

Eugene Facet Search — End-to-End Flow

Developer walkthrough: HTTP boundary → validators → DI → Neo4j facet adapter → APOC-built JSON

Audience

Engineers onboarding to the Eugene biomedical knowledge graph who need a single, file-referenced trace of the facet endpoint. Every reference uses the form `path/to/file.py:line` so you can open it directly in your IDE and step through with a debugger.

Reference endpoint

POST `/facet/{label}` where `label` is one of `drug`, `disease`. The request body is a `FacetSearchValues` JSON object: `{ "values": [ "Lupus", "rituximab" ] }`.

Sample call (Disease facets filtered by name substring)

```
curl -X POST http://localhost:8080/facet/disease \
  -H 'Content-Type: application/json' \
  -d '{"values": ["Lupus"]}'
```

Sample call (Drug facets, no filter)

```
curl -X POST http://localhost:8080/facet/drug \
  -H 'Content-Type: application/json' \
  -d '{"values": []}'
```

Declared in `src/foundation/router/facet_router.py:27-52`. Only `drug` and `disease` are accepted today (the adapter raises on any other label).

How to read this guide

- Sections follow the request path top-down: HTTP → validator → DI factory → adapter → Cypher → JSON.
- Section 0 explains what a "facet" means in Eugene — a per-label aggregate of related node-name frequencies, built with `apoc.coll.frequenciesAsMap`.
- Section 4 lists a concrete debugging walk (curl, breakpoints, Cypher verification).
- Watch for the SQL-injection FIXME the adapter author left in `neo4j_foundational_facet_adapter.py:48-50` — it is load-bearing for future hardening.

0. What a "facet" means in Eugene

In the UI, a *facet* is the left-rail filter widget that shows "Lupus (12), Rheumatoid Arthritis (8), ..." next to a drug list. The facet router serves the data behind that widget: for every node with a given label (e.g. :drug), it walks one hop to chosen sibling labels and returns frequency maps keyed by node_name.

Two flavours, hard-coded in the adapter

- `FoundationalNodeEnum.DRUG` → returns frequency maps for indications, contraindications, offLabelUses (each a disease-name → count map).
- `FoundationalNodeEnum.DISEASE` → returns frequency maps for drugs and geneProteins.
- Any other label raises `ValueError("Found {label} but facet only supports drug or disease!")` from `neo4j_foundational_facet_adapter.py:52-56`.

Relevant enum values

[src/foundation/model/foundational_node_enum.py](#)

```
class FoundationalNodeEnum(Enum):
    DISEASE = 7, "DISEASE".lower() # graph label :disease
    DRUG     = 8, "DRUG".lower()   # graph label :drug
    GENE_PROTEIN = 12, "gene_protein"
```

The enum's second tuple slot is the Neo4j label string. `str_to_label_enum(value)` in `src/foundation/router/validate_util.py:107-111` reverses that mapping (string → enum).

Relationship types touched by the Cypher

[src/foundation/model/foundational_relationship_enum.py](#)

- `:indication` — drug treats disease.
- `:contraindication` — drug should not be used in disease.
- `:`off-label use`` — note the backticks: the label contains a hyphen and a space, so Cypher must quote it.
- For the disease flavour, the adapter does NOT pin a relationship type (`[(node) - [] - (x:drug) | properties(x)]`) — any edge between a `:disease` and a `:drug` contributes.

Implication for debugging

Keep the two enum files open while reading the adapter; every Cypher string interpolates label strings from them. Also keep `src/graph/util/node_util.py` open — `normalize_node_label` is applied to the label before interpolation (lowercase, strip dots, spaces → underscores).

1. HTTP entry — the FastAPI router

Wiring

src/eugene_ws.py:45 — from foundation.router import facet_router as foundation_facet_router. src/eugene_ws.py:67 appends foundation_facet_router to the routers list, and src/eugene_ws.py:77-78 loops through and calls app.include_router(router.router).

