System Design: Health Tables FHIR Connector Service

1. Overview

This document outlines the design of a **real-time synchronization connector** that integrates a PostgreSQL-based Health Tables database with a FHIR server. The connector guarantees that any modification to patient records and their associated identifiers is reflected on the FHIR server with **sub-second latency**. By leveraging FHIR's built-in versioning, the system maintains a **complete and immutable change history**, ensuring regulatory compliance, auditability, and data integrity across healthcare systems.

2. Assumptions

- Database is accessed concurrently by multiple applications, scripts, and direct SQL queries.
- The FHIR server automatically maintains version history for resources.
- Connector instances are **stateless** and all persistent state is maintained in PostgreSQL.
- The system must be **containerizable** for cloud-native deployment.

3. Constraints

- **Database**: PostgreSQL is the source of truth; no external state store is permitted.
- **Delivery Semantics**: Aim for exactly-once delivery; fallback is at-least-once with idempotency.
- **Latency**: Maximum allowed end-to-end delay is **< 1 second**.
- **Standards**: Mapping must adhere to **FHIR R4 Patient resource** specifications (preferably US Core).

4. Goals & Requirements

4.1 Functional Requirements

1. **Change Synchronization**
   - Detect and synchronize **INSERT, UPDATE, DELETE** events from the Health Tables
2. **Low-Latency Updates**
   - Deliver updates to the FHIR server within **< 1 second** of database modification
3. **Transactional Grouping**
   - Consolidate multiple changes in a **single transaction** into one FHIR update call
4. **Version Tracking**
   - Each database change results in a **new FHIR resource version**, ensuring an auditable trail.
5. **FHIR Resource Mapping**

- Map `patient` → FHIR `Patient` resource
- Map `patient_other_identifiers` → `Patient.identifier[]`

4.2 Non-Functional Requirements

1. **Scalability (Out of Scope)**
    - **Elastic scaling**: Connector instances can be scaled horizontally.
    - **Throughput handling**: Capable of processing bursts of thousands of events per second.
    - **Stateless design**: Enables load balancing and distribution across containers.
2. **Reliability**
    - **Durable persistence**: All events are logged in PostgreSQL before dispatch.
    - **Delivery guarantees**: Exactly-once goal; fallback at-least-once with idempotent FHIR writes.
    - **Crash recovery**: Automatic resumption of synchronization after restarts.
3. **Fault Tolerance**
    - **Retry with backoff**: Automatic retries for transient FHIR API failures (Response Code Specific)
    - **Dead Letter Queue (DLQ)**: Persist problematic events with error metadata for later resolution.
    - **Isolation of failures**: One failing event does not block others.
4. **Performance**
    - **Low latency**: Sub-second end-to-end propagation.
    - **Efficient event handling**: Use lightweight triggers, LISTEN/NOTIFY or WAL decoding for minimal DB overhead.
    - **Optimized batching**: Batch events where possible without compromising FHIR versioning.
5. **Properly Monitored**
    - **Metrics & Observability:** Collect metrics for latency, throughput, retry counts, and error rates.
    - **Probes**: Liveness/readiness checks for Docker.
    - **Alerting**: On high retry counts, latency violations, or DLQ growth.
    - **Tracing**: End-to-end correlation of DB events with FHIR updates
6. **Deployment and Operations**
    - **Stateless Connector:** All states reside in PostgreSQL (no in-memory state dependency).
    - **Containerized:** Packaged as a lightweight, reproducible container (Docker ready).
    - **Concurrency-safe**: Handles multiple DB writers and overlapping transactions.

5. Failure Scenarios & Mitigations

- **DB change missed due to connector crash** → Outbox table ensures recovery on restart.
- **FHIR server downtime** → Retry with exponential backoff, eventually DLQ.
- **Network partition** → Buffered persistence in PostgreSQL until connectivity restores.
- **Transaction bursts** → Horizontal scaling + batching mechanisms.

6. Database Schema Context

6.1 Tables involved

- `patient` → `Patient` demographics + primary identifier
- `patient_other_identifiers` → Additional `Patient.identifier` values
- Supporting tables (`form`, `blood_pressure`, `pulse`) are out of current scope, but could later map to FHIR `Encounter` and `Observation`.

6.2 Triggers already present

- Update triggers maintain `updated_at` timestamps.

