

Eugene Facts Route — End-to-End Flow

Developer walkthrough: HTTP request → FastAPI router → shared n-hop Cypher → FactsMapper → sentence-shaped ListResponse

Audience

Engineers who need to understand how Eugene turns its graph neighborhood into LLM-ready natural-language sentences. The facts route is a thin re-skin over the same n-hop Cypher used by `/graph/relationship/start/{start_id}`, but where that route returns an adjacency-list Graph, this one collapses each (start, rel, end) triple into a single English sentence and returns them as a flat list of strings. Every reference uses the form `path/to/file.py:line`.

Reference endpoint

- GET `/graph/facts/start/{start_id}?page=1&page_size=10` — 1-hop facts around the anchor, paginated.
- **Hop depth is fixed at 1.** The router handler hard-codes `n_hop=1` when calling the orchestrator (line ~69). A commented-out `n_hop` query param sits in the signature as a reminder — do not uncomment without also reworking the mapper, which assumes a star-shaped subgraph.
- `page` is `gt=0`, `page_size` is `gt=0, le=50` — Pydantic rejects 0 / 51+ before the handler runs.
- `start_id` is `Annotated[str]` with `max_length=512` and an `AfterValidator(validate_id)`.

Caveat

Facts are derived directly from graph topology. There is no curated sentence template per relationship type — `FactsMapper._to_fact` mechanically interpolates the label and the relationship type into the same string shape. The resulting sentences are *truthful* but read awkwardly for some rel types (e.g. `drug_drug` becomes a `relationship to drug` clause via the redundancy fix-up). If you need polished prose, do it in the LLM layer, not here.

0. Schema (read this first)

Facts derive directly from graph topology — every sentence corresponds to exactly one `(startNode) - [r] - (endNode)` triple in Neo4j. The route does not introspect a schema registry; FactsMapper just pretty-prints whatever labels and relationship types the database returns.

The label and relationship-type names that get rendered are the canonical Eugene enums:

```
# src/foundation/model/foundational_node_enum.py
FoundationalNodeEnum:
    DRUG                = 'drug'
    DISEASE              = 'disease'
    GENE_PROTEIN         = 'gene_or_protein'
    PATHWAY              = 'pathway'
    EFFECT_PHENOTYPE     = 'effect_or_phenotype'

# src/foundation/model/foundational_relationship_enum.py
FoundationalRelationshipEnum:
    INDICATION           = 'indication'
    DRUG_DRUG            = 'drug_drug'
    DRUG_EFFECT          = 'drug_effect'
    DRUG_PROTEIN         = 'drug_protein'
    DISEASE_PROTEIN      = 'disease_protein'
    PATHWAY_PROTEIN      = 'pathway_protein'
    PROTEIN_PROTEIN      = 'protein_protein'
```

- **Label-agnostic Cypher:** the underlying query (section 2) matches any label on either side. The enums above are the *observed* values FactsMapper will see; new labels would be rendered just by replacing underscores with spaces.
- **Single-label assumption:** FactsMapper `._to_fact` calls `set(start_node.labels).pop()` — if a node carries two labels the choice is non-deterministic. In practice Eugene nodes ship with one label so this is fine, but be aware before bolting on multi-label data.
- **Relationship types from data, not enum:** the Cypher returns `type(rel)` verbatim. If the ETL writes an unknown relationship type, the sentence still renders — it just may read oddly.
- Paths only exist if upstream ETL populated the graph. An empty `ListResponse` usually means the seed/ETL for that domain has not run yet, not that the route is broken.

1. HTTP entry — router, validators, DI

[src/foundation/router/facts_router.py](#)

- Module-load DI at line 22: `facts_orchestrator = foundational_facts_orchestrator()`. Instantiated once on import; the orchestrator (and its driver-backed adapter) is reused for every request.
- Router prefix is `/graph` (line 17-20: `APIRouter(prefix="/graph", tags=["facts"])`), so the full path is `GET /graph/facts/start/{start_id}`.
- Decorator (lines 25-29): `response_model=ListResponse, response_model_exclude_none=True`.
- Path param `start_id` (lines 31-40): `Annotated[str, Path(..., max_value=512, min_length=0)]` plus an `AfterValidator(validate_id)`.
- Query param `page` (lines 41-48): `int, gt=0, required`.
- Query param `page_size` (lines 49-57): `int, gt=0, le=50, required`.
- Handler body (lines 68-75) is a three-liner: call `facts_orchestrator.find_facts_by_start_id(start_id, n_hop=1, page_size, page)`, guard against `None`, wrap the resulting `set[str]` as `ListResponse(count=len(facts), results=list(facts))`.

