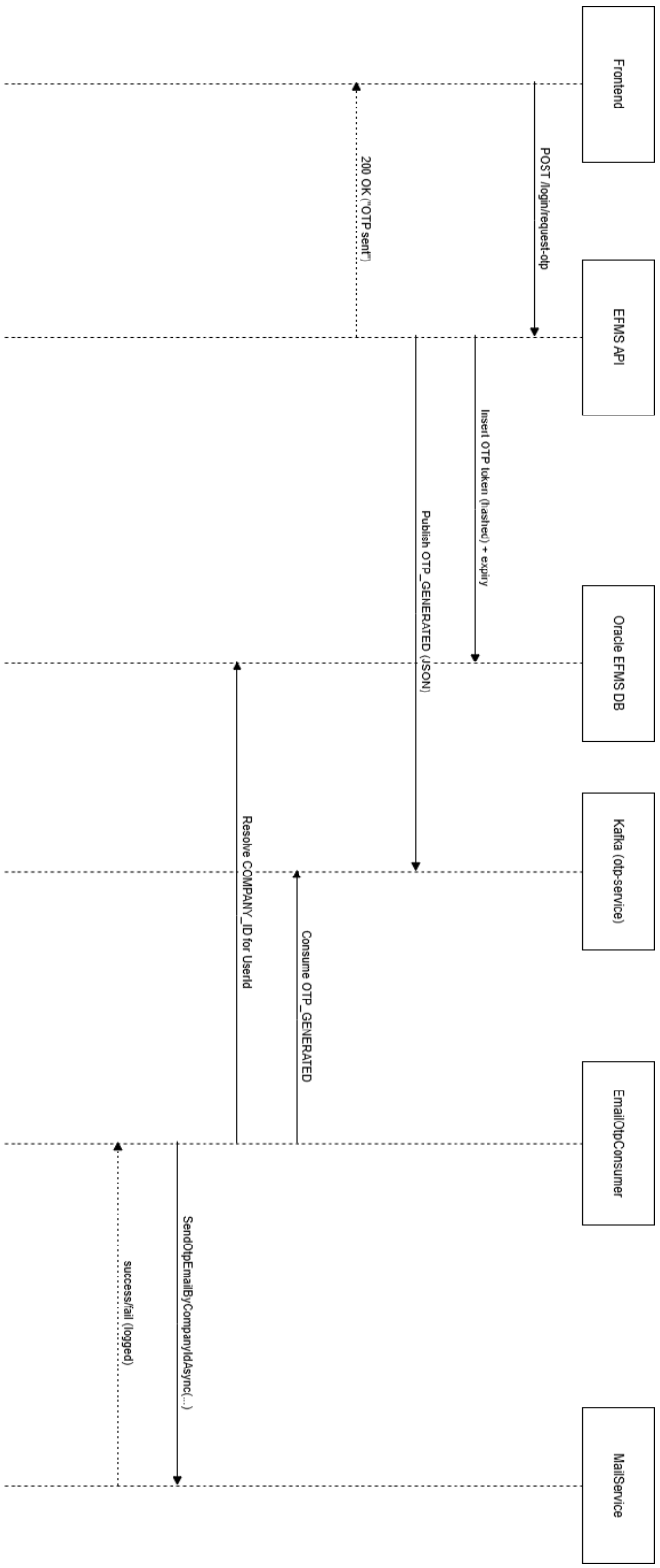
- API generates OTP and stores hashed OTP in EFMS DB.
- API publishes an OTP_GENERATED event to Kafka (otp-service).
- EmailOtpConsumer consumes it, resolves companyId (DB lookup), sends OTP email.

Why this design

- API returns fast (“OTP sent”) without waiting for email.
- Email sending is retryable and inspectable.