7. System Components

7.1 Outbox Table

- Captures all committed DB changes with fields: `id, txid, table_name, record_id, op, payload_json, sequence_in_tx, created_at, processed, attempts, locked_by, lock_expires_at, processed_at, fhir_resource_id, fhir_version, last_error`.
- Ensures durability and transactional consistency.
- Serves as the central event log for synchronization and replay.

7.2 Database Triggers

- Defined on `patient` and `patient_other_identifiers` (AFTER INSERT/UPDATE/DELETE).
- Write one outbox row per changed record, tagged with `txid`. Grouping of related changes happens in the connector by collecting all rows for the same `txid`+`patient_id`.
- Issue `pg_notify` events to wake the connector for sub-second latency.

7.3 Connector Process (Stateless Workers)

- Stateless NodeJS service
- Subscribes to LISTEN/NOTIFY, with polling fallback
- Claims outbox rows safely using row locks (`locked_by`, `lock_expires_at`).
- Groups changes by `txid` for atomic FHIR updates.
- Transforms DB payload into FHIR Patient JSON.
- Performs idempotent FHIR calls:
  - Conditional `PUT /Patient?identifier=system|value`, or
  - Direct `PUT /Patient/{id}` when FHIR resource ID is known.
- On success: marks outbox row as processed and stores FHIR metadata.
- On failure: retries with backoff or moves record to DLQ.

7.4 Dead Letter Queue (DLQ)

- Stores permanently failing records with original payload, error, and attempts count.
- Provides operators with visibility into sync issues.
- Integrated with monitoring/alerting for proactive detection.

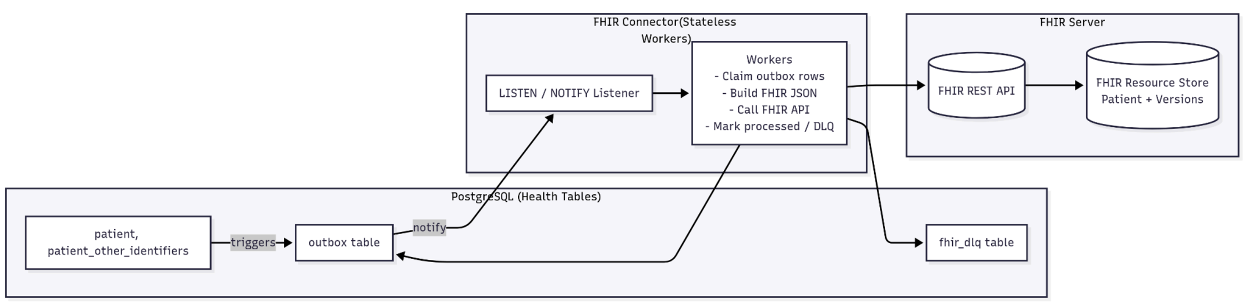
7.5 FHIR Server

- Medblocks Bootcamp FHIR server: https://fhir-bootcamp.medblocks.com/fhir/
- Exposes standard FHIR R4 REST API.
- Auto-versions resources on each update (`meta.versionId`).
- Provides a complete audit trail of all changes.
- Supports conditional operations for idempotency.

Having established the key components, the next section illustrates how they interact through different system flows (normal, retry, and transactional grouping).

8. System Architecture

8.1 High-Level Flow

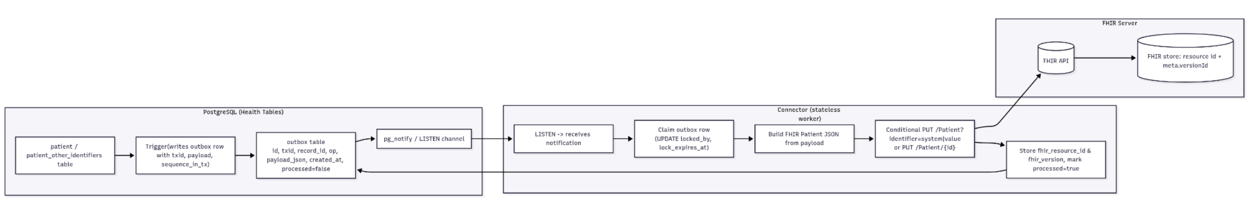


The above diagram shows the core system components viz. database tables, triggers, outbox, connector, DLQ, and FHIR server, and how they interact at a high level.