Validator

[src/foundation/router/validate_util.py](#)

```
def validate_id(id: str) -> str:
    invalid_chars = ['%', '$', ';', ':', '^', '*']
    is_valid = not has_invalid_char(
        invalid_chars=invalid_chars, value=id
    )
    if not is_valid:
        raise ValueError(
            f"id cannot have a special character: {invalid_chars}"
        )
    return id
```

Defense-in-depth: the Cypher uses parameterised `$start_id` so injection is not the concrete threat. The validator exists to fail fast on URL-encoded payloads, regex metacharacters, and similar garbage before the driver round-trips to Neo4j.

DI factory

[src/foundation/conf/conf.py:406-410](#)

```
def foundational_facts_orchestrator() -> FoundationalFactsOrchestrator:
    return _foundational_facts_orchestrator(
        foundational_n_hop_provider=foundational_n_hop_provider(),
        facts_mapper=facts_mapper(),
    )
```

The orchestrator composes the **same** `foundational_n_hop_provider` the `/graph/relationship/start/...` route uses with a domain-specific `FactsMapper` in place of the adjacency-list `GraphMapper`. This is the key structural point of the route: *same Cypher, different mapper*.

Response model

[src/foundation/router/model/list_response.py](#)

```
class ListResponse(BaseModel):  
    count: int  
    results: list[str]
```

Deliberately flat. Each entry is one rendered fact sentence; count is just `len(results)` — no separate total/total-pages field because the underlying n-hop adapter does not currently surface a global count.

2. Underlying Cypher — shared with the n-hop route

`src/foundation/infra/db/adapters/neo4j_foundational_n_hop_adapter.py`

The facts route does **not** have its own adapter. The orchestrator calls `foundational_n_hop_provider.find_subgraph_by_start_id_and_end_id`, which calls the same n-hop adapter that powers `/graph/relationship/start/...`. The Cypher is exactly what that route's flow guide documents — reproduced here for convenience:

```
MATCH (startNode { node_id: $start_id })-[r]-{0,N_HOP}(endNode)
UNWIND(r) AS rel
RETURN DISTINCT
    startNode(rel).node_name          AS startName,
    toStringOrNull(startNode(rel).node_id) AS startId,
    labels(startNode(rel))            AS startLabels,
    endNode(rel).node_name            AS endName,
    toStringOrNull(endNode(rel).node_id) AS endId,
    labels(endNode(rel))              AS endLabels,
    type(rel)                         AS relType
ORDER BY startName
SKIP $offset
LIMIT $limit
```

- **N_HOP is always 1** for this route — the facts router hard-codes `n_hop=1` (line ~69) when calling the orchestrator. The adapter still goes through its `_ensure_valid_n_hop_size_or_throw` guard (`MAX_SUPPORTED_HOPS=2`), which is a no-op for 1.
- **No end_id branch**: facts always wants the open neighborhood, so the orchestrator passes `end_id=None`, which makes the adapter emit `(endNode)` with no property filter.
- **Pagination** via `SKIP / LIMIT` from the router's `page / page_size`. Offset is computed by `Pagination.calculate_limit_and_offset` in `src/foundation/infra/db/util/pagination.py`.
- `MAX_RESULT_COUNT = 10_000` ceiling in `FoundationalNHopProvider`: if the `DataFrame` exceeds that, the provider raises rather than truncating. Page sizes are capped at 50 at the router so a single request cannot trip this, but cumulative caller behaviour can if the start node sits in a hub.
- **Result transformer**: `result_transformer_=neo4j.Result.to_df`. The `DataFrame` columns match `src/foundation/mapper/const.py` constants which `GraphMapper` then turns into the `Graph` that `FactsMapper` consumes.

3. Facts mapper — triples to sentences

[src/foundation/mapper/facts_mapper.py](#)

`FactsMapper.map(graph)` (line 13) is the only public entry. It builds a dict `{node_id -> Node}` from `graph.nodes`, then iterates every key of `graph.relationships` and renders one fact per outgoing `Relationship`. Output type is `set[str]` — duplicate sentences (e.g. from the undirected `-[r]-` match producing the triple in both directions) are de-duped naturally.

