PETI Read-API technical plan

Version 24.11.2025

Summary

- Purpose: Read-only API for exporting NeTEx entities from a PostgreSQL table (ext_cache_netex_entities), updated via event listeners and a nightly batch process.
- Data Source: Table is repopulated nightly and updated in real-time by EntityChangedListener events.
- API: Exposes REST endpoints (and optionally S3/exports) for filtered entity access (e.g., by transport_mode and area_codes).
- Entities: Supports multiple NeTEx types (StopPlace, Parking, ScheduledStopPoint, etc.), all pre-marshalled as XML.
- Update Logic: Uses strict transaction management and UPSERTs to avoid race conditions; marks old rows as STALE, upserts new/changed data, and deletes obsolete rows.
- Querying: Single query fetches all relevant entities, ordered for correct NeTEx output, with efficient filtering and referencing.
- Extensibility: Designed to add new entity types and listeners with minimal changes.
- Caveats: Requires careful transaction isolation (SERIALIZABLE), robust indexing, and failure handling to ensure data consistency and performance.

Technical spec

- A read-only API is served from a PostgreSQL table named `ext_cache_netex_entities`.
  - Table entities are updated based on new events from `EntityChangedListener`.
  - The table is repopulated once a day (nightly).
- Exports are provided either:
  - a) Via REST or Async S3 from the `ext_cache_netex_entities` table
  - b) Through a separate export process using Tiamat Native functionality
- No changes are required to the Tiamat internal data structure or Hibernate layer.
  - New onChange event listeners may be required for some entity classes, such as Parking, TopographicPlace, etc.

Tiamat / Spring Boot

- Component: `FintrafficEntityChangePublisher`
  - Active only with the `"fintraffic"` profile
  - Extends `EntityChangedEventPublisher`
  - Implements `EntityChangedListener`
  - Responsibilities:
    - Listens for changes in `EntityInVersionStructure` and writes changes to the `ext_cache_netex_entities` table.

- If `EntityChangedEvent.CrudAction` is `DELETE` or `REMOVE`, sets the status in the database to `DELETED`.
- Determines which `TopographicPlaces` (TVV area code in Finland) overlap with the `EntityInVersionStructure` and sets `area_codes` accordingly.

PostgreSQL

Postgres Table: `ext_cache_netex_entities`

- `id` TEXT PRIMARY KEY
- `type` TEXT NOT NULL -- e.g. "StopPlace", "Parking", "ScheduledStopPoint", etc.
- `transport_mode` TEXT NOT NULL
- `area_codes` CHAR(3)[] NOT NULL
- `xml` TEXT NOT NULL
- `stop_place_version` INT NOT NULL
- `stop_place_changed` BIGINT NOT NULL
- `status` ENUM('STALE', 'CURRENT', 'DELETED')
- `cache_entity_updated` TIMESTAMP DEFAULT now()
- `parent_refs` TEXT[] -- references to other entities (e.g. StopPlace id for ScheduledStopPoint)
- Index on (`transport_mode` , `area_codes`)
  - Supports queries such as: `SELECT id, xml FROM ext_cache_netex_entities WHERE 'MHU' = ANY(area_codes) AND transport_mode = 'bus'`

Update Process (executed as a separate task once a day)

- Before updating, mark all rows as `STALE` (in a separate transaction).
- During the update, update a row only if it does not have a newer version (use UPSERT).
  - If a row has been updated after being marked as `STALE`, keep the latest version.
- After the update, remove all rows marked as `STALE` or `DELETED` (in a separate transaction).

REST API Endpoints

- `/api/ext/fintraffic/v1/entities/?transportMode=bus&areaCode=MHU`
- The API returns only entities with a status of `STALE` or `CURRENT`.
- To be decided later:
  - Return full PublicationDelivery or only specific Frames (SiteFrame, ServiceFrame + SiteFrame)

Entity Model and Querying

- The table contains multiple NeTEx entity types: StopPlace, ScheduledStopPoint, PassengerStopAssignment, Parking, etc.
- All data is fetched in a single query, ordered as follows:
  a. ScheduledStopPoint

b. PassengerStopAssignment
c. StopPlace
d. Parking

- This approach ensures:
  - All records are from the same snapshot (single transaction).
  - Entities are sorted for correct NeTEx XML output.
  - The model is extensible to new types (e.g. FareZones).
  - There is no mixed logic of raw XML and marshalling XML from entities; all entities are pre-marshalled.
- ScheduledStopPoint and PassengerStopAssignment are derived from StopPlace and written to the table, referencing the StopPlace via `parent_refs`.