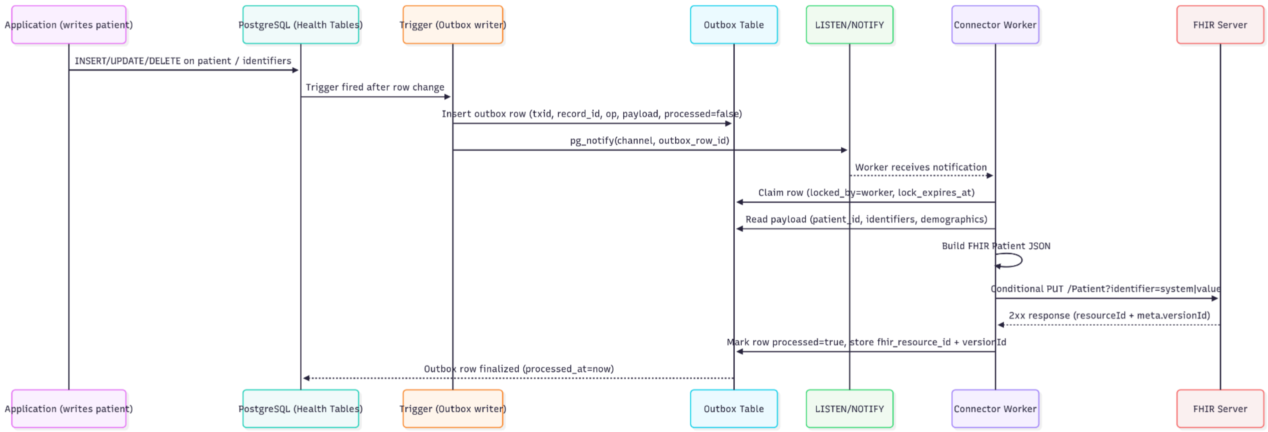
8.1 Normal Flow (INSERT/UPDATE/DELETE → FHIR)

Illustrates the standard synchronization path where a database change is written to the outbox, consumed by a connector worker, transformed into FHIR JSON, and successfully applied to the FHIR server.