Sequence Diagram

Image 4



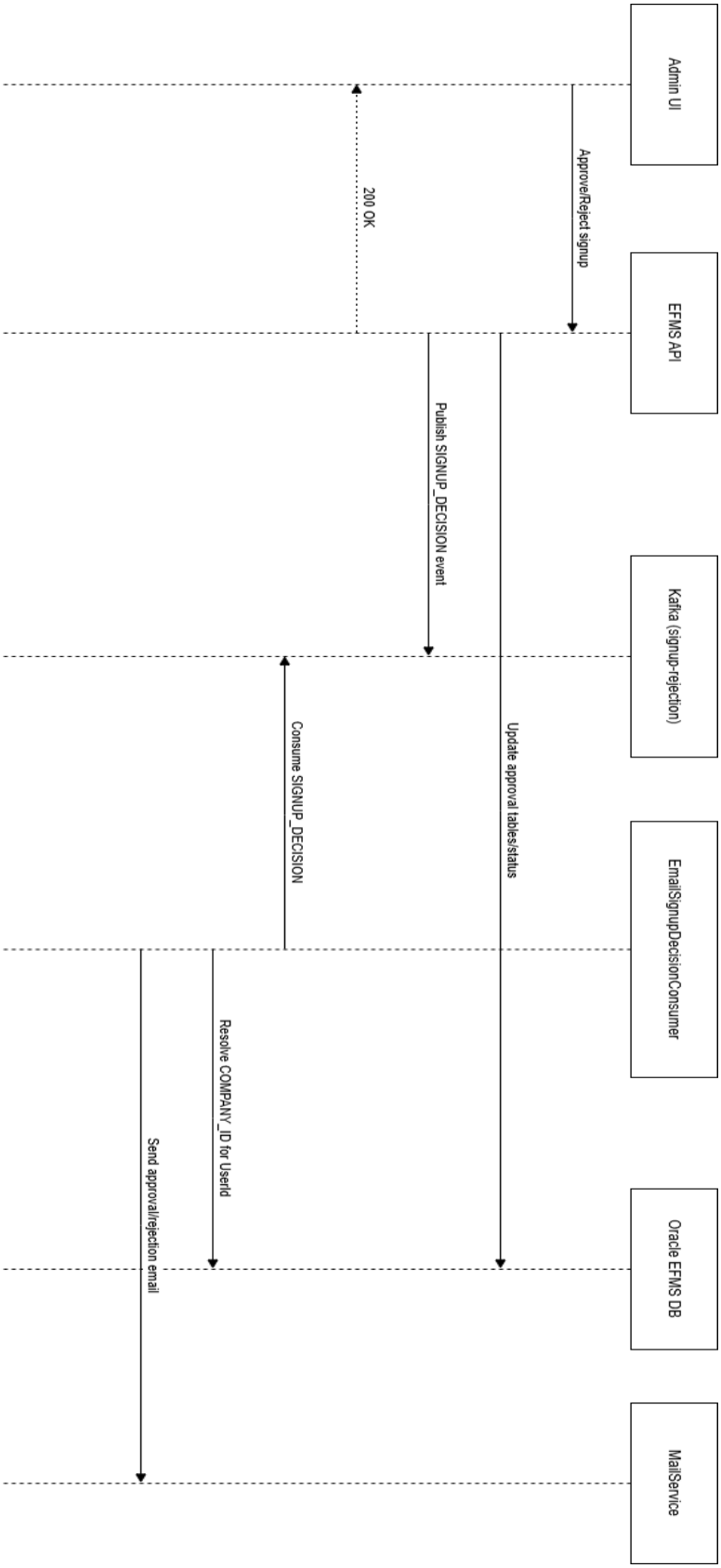
7.2 Signup Approval/Rejection Email (Kafka)

What happens

- After approval flow, EFMS publishes SignupDecisionEvent to topic signup-rejection (it currently carries both approved and rejected decisions).
- EmailSignupDecisionConsumer sends the appropriate email template.