The template

`_to_fact(start_node, rel, end_node)` (line 34) renders, after redundancy fix-up:

```
"{StartLabel} {StartValue} has {RelType} {EndLabel}, {EndValue}"
```

Each piece is run through `_label_to_text` which replaces underscores with spaces, and through `_remove_redundant` which strips a leading word of `clause2` when it matches the trailing word of `clause1`. The start label is `.capitalize()`'d. For the triple `(:drug {Flurandrenolide})-[:drug_effect]-(:effect_or_phenotype {Pruritus})`:

```
start_label = 'drug'
relationship = 'drug effect'          # _label_to_text(rel)
fixed_rel    = _remove_redundant('drug', 'drug effect')
              = 'effect'              # 'drug' matched both sides -> drop
end_label    = 'effect or phenotype'
fixed_end    = _remove_redundant('drug effect', 'effect or phenotype')
              = 'or phenotype'       # trailing 'effect' matched leading 'effect'
# final sentence:
# 'Drug Flurandrenolide has effect or phenotype, Pruritus'
```

Aside: the redundancy fix-up is heuristic. For `drug_drug`, `_remove_redundant` has a special branch (lines 61-63) that detects `splits2[0] == splits2[1]` and rewrites the clause to relationship to `drug` so the sentence does not read like ... has drug drug drug ...

Worked example

```
# Triple coming out of the n-hop Cypher:
(:drug {node_name: 'Flurandrenolide'})
-[:drug_effect]-
(:effect_or_phenotype {node_name: 'Pruritus'})
```

```
# Sentence produced by FactsMapper._to_fact:
'Drug Flurandrenolide has drug effect, Pruritus'
```

Other shapes follow the same pattern: 'Drug Imatinib has drug protein, ABL1' from a `:drug_protein` edge, 'Disease Hemophilia A has indication, Factor VIII' from a `:indication` edge, and so on. Pathway and protein-protein triples render analogously.

Edge cases

- `graph is None` → returns empty set. The router then wraps that as `ListResponse(count=0, results=[])`.
- Node with multiple labels → `set(...).pop()` is non-deterministic; the sentence will still render but the picked label is arbitrary.
- Self-loops (the `0` in `-[r]-{0,N}` can produce a self-row from the anchor): same start and end node, treated as a regular triple. In practice these are rare in Eugene's data.
- **No ordering:** `set[str]` is unordered. The router converts to `list(facts)` for the response, but two calls with the same input may return the same strings in different orders.

4. Response model — sample JSON

The wire format is the flat `ListResponse` from `src/foundation/router/model/list_response.py`:

```
{
  "count": 4,
  "results": [
    "Drug Flurandrenolide has drug effect, Pruritus",
    "Drug Flurandrenolide has drug effect, Erythema",
    "Drug Flurandrenolide has drug protein, NR3C1",
    "Drug Flurandrenolide has indication, Atopic dermatitis"
  ]
}
```

- `count` is always `len(results)`. It is a convenience for the UI — there is no separate total-pages count.
- `results` is `list[str]`: each entry is one rendered fact. Order is not stable across calls (the underlying set is unordered).
- Empty graph → `{"count": 0, "results": []}` (the router's explicit None guard at lines 72-73).
- `response_model_exclude_none=True` is set but the model has no optional fields, so it is a no-op here — kept for consistency with the other routes.

5. Data source — where the edges come from

This route is **purely read-side**. It does not create or merge anything. The facts you see are entirely a function of what the global ETL pipelines have written into Neo4j:

- `src/ingest_drug_aliases.py` — CSV ingest of drug product names & synonyms; attaches `:drug_product` and `:drug_synonym` to canonical `:drug` nodes via `:has_drug_alias`. See [EUGENE_DRUG_ALIAS_FLOW_GUIDE.pdf](#).
- `src/download_patents.py` — USPTO / patent metadata; attaches `:patent` nodes to assignees and target drugs via `:has_patent` edges. See [EUGENE_PATENT_FLOW_GUIDE.pdf](#).
- `src/download_clinicaltrail.py` — [clinicaltrials.gov](#) records; connects `:clinical_trial` nodes to drug + indication.
- `src/download_pubmed.py` — PubMed publication ingest; attaches `:publication` nodes and citation edges.
- Seed Cypher under `bin/seed/` — creates the canonical `:drug`, `:disease`, `:gene_or_protein`, `:pathway`, `:effect_or_phenotype` nodes everything else hangs off.