8.1.1 Data Flow



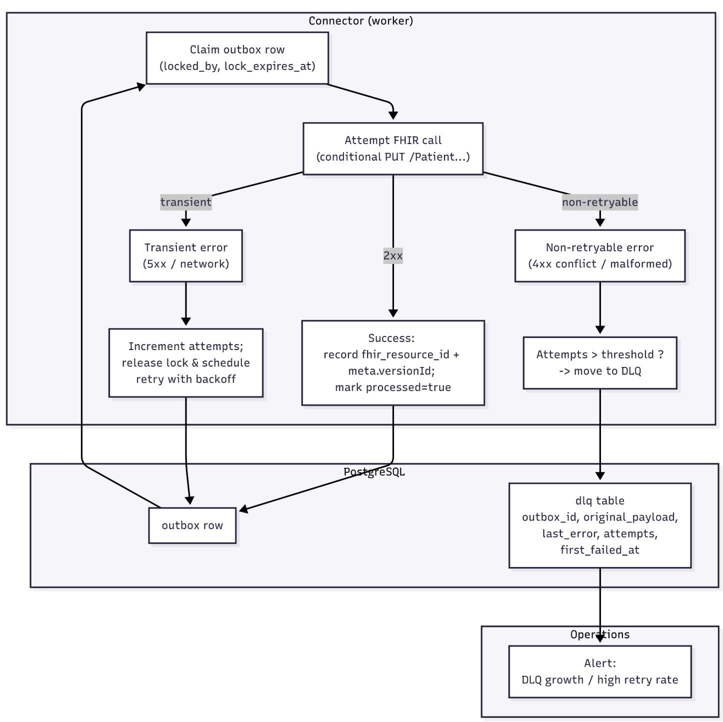
8.1.2 Sequence Diagram



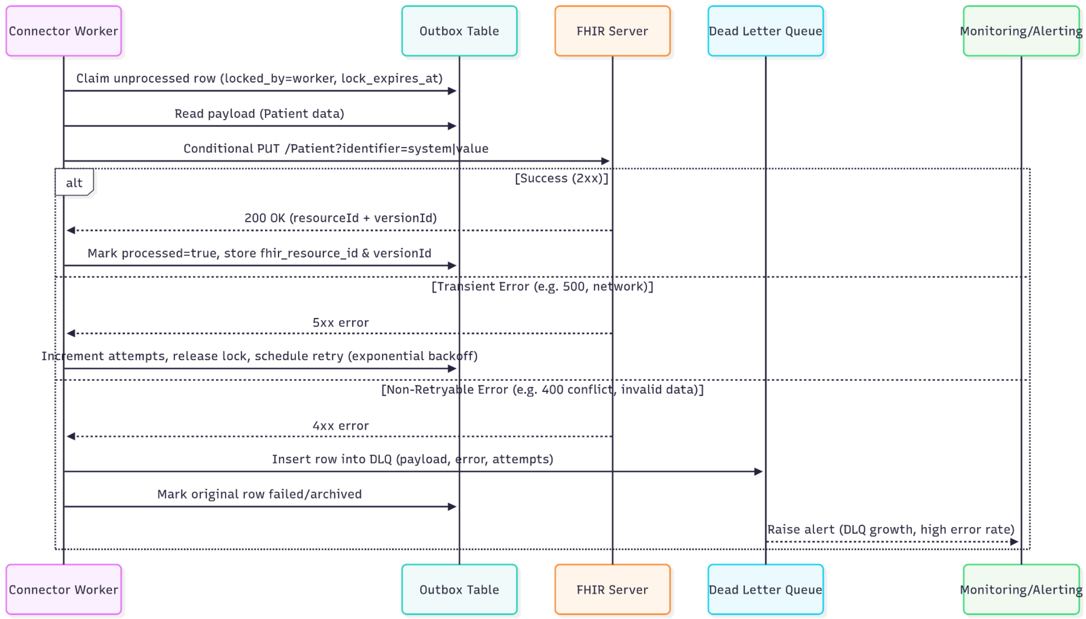
8.2 Retry & DLQ Flow

Depicts how the system handles errors: retrying transient failures with backoff, moving unrecoverable events into the DLQ, and raising alerts for operator visibility.