Sequence Diagram

Image 5



7.3 User Deactivation → Force Logout (SignalR) + Backend Enforcement (Kafka)

What happens

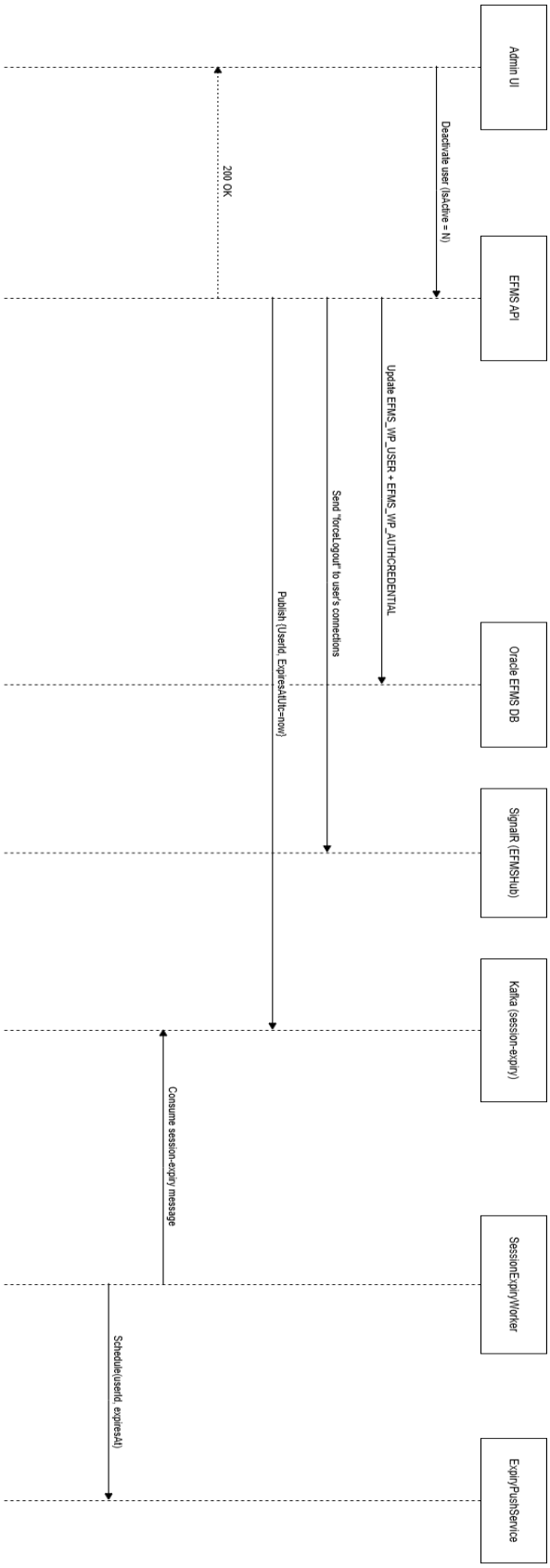
- Admin/API deactivates user in DB (EFMS_WP_USER + EFMS_WP_AUTHCREDENTIAL).
- EFMS sends SignalR forceLogout to all active connections of that user (immediate UI action).
- EFMS publishes Kafka session-expiry event (consistent backend enforcement).
- SessionExpiryWorker receives it and triggers expiry scheduling through ExpiryPushService.

Why both are used

- SignalR handles the “close user session now” experience.
- Kafka ensures “even if SignalR misses a client” the backend still processes the expiry event reliably.

Sequence Diagram

Image 6



8) Deployment Notes (Docker Hosting Reality)

8.1 If EFMS backend runs on the HOST machine (current config)

Use:

- Kafka:BootstrapServers = localhost:9092,localhost:9093,localhost:9094

Because those are SASL_PLAINTEXT ports exposed to host

8.2 If EFMS backend is moved into Docker (important when “host using docker”).

Inside Docker, localhost means the container itself, not the host.

So switch bootstrap servers to the docker listener addresses:

- kafka1:19092,kafka2:19093,kafka3:19094 (PLAINTEXT)

Recommended approach

- For container-to-container communication, typically enable SASL on the docker listener too (or keep PLAINTEXT only for dev).
- If keep the docker listener as PLAINTEXT (as now), ensure this is restricted to internal networks only.