Router file

src/foundation/router/facet_router.py

Imports (lines 1-14):

```
from fastapi import APIRouter, Path
from pydantic import AfterValidator
from foundation.conf.conf import neo4j_foundational_facet_adapter
from foundation.router.validate_util import (
    str_to_label_enum,
    validate_label,
)
from foundation.router.model.facet_search_values import FacetSearchValues
from foundation.router.model.facet_label import FacetLabel
```

Router declaration (lines 19-22):

```
router = APIRouter(
    prefix="/facet",
    tags=["facet"],
)
```

Module-load dependency injection (line 24)

```
foundational_facet_adapter = neo4j_foundational_facet_adapter()
```

This is Eugene's manual DI pattern: a module-scope call constructs the adapter once at import-time. The factory lives in src/foundation/conf/conf.py:157-164:

```
def neo4j_foundational_facet_adapter() -> Neo4jFoundationalFacetAdapter:
    return _neo4j_foundational_facet_adapter(driver=_neo4j_driver())

def _neo4j_foundational_facet_adapter(
    driver: Driver,
) -> Neo4jFoundationalFacetAdapter:
    return Neo4jFoundationalFacetAdapter(driver=driver)
```

_neo4j_driver() elsewhere in conf.py returns a singleton neo4j.Driver configured from env vars (NEO4J_URI, NEO4J_USERNAME, NEO4J_PASSWORD). There is no framework, no decorators — module import order is the DI graph.

Endpoint handler (lines 27-52)

```
@router.post(
   ("/{label}",
    response_model_exclude_none=True,
)
async def collect_facets_by_label_and_values(
    label: Annotated[
        FacetLabel,
        Path(
            description="The node type to list. e.g. drug, disease",
            max_length=64, min_length=0,
            examples=["drug, disease"],

```

```

        ),
        AfterValidator(validate_label),
    ],
    facet_search_values: FacetSearchValues,
):
    label_enum = str_to_label_enum(label.value[1])
    values = (
        facet_search_values.values if facet_search_values.values is not None else []
    )
    json_blob = foundational_facet_adapter.collect_facet_by_label(
        label=label_enum, values=values
    )
    logger.debug(f"facet: {json_blob}")
    return json_blob

```

Two validation layers

- **Path enum constraint** — `FacetLabel`

(`src/foundation/router/model/facet_label.py`) is a `str`, `Enum` with only `disease` and `drug` members. FastAPI rejects anything else with HTTP 422 before the handler runs.

- **AfterValidator** — `validate_label` in

`src/foundation/router/validate_util.py:13-21` double-checks membership in the broader `Label` enum and returns the `FoundationalNodeEnum`. Its return is discarded by the handler (it then re-derives the enum via `str_to_label_enum`), but the side-effect (raise on invalid) is the point.

- **Body validation** — Pydantic auto-validates `FacetSearchValues` (`List[str]`). There is *no* `validate_facet_search_values` call in this router, even though `validate_util.py:24-32` defines one that rejects special characters. **This is a known gap** — raw values flow straight into the Cypher regex (see Section 2 FIXME).

Set your first breakpoint

At `facet_router.py:48` (`json_blob = foundational_facet_adapter.collect_facet_by_label(...)`). Inspect:

- `label` — the raw `FacetLabel` instance.
- `label.value[1]` — should be `"drug"` or `"disease"`.
- `label_enum` — the `FoundationalNodeEnum` member.
- `values` — the list of name fragments to filter by.

2. Query adapter — the Cypher

`src/foundation/infra/db/adapter/neo4j_foundational_facet_adapter.py`

Public entry point (lines 23-27)

```
def collect_facet_by_label(
    self, label: FoundationalNodeEnum, values: list[str] = []
) -> str | None:
    ensure_connection(self.driver)
    return self._collect_facet_by_label(label=label, values=values)
```

`ensure_connection` from `src/graph/util/util.py` pings the driver before each request — a defence against stale connections after Neo4j restarts.

