

Eugene N-Hop Neighborhood — End-to-End Flow

Developer walkthrough: HTTP request → FastAPI router → Neo4j variable-length MATCH → DataFrame → adjacency-list Graph response

Audience

Engineers who need to understand how Eugene expands the neighborhood around a node — one or two hops out — and returns it as an adjacency-list subgraph. This is the read-side route behind the UI's *expand* action and the MCP tool that fetches relationships for a node. Every reference uses the form `path/to/file.py:line`.

Reference endpoints

- GET `/graph/relationship/start/{start_id}` — one-hop neighborhood (`n_hop` default 1).
- GET `/graph/relationship/start/{start_id}?n_hop=2` — two-hop neighborhood. Two is the hard ceiling: `n_hop` is constrained `gt=0, le=2` at the router and the adapter re-checks via `MAX_SUPPORTED_HOPS = 2`.
- GET `/graph/relationship/start/{start_id}?end_id=DB00538&n_hop=2` — bounded path search: same query, but the end node is also pinned by `node_id`.

Caveat

The Eugene graph is densely connected. The adapter comment (`neo4j_foundational_n_hop_adapter.py:17-19`) warns that a sufficient number of hops causes the entire graph to return — hence the `MAX_SUPPORTED_HOPS = 2` ceiling. Even at 2 hops, popular anchors (a common drug, a hub pathway) can blow past the provider's `MAX_RESULT_COUNT = 10_000` guard and raise. Treat this route as *local-neighborhood* expansion, not general traversal.

0. Schema (read this first)

Unlike the drug-alias or patent routes, the n-hop query is **label-agnostic**. It matches any node label and any relationship type, anchored only by a `node_id` property on the start (and optionally end) node:

```
(startNode { node_id: $start_id })-[r]-{0,N}(endNode [{ node_id: $end_id }])  
// any label on either side, any relationship type r
```

- Node labels that appear in results are whatever exists in the graph: `:drug`, `:disease`, `:gene_or_protein`, `:pathway`, `:patent`, `:clinical_trial`, `:organization`, etc. The canonical set lives in `src/foundation/model/foundational_node_enum.py`.
- Relationship types come from `src/foundation/model/foundational_relationship_enum.py` — `HAS_DRUG_ALIAS`, `HAS_INDICATION`, `HAS_TARGET`, `HAS_PATENT`, etc.
- **Important:** the n-hop Cypher does **not** reference these enums. They constrain only what callers (e.g. the UI legend, the mapper consumers) *interpret* — the database query happily returns any label/type that exists.
- Paths only exist if upstream ETL has populated the graph. An empty subgraph response usually means the seed/ETL steps for that domain have not run, not that the route is broken.
- Idempotency in Eugene is by `MERGE` on `node_id`; there are no Neo4j `CREATE CONSTRAINT` statements.

1. HTTP entry — router & wiring

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

- Module-load DI at line 21: `n_hop_provider = foundational_n_hop_provider()`. The provider (and its driver-backed adapter) is built once on import and reused for every request.
- Endpoint declared at lines 24-27: `@router.get("/relationship/start/{start_id}", response_model_exclude_none=True)`. The router prefix `/graph` is applied in the `APIRouter(prefix="/graph", tags=["foundation"])` declaration (lines 16-19).
- Path param `start_id` (lines 29-38) — annotated with `AfterValidator(validate_id)`.
- Query param `end_id` (lines 39-47) — optional, same `validate_id` `AfterValidator`.
- Query param `n_hop` (lines 48-56) — `int`, `gt=0`, `le=2`, default 1. Pydantic rejects 0 and 3+ before the handler runs.
- Handler body (line 58-60) is a one-liner:
`n_hop_provider.find_subgraph_by_start_id_and_end_id(start_id=start_id, end_id=end_id, n_hop=n_hop)`.

Validators

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

```
def validate_id(id: str) -> str:                                     # line 98
    # these are regex like characters that do not show up in our ids
    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
```

This is a defense-in-depth check — the underlying Cypher uses parameterised `$start_id` / `$end_id` so injection is not the threat. The validator exists to fail fast on obvious garbage ids (URL-encoded payloads, regex meta in logs) before the driver round-trips to Neo4j.

DI factories

[src/foundation/conf/conf.py](#)

- `neo4j_foundational_n_hop_adapter()` at lines 137-138 — pulls the shared driver via `_neo4j_driver()`.
- `_neo4j_foundational_n_hop_adapter(driver)` at lines 141-144 — pure constructor over the driver.
- `foundational_n_hop_provider()` at lines 341-345 — composes the adapter with `graph_mapper()`.