Host vs Docker Connectivity Diagram

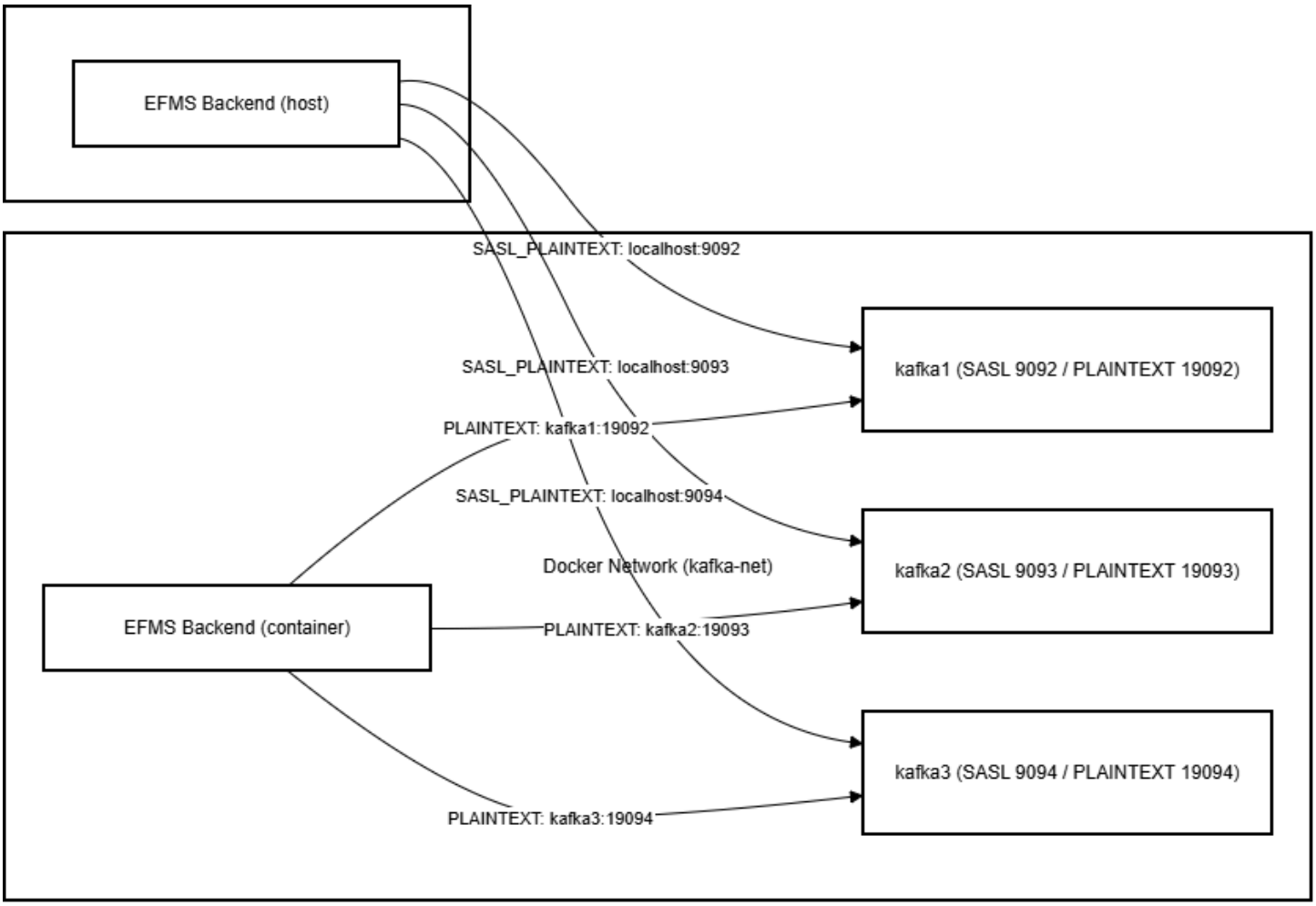


Image 7

9) Reliability & Operational Behavior

9.1 Message durability

Kafka stores messages on disk (replicated across brokers in your setup). This means:

- If EFMS restarts, consumers continue from last committed offset.
- If email service is temporarily down, messages remain in topic.

9.2 Re-processing

Because events remain in Kafka, you can re-process by:

- using a new consumer group id (starts fresh depending on offsets policy)
- or resetting offsets (admin operation)

9.3 Offsets and duplicates (what to expect)

Kafka consumers may deliver messages more than once in failure scenarios. That means:

- Email sending should be treated as **at-least-once** delivery unless you add idempotency controls.
- For OTP: duplicates are not fatal, but can confuse users. A common improvement is to include an **OtpId** in the event and store “sent status” in DB.

10) Observability (How you troubleshoot)

10.1 Kafdrop

Kafdrop helps you:

- see topic list
- confirm partitions/replication
- inspect messages (JSON payloads)
- verify consumer groups and lag

10.2 EFMS logging

Your hosted services log:

- “Listening…” startup lines
- JSON parse errors
- send failures
- consume errors

Recommended minimum logs per event

- event type
- user id
- topic + partition + offset
- timestamp
- outcome (success/failure + reason)

11) Event Contracts (What EFMS publishes)

11.1 OTP event (current shape)

EFMS publishes a JSON payload like:

- Event, Context, UserId, CompanyId, Channel, Target, Otp, ExpiresAt

- Improvement suggestion (still simple):
  Add:
- EventId (GUID)

- OtpId (DB key)
  So consumers can deduplicate.

11.2 SignupDecisionEvent (your class)

Fields:

- Event = SIGNUP_DECISION

- UserId, Email, Decision, Response, ApprovedBy, ApprovedName, Timestamp

11.3 Session expiry message (current)

Fields:

- UserId

- ExpiresAt (UTC)

12) Why this Architecture Works Well for EFMS

- OTPService stays fast: it writes OTP to DB and emits an event. Email sending becomes a background concern.
- Approval messaging is consistent: decision emails don’t slow down admin actions.
- Deactivation is immediate AND reliable:
    - SignalR forces logout for connected users instantly.
    - Kafka ensures backend enforcement even if SignalR misses a client.