Inner execution (lines 29-43)

```
def _collect_facet_by_label(self, label, values) -> str | None:
    query = self._build_facet_by_label_query(label=label, values=values)
    logger.info(f"facet: {query}")
    try:
        records = self.driver.execute_query(
            query=query,
            result_transformer=neo4j.Result.single,
        )
        return records["json"]
    except (DriverError, Neo4jError) as exception:
        logging.error(...)
        raise exception
```

- `Result.single` expects exactly one record — the Cypher ends with `RETURN { facets: { ... } }` as json, which always produces one row.
- The return type is annotated `str | None` but in practice the driver returns a Python dict — FastAPI then serialises it as JSON.
- Note the missing parameter binding: no *params* are passed to `execute_query`. The Cypher is built with string interpolation. See the FIXME at lines 48-50.

Cypher generator (lines 45-122) — structure

Built in three concatenated chunks:

```
facet_count_query = "\n".join([
    match_clause,          # MATCH (node:<label>)
    optional_where_clause, # WHERE node.node_name =~ '(?i).*<v>.*' OR ...
    facet_count_clause,    # the APOC frequency aggregation
])
```

Generated Cypher for /facet/drug with values=[]

```
MATCH (node:`drug`)

WITH apoc.map.merge(properties(node), {
    nodeId: id(node),
    indications: [(node)-[:indication]-(x:disease) | properties(x)],
    contraindications: [(node)-[:contraindication]-(x:disease) | properties(x)],
    offLabelUses: [(node)-[:`off-label use`]-(x:disease) | properties(x)]
}) as node_order_by_node_title
WITH collect(node) as nodes,
[x in apoc.coll.flatten(collect(node.indications)) | x.node_name] as indications,
[x in apoc.coll.flatten(collect(node.contraindications)) | x.node_name] as contraindications,
[x in apoc.coll.flatten(collect(node.offLabelUses)) | x.node_name] as offLabelUses,
[x in apoc.coll.flatten(collect(node.indicationCount)) | x.node_name] as indicationCount,
```

```

    [x in apoc.coll.flatten(collect(node.contraindicationCount)) | x.node_name] as contraindicationCount,
    [x in apoc.coll.flatten(collect(node.offLabelCount)) | x] as offLabelCount
WITH nodes,
    apoc.coll.frequenciesAsMap(indications) as indications,
    apoc.coll.frequenciesAsMap(contraindications) as contraindications,
    apoc.coll.frequenciesAsMap(offLabelUses) as offLabelUses
RETURN {
    facets: {
        indications: indications,
        contraindications: contraindications,
        offLabelUses: offLabelUses
    }
} as json

```

Generated Cypher for /facet/disease with values=["Lupus"]

```

MATCH (node:`disease`)
where node.node_name =~ '(?i).*Lupus.*'

WITH apoc.map.merge(properties(node), {
    nodeId: id(node),
    drugs: [(node)-[]-(x:drug) | properties(x)],
    geneProteins: [(node)-[]-(x:gene_protein) | properties(x)]
}) as node order by node.title
WITH collect(node) as nodes,
    [x in apoc.coll.flatten(collect(node.drugs)) | x.node_name] as drugs,
    [x in apoc.coll.flatten(collect(node.geneProteins)) | x.node_name] as geneProteins,
    [x in apoc.coll.flatten(collect(node.drugCount)) | x] as drugCount,
    [x in apoc.coll.flatten(collect(node.geneProteinCount)) | x.node_name] as geneProteinCount
WITH nodes,
    apoc.coll.frequenciesAsMap(drugs) as drugs,
    apoc.coll.frequenciesAsMap(geneProteins) as geneProteins
RETURN {
    facets: {
        drugs: drugs,
        geneProteins: geneProteins
    }
} as json