First breakpoint

Set a breakpoint on `src/foundation/router/n_hop_router.py:28` (the `async def find_n_hop` signature). Hit the endpoint with curl and inspect the bound parameters *after* validators have run — you should see `start_id`, `end_id`, `n_hop` all as clean Python values.

2. Query adapter — the variable-length Cypher

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

Public entry: `collect_by_start_id_and_end_id` (line 38). The sequence is straightforward: clamp `n_hop`, ensure the driver is alive, build the Cypher, run it, return a `DataFrame`.

```
def collect_by_start_id_and_end_id(self, start_id, page, page_size,
                                   end_id=None, n_hop=2):          # line 38
    n_hop = self._ensure_valid_n_hop_size_or_throw(n_hop)         # MAX_SUPPORTED_HOPS = 2
    ensure_connection(self.driver)
    return self._collect_n_hop_by_id(
        start_id=start_id, end_id=end_id, n_hop=n_hop,
        page=page, page_size=page_size,
    )
```

The Cypher (built around lines 138-147 + 158-170)

```
MATCH (startNode { node_id: $start_id })-[r]-{0,N}(endNode{ node_id: $end_id })
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 is interpolated**, not parameterised: `"%s" % n_hop` in `_build_n_hop_by_id_query` (line 138). Neo4j requires variable-length pattern bounds to be literals, so they cannot be bound via `$params`. The `n_hop` value is already clamped by both the FastAPI `le=2` and the adapter's `_ensure_valid_n_hop_size_or_throw` guard (line 149), so the substitution is safe.
- **Optional end_id branch**: when `end_id` is `None`, line 145 emits `(endNode)` with no property filter — i.e. *any* reachable node within N hops. When provided, the `{ node_id: $end_id }` clause pins the far end and the query becomes a bounded path search.
- **The `-[r]-{0,N}` form** walks 0 to N hops in either direction (undirected). The lower bound 0 means the start node itself is included in the result (as a self-row), which is why the mapper sees the anchor in its `uniq-ids` set even when no edges exist.
- **Pagination** via `SKIP/LIMIT` comes from `Pagination.calculate_limit_and_offset(page, page_size)` in `src/foundation/infra/db/util/pagination.py`. The router does not currently expose `page / page_size` — the provider passes `page=1, page_size=MAX_RESULT_COUNT=10000`, so in practice every call asks for the whole 10k window.
- `ensure_connection(self.driver)` (imported from `graph.util.util`) verifies the Neo4j driver can ping before issuing the query — useful for surfacing auth/network failures as a clear error instead of a hung session.
- **Result transformer**: `result_transformer=neo4j.Result.to_df` (line 118) hands the work to the Neo4j driver's built-in pandas converter. The adapter returns a `DataFrame` — no row-by-row Python loop on the hot path.

3. Mapper & response model

`src/foundation/provider/foundational_n_hop_provider.py`

The provider sits between router and adapter. After the DataFrame comes back it enforces `MAX_RESULT_COUNT = 10_000` (line 17): if the DataFrame has more rows than that, the call *raises* rather than truncating, on the theory that a 10k+ subgraph is almost certainly the wrong answer and the caller should pick a less central anchor.

```
df = self.neo4j_foundational_n_hop_adapter.collect_by_start_id_and_end_id(...)
self._ensure_result_size_or_raise(df)          # ValueError if > 10_000 rows
return self.graph_mapper.map(df=df)
```

`src/foundation/mapper/graph_mapper.py`

`GraphMapper.map()` (line 32) turns the row-shaped DataFrame into an adjacency-list Graph in two passes:

- **Unique nodes** (`_build_uniq_nodes`, line 52): union of `startId` / `endId` columns, then for each id pick the first row in which it appears and read its name + labels.
- **Relationships** (`_build_relationships`, line 96): group rows by `startId`; for each start node build a set of `Relationship(id=endId, rel=relType)`.
- DataFrame column names come from `src/foundation/mapper/const.py`: `startId`, `startName`, `startLabels`, `endId`, `endName`, `endLabels`, `relType`.

Response dataclasses

```
# src/foundation/model/graph/graph.py
@dataclass(frozen=True, unsafe_hash=True)
class Graph:
    node_count: int
    nodes: set[Node]
    relationships: dict[str, set[Relationship]]

# src/foundation/model/graph/node.py
class Node:
    id: str
    value: str          # node_name
    labels: frozenset[str] # sorted, frozen for hashability

# src/foundation/model/graph/relationship.py
class Relationship:
    id: str  # the OTHER endpoint's node_id
    rel: str  # relType, e.g. 'has_target'
```

Example JSON (FastAPI serialisation)