Example SQL Query

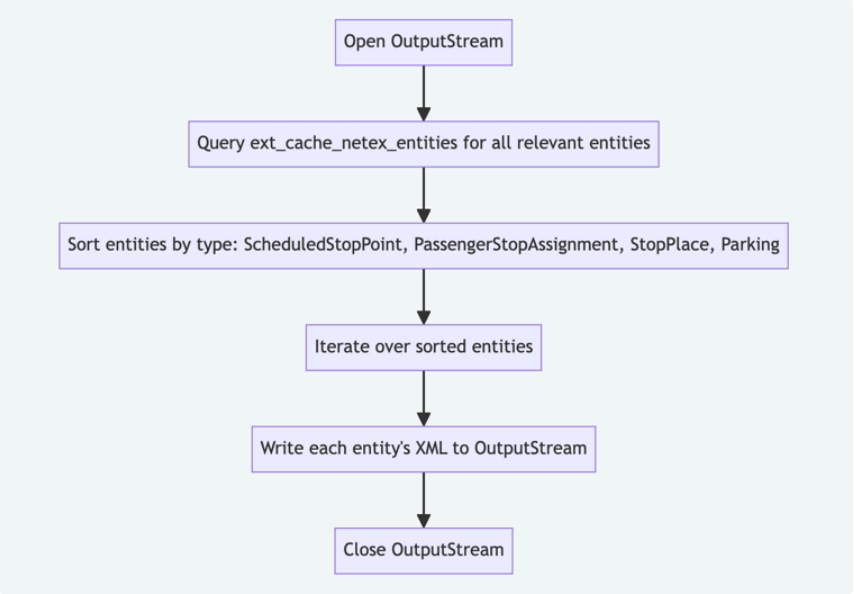
```
SELECT *
FROM ext_cache_netex_entities
WHERE (
    type = 'StopPlace' AND 'MHU' = ANY(area_codes) AND transport_mode = 'bus'
) OR (
    type IN ('ScheduledStopPoint', 'PassengerStopAssignment', 'Parking') AND parent_refs && (
        SELECT ARRAY(SELECT id FROM ext_cache_netex_entities WHERE type = 'StopPlace'
AND 'MHU' = ANY(area_codes) AND transport_mode = 'bus')
    )
)
AND status IN ('STALE', 'CURRENT')
ORDER BY
    CASE type
        WHEN 'ScheduledStopPoint' THEN 1
        WHEN 'PassengerStopAssignment' THEN 2
        WHEN 'StopPlace' THEN 3
        WHEN 'Parking' THEN 4
        ELSE 5
    END;

```

- This query fetches all relevant entities in a single snapshot, ordered for correct NeTEx output. Filtering is done for StopPlace by `area_codes` and `transport_mode`. Related entities are included if they reference the filtered StopPlaces via `parent_refs`.

Implementation details

High-level process diagram for composing the XML file in a single SQL transaction



- The implementation opens an output stream, queries the database for all relevant entities, sorts them by the required order, iterates over the sorted entities, writes each entity's XML to the output stream, and finally closes the stream.

Rework / refactoring / changes

- Create a new FintrafficApiService and FintrafficApiController for reading and streaming entities from `ext_cache_netex_entities`.
- Create a new FintrafficApiController for new APIs.
- Create a cache table repopulation task (nightly).
- Create a new listener for EntityChangedListener.
  - Attach onChange/onDelete listeners to all supported NeTEx object types (Parking, TopographicPlace).
    - FareZone and TariffZone to be added later. The implementation has theoretical support for them.

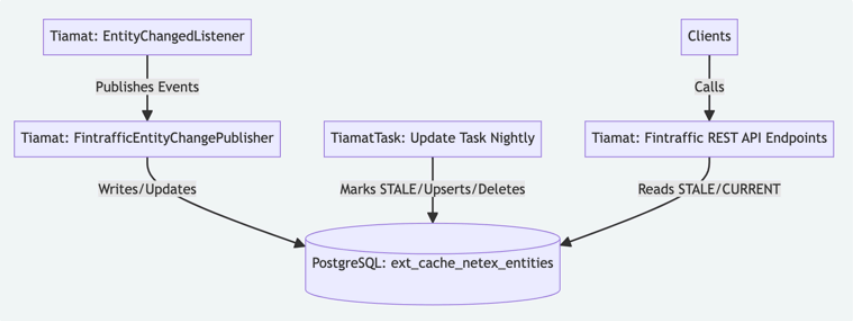
Update Process

1. The update task marks all rows as STALE. (Transaction 1)
2. The update task fetches new data (this takes approximately 15 minutes).
3. During this time, a row may be updated by the main application and marked as CURRENT. (Transaction 2)
4. The update task then starts writing new data to the table. It should skip all rows that are marked as CURRENT and have stop_place_changed > EXCLUDED.stop_place_changed and stop_place_version > EXCLUDED.stop_place_version. (Transaction 3)

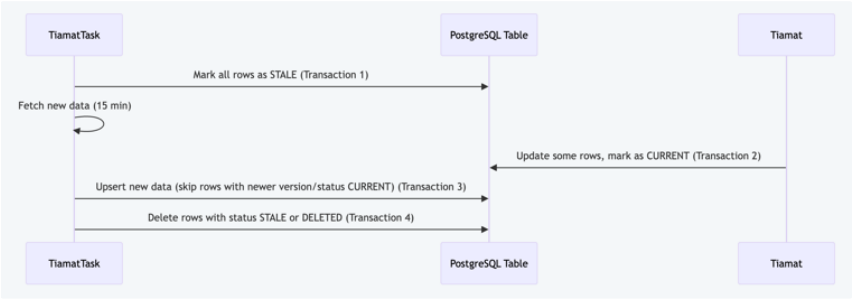
Transaction management

- Use strict UPSERT logic: always compare both status and version fields before overwriting.
- Log or audit changes during the update window to detect and resolve conflicts.
- Use SERIALIZABLE for maximum safety and to avoid all concurrency anomalies.
  - Transactionality :: Spring Data Relational

StopPlace read API



StopPlace cache table transaction management



Checklist

- [ ] Race conditions: If the main app updates a row after it is marked STALE but before the upsert, there is a risk of lost updates unless transaction isolation is strictly enforced.
- [ ] Indexing: The index on area_codes (an array) and transport_mode may not be efficient for large tables; consider GIN indexes for arrays.
- [ ] Status handling: The API returns STALE rows, which may be confusing for clients expecting only up-to-date data.
- [ ] Transaction isolation: The plan suggests using SERIALIZABLE isolation, which can impact performance and may cause transaction rollbacks under high concurrency.
- [ ] Data consistency: If the update process is interrupted, the table may contain a mix of STALE, CURRENT, and partially updated rows.
- [ ] Failure handling: Handle network issues and other sudden problems with a retry logic.