```

Step-by-step what the Cypher does

- **MATCH** all nodes carrying the requested label (:drug or :disease). The label string is normalized through `normalize_node_label` (`src/graph/util/node_util.py:4-9`) before backtick-quoting.
- **Optional WHERE** — if `values` is non-empty, one `node.node_name =~ '(?i).*<value>.*'` clause per value, joined with `or`. Case-insensitive substring match.
- **Per-row map merge** — for every matched node, `apoc.map.merge` attaches three (drug) or two (disease) lists of related node properties, gathered by inline `[(node)-[:rel]-(x:label) | properties(x)]` pattern comprehensions.
- **Flatten & pluck** — `apoc.coll.flatten(collect(...))` concatenates all the per-node lists into one big list, and `| x.node_name` extracts just the names.
- **Frequency map** — `apoc.coll.frequenciesAsMap` converts `["Lupus", "Lupus", "RA"]` into `{ "Lupus": 2, "RA": 1 }`. This is the facet count the UI renders.
- **RETURN** — one Cypher map literal aliased as `json`; the row count is always 1 because of the surrounding `WITH ... collect(...)` aggregations.

The injection hazard the adapter flags

[neo4j_foundational_facet_adapter.py:48-50](#)

```
# FIXME: cypher injection issue
# WARNING: this is wrong to inject a user query into the cypher
# however the api does not let me pass a parameter into a query when
# using a regex like this?
```

- User input flows raw into `node.node_name =~ '(?i).*'` — a ' or } in the value would break out of the regex literal.
- The router does not call `validate_facet_search_values` from `validate_util.py:24-32`, which would reject the special chars `% _ $ ; : ^ *`. Until that is wired in, treat the endpoint as un-sanitised.
- The fix is parameterised regex: `node.node_name =~ '(?i).*' + $value + '.*'` — Cypher allows string concatenation in regex operands, contrary to the FIXME's claim.

3. Response shape — what the caller sees

Unlike most Eugene routers (patent, pubmed, etc.) there is **no Pydantic response model and no mapper**. The Cypher returns a Map; the Python driver hands it back as a dict; FastAPI serialises that dict directly. The absence of a typed response is intentional — the keys of the frequency maps are user data (disease names, drug names) and cannot be enumerated at schema time.

Example response — POST /facet/drug, values=[]

```
{
  "facets": {
    "indications": {
      "Systemic Lupus Erythematosus": 12,
      "Rheumatoid Arthritis": 8,
      "Multiple Sclerosis": 5
    },
    "contraindications": {
      "Pregnancy": 21,
      "Hepatic Impairment": 14
    },
    "offLabelUses": {
      "Idiopathic Thrombocytopenic Purpura": 3
    }
  }
}
```

Example response — POST /facet/disease, values=["Lupus"]

```
{
  "facets": {
    "drugs": {
      "rituximab": 4,
      "hydroxychloroquine": 7,
      "belimumab": 2
    },
    "geneProteins": {
      "IFNAR1": 3,
      "TNFSF13B": 2
    }
  }
}
```

Request models — the only typed surface

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

```
from enum import Enum
```

```
class FacetLabel(str, Enum):
    disease = "disease"
    drug    = "drug"
```

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

```
from typing import Annotated
from pydantic import BaseModel, Field
```

```
class FacetSearchValues(BaseModel):
    values: list[Annotated[str, Field(None, examples=["Lupus"])]]
```

- `values` is a required field (no default). To get unfiltered facets, send `{ "values": [] }`, not `{}`.
- The handler defensively guards `facet_search_values.values` is not `None` at `facet_router.py:45-47`, but Pydantic would already have rejected a `null` (the type is


```
list[str], not list[str] | None).
```

4. Suggested debugging walk

4.1 Start the API

```
# from repo root
uvicorn src.eugene_ws:app --reload --port 8080
```

On import, `src/foundation/router/facet_router.py:24` runs `neo4j_foundational_facet_adapter()`. If Neo4j is not reachable, this is where you'll see the connection error — not when the first request hits the endpoint.