8.2.1 Data Flow



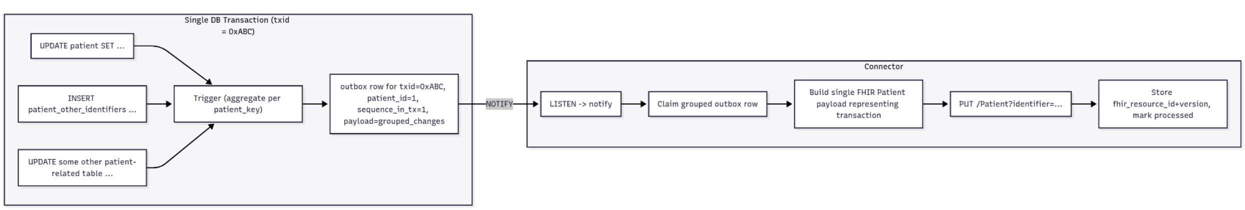
8.2.2 Sequence Diagram



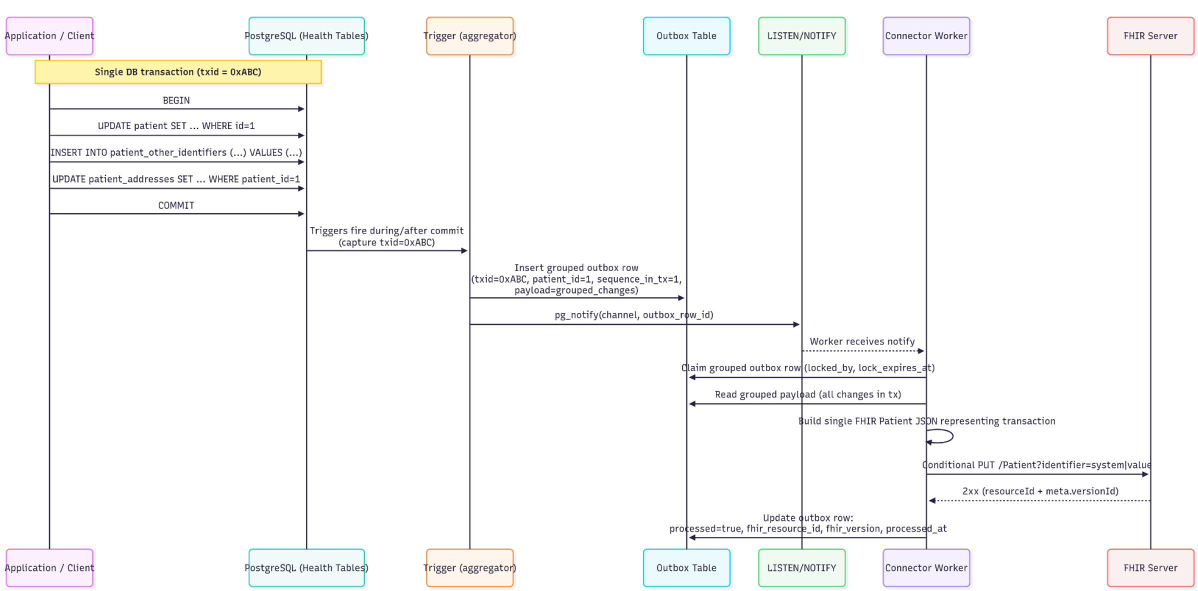
8.3 Transactional Grouping Flow

Demonstrates how multiple changes within a single database transaction are grouped into one outbox event and propagated as a single atomic FHIR update call.

8.3.1 Data Flow



8.3.2 Sequence Diagram



9. Database Modifications

9.1 Outbox Table

```
CREATE TABLE fhir_outbox (
  id BIGSERIAL PRIMARY KEY,
  txid BIGINT NOT NULL,
  table_name TEXT NOT NULL,
  record_id INTEGER NOT NULL,
  operation CHAR(1) NOT NULL,     -- 'I'|'U'|'D'
  payload_json JSONB NOT NULL,    -- full row payload (NEW or OLD)
  patient_id INTEGER,             -- extracted canonical patient id if
available
  identifier_system TEXT,         -- canonical identifier system (if
available)
  identifier_value TEXT,          -- canonical identifier value (if
available)
  created_at TIMESTAMPTZ NOT NULL DEFAULT now(),
  processed BOOLEAN NOT NULL DEFAULT FALSE,
  attempts INTEGER NOT NULL DEFAULT 0,
  next_retry_at TIMESTAMPTZ NULL,
  locked_by TEXT NULL,
  lock_expires_at TIMESTAMPTZ NULL,
  processed_at TIMESTAMPTZ NULL,
  fhir_resource_id TEXT NULL,
  fhir_version TEXT NULL,
  last_error TEXT NULL
);

-- Useful indexes
CREATE INDEX idx_fhir_outbox_unprocessed ON fhir_outbox (processed,
next_retry_at, created_at);
CREATE INDEX idx_fhir_outbox_txid ON fhir_outbox (txid);
CREATE INDEX idx_fhir_outbox_patient ON fhir_outbox (patient_id);
```

9.2 Trigger Example

```
CREATE OR REPLACE FUNCTION fhir_outbox_trigger()
RETURNS TRIGGER AS $$
DECLARE
  payload jsonb;
  rec_id integer;
  outbox_id bigint;
BEGIN
  IF TG_OP = 'DELETE' THEN
    payload := to_jsonb(OLD);
```

```
    rec_id := OLD.id;
  ELSE
    payload := to_jsonb(NEW);
    rec_id := NEW.id;
  END IF;

  INSERT INTO fhir_outbox (
    txid, table_name, record_id, operation, payload_json, patient_id,
    identifier_system, identifier_value, created_at
  )
  VALUES (
    txid_current(),
    TG_TABLE_NAME,
    rec_id,
    SUBSTRING(TG_OP, 1, 1),
    payload,
    -- optional: extract patient_id / identifiers if present
    (payload ->> 'patient_id')::int,
    (payload ->> 'identifier_system'),
    (payload ->> 'identifier_value'),
    now()
  )
  RETURNING id INTO outbox_id;

  PERFORM pg_notify('fhir_outbox_event', outbox_id::text);

  -- For AFTER triggers: return NEW, for DELETE return OLD as appropriate
  IF TG_OP = 'DELETE' THEN
    RETURN OLD;
  ELSE
    RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

10. FHIR Mapping

10.1 Patient (Field Maping)

10.1.1 Identifiers

- **Primary Identifier**
  - `patient.identifier_value` + `identifier_system` → `Patient.identifier[0]`
- **Additional Identifiers**
  - `patient_other_identifiers` → additional `Patient.identifier[]` elements

- Each entry maps to:
  - `identifier.value` ← `identifier_value`
  - `identifier.system` ← `identifier_system`
  - `identifier.type` (optional, if type information available in schema)

10.1.2 Demographics

- **Name**
  - `name_family`, `name_given`, `name_text` → `Patient.name`
  - `name_family` → `name.family`
  - `name_given` → `name.given[]`
  - `name_text` → `name.text` (full name as a single string)
- **Birth Date**
  - `birth_date` → `Patient.birthDate`
- **Gender**
  - `gender` → `Patient.gender` (must conform to FHIR values: `male | female | other | unknown`)

10.1.3 Contact Information

- **Phone Number**
  - `phone_number` → `Patient.telecom[]` with `system = phone` and `value = phone_number`
- **Email**
  - `email` → `Patient.telecom[]` with `system = email` and `value = email`
- Multiple telecom entries supported.

10.1.4 Address

- `address_line` → `Patient.address.line[]`
- `address_city` → `Patient.address.city`
- `address_state` → `Patient.address.state`
- `address_postal_code` → `Patient.address.postalCode`
- `address_country` → `Patient.address.country`

10.2 Example Patient Resource (JSON)

10.2.1 Health Tables Input

`patient`