If a fresh database returns `{"count": 0, "results": []}` for a known good `start_id`, the failure is almost always upstream — the seed Cypher or one of the ingest CLIs did not run for the domain you are testing.

How to confirm there is something to render

```
// In neo4j-browser:
MATCH (n { node_id: '19725' }) RETURN n, labels(n)
MATCH (n { node_id: '19725' })-[r]-(m) RETURN type(r), count(*)
// If the second query is empty the facts response will be too.
```


6. Suggested debugging walk

- **Bring the stack up:** `docker-compose up eugene_neo4j eugene_ws`. Confirm the seed Cypher has run (see section 5) so there is something to traverse.
- **Sanity-curl:** `curl -s '$EUGENE_WS/graph/facts/start/19725?page=1&page_size=10' | jq`. Expect a non-empty results array of English sentences and a count that matches `results | length`.
- **Breakpoint at the router handler:** `src/foundation/router/facts_router.py:68` (the `facts_orchestrator.find_facts_by_start_id` call). Inspect `start_id`, `page`, `page_size` after FastAPI validation and confirm the hard-coded `n_hop=1`.
- **Step into the orchestrator:**
`src/foundation/provider/foundational_facts_orchestrator.py:29` — watch it delegate to `foundational_n_hop_provider.find_subgraph_by_start_id_and_end_id`. The intermediate Graph object is the same shape the `/graph/relationship/...` route returns.
- **Step through the n-hop adapter:** breakpoint at `src/foundation/infra/db/adapter/neo4j_foundational_n_hop_adapter.py:112` (`_collect_n_hop`). Copy the assembled Cypher string + params dict into Neo4j Browser to feel what it returns; confirm the `-[r]-{0,1}` bound.
- **Inspect the mapper:** breakpoint at `src/foundation/mapper/facts_mapper.py:34` (`_to_fact`). Step into `_remove_redundant` and confirm the redundancy fix-up does what you expect for your label / rel-type combo — especially for tricky cases like `drug_drug` or `protein_protein`.

7. Reference index

Schema (informational)

```
src/foundation/model/foundational_node_enum.py      # DRUG, DISEASE, GENE_PROTEIN, PATHWAY, EFFECT
src/foundation/model/foundational_relationship_enum.py # INDICATION, DRUG_DRUG, DRUG_EFFECT, DRUG_PROTEIN
                                                    # DISEASE_PROTEIN, PATHWAY_PROTEIN, PROTEIN_PATHWAY
# NOTE: the Cypher is label-agnostic; enums describe what FactsMapper will see, not what it filters on
```

Router & wiring

```
src/foundation/router/facts_router.py               # endpoint, line 22 DI, lines 25-29 decorator
src/foundation/router/validate_util.py              # validate_id, line 98-104
src/foundation/router/model/list_response.py        # ListResponse{count, results}
src/foundation/conf/conf.py                         # foundational_facts_orchestrator factory L40
```

Orchestrator & mapper

```
src/foundation/provider/foundational_facts_orchestrator.py # find_facts_by_start_id, line 25-34
src/foundation/mapper/facts_mapper.py               # map L13, _to_fact L34, _remove_redundant L35
```

Shared n-hop machinery (same as /graph/relationship route)

```
src/foundation/provider/foundational_n_hop_provider.py # MAX_RESULT_COUNT=10_000
src/foundation/infra/db/adapters/neo4j_foundational_n_hop_adapter.py # Cypher L138-170
src/foundation/infra/db/util/pagination.py            # SKIP/LIMIT helper
src/foundation/mapper/graph_mapper.py                # DataFrame -> Graph (input to FactsMapper)
src/foundation/mapper/const.py                      # DataFrame column names
src/foundation/model/graph/graph.py                 # Graph dataclass
src/foundation/model/graph/node.py                  # Node dataclass
src/foundation/model/graph/relationship.py           # Relationship dataclass
```

ETL (for graph data)

```
src/ingest_drug_aliases.py
src/download_patents.py
src/download_clinicaltrail.py
src/download_pubmed.py
bin/seed/*.cypher # canonical node seed
```