4.2 Issue a request

```
curl -s -X POST http://localhost:8080/facet/disease \
  -H 'Content-Type: application/json' \
  -d '{"values": ["Lupus"]}' | jq .
```

4.3 Recommended breakpoints

- `src/foundation/router/facet_router.py:44` — inspect `label` and `facet_search_values.values`.
- `src/foundation/router/facet_router.py:48` — about to hand off to the adapter; verify `label_enum` is the right `FoundationalNodeEnum` member.
- `src/foundation/infra/db/adapter/neo4j_foundational_facet_adapter.py:33` — a `logger.info(f"facet: {query}")` prints the fully interpolated Cypher. Copy this into Neo4j Browser if the response looks wrong.
- `neo4j_foundational_facet_adapter.py:35-38` — the actual `driver.execute_query` call; step over to see the raw records dict.

4.4 Tail logs for the generated Cypher

```
# In another terminal
tail -f logs/eugene_ws.log | grep -E '^facet: |cypher response'
```

```
# The adapter logs the Cypher at INFO and the response at DEBUG.
# Set LOG_LEVEL=DEBUG in your env to see both.
```

4.5 Verify directly against Neo4j

Paste the logged Cypher into Neo4j Browser (or `cypher-shell`):

```
cypher-shell -u neo4j -p password \
  "MATCH (node:`disease`) WHERE node.node_name =~ '(?i).*Lupus.*' \
  RETURN node.node_name, node.node_id LIMIT 10;"
```

If this returns zero rows, your facet response will be `{ "facets": { "drugs": {}, "geneProteins": {} } }` — an empty result, not an error.

4.6 Sanity-check the relationship coverage

```
MATCH (d:drug)-[r:indication|contraindication|`off-label use`]-(:disease)
RETURN type(r) AS rel, count(*) AS n
ORDER BY n DESC;
```

If one of these rel types returns 0 rows, the facet response will have an empty bucket for that key — a hint the underlying ETL never created those edges (check the drug ingest job and the `foundational_relationship_enum.py` entries).

4.7 Common failure modes

- **422 Unprocessable Entity** — you sent `/facet/gene_protein` or any label outside the `FacetLabel` enum. The route never reaches the handler.
- **500 with "Found ... but facet only supports drug or disease"** — impossible via HTTP today (the path enum blocks it), but possible from a unit test that calls the adapter directly with another `FoundationalNodeEnum` member.
- **Neo4j syntax error** — almost always a special char in values that broke the regex literal. See the FIXME at `neo4j_foundational_facet_adapter.py:48-50`. Sanitise on the client until that's fixed.
- **Empty facets buckets** — the substring matched no nodes, or the matched nodes have no neighbouring edges of the expected type. Run the verification Cyphers in 4.5 / 4.6.

5. Reference index (file paths)

Router & request models

```
src/foundation/router/facet_router.py
src/foundation/router/model/facet_label.py
src/foundation/router/model/facet_search_values.py
src/foundation/router/validate_util.py
```

Wiring

```
src/eugene_ws.py                (lines 45, 67, 77-78)
src/foundation/conf/conf.py      (lines 157-164)
```

Adapter (Cypher generation + execution)

```
src/foundation/infra/db/adapter/neo4j_foundational_facet_adapter.py
src/graph/util/util.py          (ensure_connection)
src/graph/util/node_util.py     (normalize_node_label)
```

Schema enums

```
src/foundation/model/foundational_node_enum.py
src/foundation/model/foundational_relationship_enum.py
```

Related routes that share the validator helpers

```
src/foundation/router/count_router.py
src/foundation/router/node_details_router.py
src/foundation/router/label_router.py
```

Key takeaways

- The facet endpoint is intentionally narrow: only drug and disease labels, two hard-coded aggregation shapes.
- There is no mapper, no Pydantic response model — the Cypher's `RETURN { facets: {...} }` as json is the schema, and the dict travels straight from Neo4j to the wire.
- The `=~` regex filter is the only injection-prone surface in this route. Validate values on the client until the FIXME at `neo4j_foundational_facet_adapter.py:48-50` is resolved.
- APOC procedures (`apoc.coll.frequenciesAsMap`, `apoc.coll.flatten`, `apoc.map.merge`) are required — ensure your Neo4j image includes the APOC jar (see `containers/eugene_neo4j_ce/conf/apoc.conf`).