```
id: 1
name_family: "Smith"
name_given: "John"
birth_date: "1990-01-15"
gender: "male"
phone_number: "+1-555-123-4567"
email: "john.smith@example.com"
address_line: "123 Main Street"
```

```
address_city: "Boston"
address_state: "MA"
address_postal_code: "02101"
address_country: "USA"
identifier_value: "MRN-001234"
identifier_system: "https://hospital.example.com/mrn"
```

patient_other_identifiers

```
patient_id: 1
system: "https://hospital.example.com/mrn"
value: "MRN-001234"
```

10.2.2 Mapped FHIR Patient Resource (JSON)

```json
{
  "resourceType": "Patient",
  "id": "1",
  "identifier": [
    {
      "system": "https://hospital.example.com/mrn",
      "value": "MRN-001234"
    }
  ],
  "name": [
    {
      "family": "Smith",
      "given": ["John"],
      "text": "John Smith"
    }
  ],
  "birthDate": "1990-01-15",
  "gender": "male",
  "telecom": [
    {
      "system": "phone",
      "value": "+1-555-123-4567"
    },
    {
      "system": "email",
      "value": "john.smith@example.com"
    }
  ],
  "address": [
    {
      "line": ["123 Main Street"],
```

```
      "city": "Boston",
      "state": "MA",
      "postalCode": "02101",
      "country": "USA"
    }
  ]
}
```

11. Concurrency & Delivery Guarantees

Goal: reliably propagate every committed database change to FHIR with preserved order and version history. Aim for effective exactly-once delivery by combining durable outbox semantics, idempotent conditional FHIR operations, and safe worker concurrency controls.

Key guarantees & mechanisms

- **Transaction grouping**
  - Triggers insert one outbox row per changed row containing `txid` and `patient_id`.
  - Connector groups processing by `txid+patient_id`: when a worker claims an outbox row it looks for any other unprocessed rows with the same `txid` and `patient_id` and processes them together in one FHIR call (atomically at worker level). This is operationally simple and robust.
- **Durable outbox**
  - All events persist in PostgreSQL (`outbox` table) with fields such as `txid, table_name, record_id, payload_json, created_at, processed, attempts, locked_by, lock_expires_at, fhir_resource_id, fhir_version, last_error`.
- **Worker claiming**
  - Workers claim events atomically (`locked_by` + `lock_expires_at` or `FOR UPDATE SKIP LOCKED`) to avoid concurrent processing of the same event. Abandoned locks can be reclaimed after expiry.
  - Pseudocode can be found in the appendix here.
- **Idempotency via conditional FHIR updates**
  - Primary strategy: `PUT /Patient?identifier={system}|{value}` ensures idempotency.
  - **Caveat**: This requires `identifier_system + identifier_value` to be unique in the FHIR server.
  - On first success, store the returned `fhir_resource_id` and `meta.versionId` in the outbox.
  - For subsequent updates, prefer `PUT /Patient/{id}` with `If-Match` when possible, which avoids ambiguity and improves efficiency.
- **Delivery semantics**

    - The system implements **at-least-once** delivery by default; idempotent updates + stored FHIR `resource id`/`version` enable effective exactly-once behavior for the target state.
- **Retries & backoff**
    - On transient errors increment `attempts`, release lock, and retry with exponential backoff. Permanently failing events are moved to a **Dead Letter Queue (DLQ)** for manual resolution.
- **Ordering & version history**
    - Each committed change (even multiple commits within a second) results in a separate outbox event and corresponding FHIR API call. Store `meta.versionId` returned by FHIR to maintain a 1:1 audit trail between DB changes and FHIR versions.
- **Deletes**
    - For hard deletes: call `DELETE /Patient/{id}` if `fhir_resource_id` is known.
    - For conditional deletes: call `DELETE /Patient?identifier={system}|{value}` if resource ID not known.
    - If FHIR server policy discourages hard deletes, update the resource with `active=false` instead (soft delete).
    - All delete operations must be idempotent (deleting an already-deleted or inactive resource is considered success).
- **Observability**
  Below metrics are proposed to be pushed to the prometheus, with give thresholds:
    - `fhir_outbox_lag_seconds` — age of oldest unprocessed event. Alert if > 60s (since your SLA is <1s, set a stricter alert e.g. > 5s for dev, > 30s for production).
    - `fhir_outbox_unprocessed_count` — alert on sustained > X (e.g., >1000).
    - `fhir_connector_success_total` / `fhir_connector_failure_total`.
    - `fhir_dlq_size` — alert if > 0 for prolonged time or > threshold.
    - `fhir_call_latency_seconds` (histogram) — monitor p50/p95/p99. Alert if p95 > 1s.
    - Instrument tracing: `trace_id` propagated from DB txid → outbox id → FHIR call so you can trace a single change end-to-end.

12. Technology Choices & Justification

12.1 Event Detection Strategy (Triggers vs WAL vs Polling)

- **Triggers + Outbox Table (Chosen)**
    - Provides transactional consistency by writing events only after a transaction commits.
    - Ensures grouping of related changes (e.g., patient and identifiers in the same transaction).
    - Minimal infrastructure requirement: only PostgreSQL features.

12.2 Delivery Guarantees (Exactly-once vs At-least-once)

- **Exactly-once goal, fallback to at-least-once with idempotency.**

- Achieved by persisting all events in the outbox and replaying failed deliveries without data loss.
- Idempotent FHIR API calls ensure duplicate events do not corrupt downstream state.