```
{
  "node_count": 3,
  "nodes": [
    {"id": "DB00846", "value": "Felodipine", "labels": ["drug"]},
    {"id": "P35968", "value": "VEGFR-2", "labels": ["gene_or_protein"]},
    {"id": "DB00538", "value": "Sorafenib", "labels": ["drug"]}
  ],
  "relationships": {
    "DB00846": [{"id": "P35968", "rel": "has_target"}],
    "DB00538": [{"id": "P35968", "rel": "has_target"}]
  }
}
```

Note: `response_model_exclude_none=True` on the route trims null fields. Sets are emitted as JSON arrays.

4. Data source — where the edges come from

This route is **purely read-side**. It does not create or merge anything. For a meaningful subgraph to come back, upstream ETL has to have populated nodes and `:relationship` edges. The relevant CLIs:

- `src/download_patents.py` — pulls USPTO / patent metadata and attaches `:patent` nodes to their assignees and target drugs via `HAS_PATENT` edges.
- `src/ingest_drug_aliases.py` — the alias ETL covered in *EUGENE_DRUG_ALIAS_FLOW_GUIDE*. Adds `:drug_product` and `:drug_synonym` nodes connected by `HAS_DRUG_ALIAS`.
- `src/download_clinicaltrail.py` — ingests clinicaltrials.gov records; connects `:clinical_trial` to its drug + indication.
- `src/download_pubmed.py` — PubMed publication ingest; attaches `:publication` nodes and citation edges.
- Plus the seed Cypher files under `bin/seed/` which create the canonical `:drug`, `:disease`, `:gene_or_protein`, `:pathway`, `:organization` nodes that everything else hangs off.

If a fresh database returns an empty Graph 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 anything to traverse

```
# In neo4j-browser:
MATCH (n { node_id: 'DB00846' }) RETURN n, labels(n)
MATCH (n { node_id: 'DB00846' })-[r]-(m) RETURN type(r), count(*)
// If the second query is empty, your n-hop response will be too.
```

5. Suggested debugging walk

- **Bring the stack up:** `docker-compose up eugene_neo4j eugene_ws`. Confirm the seed Cypher has run (see section 4) so there is something to traverse.
- **Sanity-curl:** `curl -s '$EUGENE_WS/graph/relationship/start/DB00846?n_hop=1' | jq`. Expect a non-empty nodes array and the start id as a key in relationships.
- **Breakpoint at the router:** `src/foundation/router/n_hop_router.py:28` (signature of `find_n_hop`). Inspect `start_id`, `end_id`, `n_hop` after FastAPI validation.
- **Validator:** drop a breakpoint at `src/foundation/router/validate_util.py:98` and try a bad id like `'DB008%46'` — you should see the `ValueError` fire before the handler runs.
- **Connection guard:** step into `neo4j_foundational_n_hop_adapter.py:47` (the `ensure_connection(self.driver)` call). If Neo4j is down or auth is wrong, this is where you will see it — not later in result mapping.
- **Inspect the assembled Cypher:** breakpoint at `_collect_n_hop` (`neo4j_foundational_n_hop_adapter.py:112`). Copy the query string + params dict into Neo4j Browser to feel what it returns. Confirm the `-[r]-{0,N}` bound matches your `n_hop` value.
- **Mapper build:** breakpoint at `src/foundation/mapper/graph_mapper.py:32` (`GraphMapper.map`). Watch `uniq_ids` grow, then the `relationships` dict. If a row from the `DataFrame` is missing from the output, it is either de-duped by set semantics or its id is null (drop site at `_build_uniq_nodes` line 59).
- **Response inspection:** confirm the final Graph's `node_count` equals `len(nodes)` and that every key in `relationships` is also present in `nodes`.

6. Reference index

Schema

```
src/foundation/model/foundational_node_enum.py      # label catalogue (informational)
src/foundation/model/foundational_relationship_enum.py # rel-type catalogue (informational)
# NOTE: the n-hop Cypher is label-agnostic; enums constrain callers, not the query
```

Router & wiring

```
src/foundation/router/n_hop_router.py              # endpoint, line 21 DI, line 28 handler
src/foundation/router/validate_util.py              # validate_id, line 98-104
src/foundation/conf/conf.py                         # factories L137-144, L341-345
```

Read path

```
src/foundation/provider/foundational_n_hop_provider.py # MAX_RESULT_COUNT=10_000, line 17
src/foundation/infra/db/adapter/neo4j_foundational_n_hop_adapter.py # MAX_SUPPORTED_HOPS=2, Cypher L
src/foundation/infra/db/util/pagination.py            # SKIP/LIMIT helper
src/foundation/mapper/graph_mapper.py                 # DataFrame -> Graph
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/download_patents.py
src/ingest_drug_aliases.py
src/download_clinicaltrail.py
src/download_pubmed.py
bin/seed/*.cypher      # canonical node seed
```