12.3 Stateless Architecture Rationale

- Connector is **stateless**; all state (outbox, retries, DLQ) lives in PostgreSQL.
- Enables **horizontal scalability** — multiple connector instances can run in parallel without coordination.
- Simplifies deployment in **containerized environments**.

13. Deployment & Operations

13.1 Containerization (Docker)

Dockerized Connector:

- Connector packaged as a lightweight Docker image.
- Includes runtime NodeJS and all required dependencies.
- Enables reproducible local development and straightforward deployment on any host.

Configuration:

- Database credentials, listen channel, retry policy, and FHIR server URL provided as environment variables.
- FHIR endpoint is fixed to the **Medblocks Bootcamp FHIR server**: `https://fhir-bootcamp.medblocks.com/fhir/`

13.2 Environment Setup (Development Only)

Local Development:

- PostgreSQL and connector run in Docker (or via Docker Compose).
- Developers can test schema changes, triggers, and outbox → connector → FHIR sync loop locally.
- All synchronization is directed to the **Medblocks Bootcamp FHIR server** for validation.

Deployment Scope:

- Currently limited to a **single development environment**.
- Future-ready for staging/production expansion if required.

14. Operational Concerns

This section summarizes the key operational practices for running and validating the connector in a Docker-based development environment.

- **Error Handling**
  - Transient errors retried with exponential backoff.
  - Permanently failing records moved to DLQ for manual resolution.
- **Monitoring (Ops-Focused View)**
  - Track outbox lag (age of oldest unprocessed event).

    - Observe throughput (events/sec), error rates, and DLQ growth.
    - Logs and traces must capture DB txid ↔ outbox row ↔ FHIR version linkage.
- **Scaling**
    - Multiple workers can run in parallel due to stateless design.
    - Partitioning (e.g., by patient ID) ensures per-patient consistency.
    - In current scope: single worker via Docker; future-ready for scaling.
- **Security**
    - DB connections must use TLS in production-ready setups.
    - For this assignment, the FHIR server is unauthenticated (`https://fhir-bootcamp.medblocks.com/fhir/`).
- **Testing Scenarios**
    - Rapid successive updates → ensure multiple FHIR versions are created.
    - Concurrent multi-row edits → validate grouping into single atomic FHIR update.
    - Deletes → confirm correct FHIR `DELETE` or `active=false` behavior.
    - Crash recovery → connector restarts without data loss, outbox replays correctly.

15. Trade-offs & Alternatives

15.1 Triggers + Outbox (Chosen Approach)

Pros:

- **Simple and self-contained** — only requires PostgreSQL features.
- **Transaction-aware**, so changes from the same transaction are grouped correctly.
- Easier to reason about and debug (outbox table is transparent to developers).

Cons:

- Adds **slight overhead** to database writes (due to triggers).
- Requires a **custom worker process** to consume outbox events and call FHIR API.

15.2 CDC via Logical Decoding (e.g., Debezium, WAL Streaming)

Pros:

- Non-invasive, no schema changes or triggers required.
- High scalability, can capture all DB changes at the WAL level.

Cons:

- **Complex infrastructure** (Kafka, Debezium, connectors).
- Harder to implement **transaction grouping** (since WAL operates at row-level granularity).
- Requires additional ops overhead for monitoring and maintaining Debezium cluster.

15.3 Periodic Polling

Pros:

- Easiest to implement, no DB triggers or WAL decoding needed.
- Low operational complexity.

Cons:

- **High latency** (depends on poll interval, usually seconds to minutes).
- Wastes resources by repeatedly scanning tables.
- Doesn't handle bursts efficiently.

17. Deliverables

- **Source Code**
  - Connector application (stateless worker) with full source code.
  - Includes `README.md` with setup instructions.
  - Includes `Dockerfile` for containerized build and run.
  - Repository hosted on GitHub: 🔗 GitHub Link – FHIR Connector

- **System Design Document**
  - This document, describing functional requirements, non-functional requirements, architecture, data flows, concurrency guarantees, deployment details, and operational concerns.
  - Includes diagrams: component overview, normal flow, retry/DLQ, and transactional grouping.

- **Test Scenarios**
  - **CRUD Operations**: Create, update, and delete patients and identifiers.
  - **Concurrency**: Verify correctness with multiple writers and simultaneous updates.
  - **Latency**: Confirm sub-second propagation from DB change → FHIR server update.
  - **Rapid Successive Updates**: Ensure multiple committed updates result in multiple FHIR versions.
  - **Transaction Grouping**: Multi-row edits in one transaction propagate as a single atomic FHIR update.
  - **Delete Behavior**: Verify deletes map correctly to FHIR deletes or `active=false`.
  - **Crash Recovery**: Simulate connector restarts; confirm no data loss and correct replay.
  - **DLQ Handling**: Confirm failing records are retried and escalated to DLQ when unrecoverable.

18. Open Questions

- ~~Validate the approach~~
  - ~~CDC using plugins vs triggers + outbox~~
- ~~Handling of the API failures on the FHIR Server (4xx)~~
- ~~Monitoring & Logging~~

Appendix

A. Worker claiming + processing pseudocode

1. `LISTEN 'fhir_outbox_event'` (wake-up), also poll `fhir_outbox` as fallback.
2. Attempt to claim a row:

```
BEGIN;
SELECT id, payload_json, txid, patient_id
FROM fhir_outbox
WHERE processed = false
  AND (next_retry_at IS NULL OR next_retry_at <= now())
ORDER BY created_at
FOR UPDATE SKIP LOCKED
LIMIT 1;
-- if row found, UPDATE fhir_outbox SET locked_by = '<worker-id>',
lock_expires_at = now() + interval '30s' WHERE id = <id>;
COMMIT;
```

3. After claim: look for **other rows with same** `txid` **&** `patient_id` (unprocessed) and fetch them for grouping.
4. Build FHIR JSON (merge all changes in grouped payloads).
5. Make FHIR call:
    a. If you have `fhir_resource_id`: `PUT /Patient/{id}` with `If-Match` if available.
    b. Else: `PUT /Patient?identifier={system}|{value}` (conditional update).
6. On **success** (2xx):
    a. Update outbox rows atomically: set `processed = true`, `processed_at = now()`, `fhir_resource_id`, `fhir_version`, clear `locked_by`.
7. On **transient failure**:
    a. `attempts = attempts + 1`, set `next_retry_at = now() + backoff(attempts)`, clear `locked_by`.
8. On **non-retryable failure** (e.g., malformed payload, 4xx not fixable):
    a. Move original rows into `fhir_dlq` and mark outbox rows processed/archived or mark with `last_error` and a `failed=true` state — surface for manual remediation.