Neo4J Demo

```
Mix.install([
  {:boltx, "~> 0.0.6"},
  {:sparql, "~> 0.3"},
  {:kino, "~> 0.10.0"},
])
```

Introduction

To get started, create the following docker compose file:

```
version: '3'
services:
  neo4j:
    environment:
      - NEO4J_AUTH=neo4j/neo4jtest
    image: neo4j:5.26.1
    ports:
      - '7474:7474'
      - '7473:7473'
      - '7687:7687'
    volumes:
      - /vol/neo4j:/data
```

Assuming that you have named the file `neo4j-docker-compose.yml` and have docker compose installed, then run:

```
docker compose -f neo4j-docker-compose.yml up
```

Note: If you have an old version installed, then you may have to run `docker-compose ...` instead of `docker compose ...`. As the interface is in flux, you might also encounter some warnings.

This should make the Neo4j web interface available and allow you to run this livebook.

Convenience

Function for illustrating data as a graph:

```
result_as_tree = fn connections ->
  contents =
    List.foldl(connections, "graph LR;\n", fn {src, dst}, acc ->
      acc <> "  #{src}-->#{dst};\n"
```

```
    end)

  Kino.Mermaid.new(contents)
end
```

Function for illustrating a Neo4j response as a table:

```
result_as_table = fn response ->
  fields = response.fields
  results = response.results

  lines =
    results
    |> List.foldl("", fn result, acc ->
      main =
        fields
        |> Enum.map(fn field ->
          case Map.get(result, field) do
            %{id: id} -> "Node id #{id}"
            res -> "#{res}"
          end
        end)
        |> Enum.join(" | ")

      acc <> "| #{main} |\n"
    end)

  Kino.Markdown.new("""
  | #{fields |> Enum.join(" | ")} |
  | #{fields |> Enum.map(fn _ -> "--" end) |> Enum.join(" | ")} |
  #{lines}
  """)
end
```

Conversion from sensor type to modality:

```
type2modality = %{
  "TemperatureSensor" => "Temperature",
  "HumiditySensor" => "RelativeHumidity",
  "PIR" => "Occupancy"
}
```

Setup

Establish connection:

```

```

```
opts = [
  hostname: "127.0.0.1",
  scheme: "bolt",
  port: 7687,
  auth: [username: "neo4j", password: "neo4jtest"],
  pool_size: 15,
]

{:ok, conn} = Boltx.start_link(opts)
```

Test ping:

```
Boltx.query!(conn, "RETURN 1 as n") |> Boltx.Response.first()
```

Model Construction

Establish clean slate:

```
queries = [
  "MATCH ()-[r]->() DELETE r",
  "MATCH (n) DELETE n"
]

Enum.map(queries, fn query -> Boltx.query!(conn, query) end)
```

Create Type Tree

```
typetree = {
  "BaseType",
  [
    {
      "Location",
      [
        {"Building", []},
        {"Floor", []},
        {"Room", []}
      ]
    },
    {
      "Modality",
      [
        {"Temperature", []},
        {"RelativeHumidity", []},
        {"AbsoluteHumidity", []},
        {"Occupancy", []}
      ]
```

```
    },
    {
      "Point",
      [
        {
          "Sensor",
          [
            {"TemperatureSensor", []},
            {"HumiditySensor", []},
            {"PIR", []}
          ]
        }
      ]
    },
    {
      "Unit",
      [
        {"Unitless", []},
        {"Kelvin", []},
        {"DegreesCelcius", []},
        {"DegreesFahrenheit", []},
        {"Percentage", []}
      ]
    },
    {
      "Data",
      []
    }
  ]
}
```

```
defmodule TypeTree do
  def establish(subtree, conn, parent \\ nil)

  def establish({name, subtypes}, conn, parent) do
    :ok = establish(name, conn, parent)

    success =
      Enum.map(subtypes, fn subtype -> establish(subtype, conn, name) end)
      |> Enum.all?(fn result -> result == :ok end)

    if success do
      :ok
    else
      :error
    end
  end

  def establish(name, conn, parent) when is_binary(name) do
    query =
      case parent do
```

```
        nil ->
          "CREATE (:Type {name: '#{name}'})"

        _ ->
          "MATCH (parent:Type {name: '#{parent}'}) MERGE (:Type {name: '#
{name}'})-[:subtypeof]->(parent)"
      end

    Boltx.query!(conn, query)
    :ok
  end
end
```

```
TypeTree.establish(typetree, conn)
```

Create Floors

```
floors = [
  "3rd"
]
```

```
floors
|> Enum.map(fn floor ->
  query = "MATCH (t:Type {name: 'Floor'}) CREATE (:Floor {name: '#
{floor}'})-[:type]->(t)"
  Boltx.query!(conn, query)
end)
```

Create Rooms

```
rooms = [
  %{floor: "3rd", area: "17 m²"},
  %{floor: "3rd", area: "23 m²"}
]
```

```
rooms
|> Enum.map(fn %{floor: floor, area: area} ->
  query = """
    MATCH (t:Type {name: 'Room'})
    MATCH (f:Floor {name: '#{floor}'})
    MERGE (r:Room {area: '#{area}'})-[:type]->(t)
    MERGE (f)-[:contains]->(r)
```

```
  """

  Boltx.query!(conn, query)
end)
```

```
Boltx.query!(conn, "MATCH (a:Room), (b:Room) WHERE a<>b CREATE (a)-
[:adjacent]->(b)")
```

Create Sensors

```
sensors = [
  %{room: "17 m²", type: "TemperatureSensor", unit: "DegreesCelcius", hist:
12, live: "17c5ae15"},
  %{room: "17 m²", type: "HumiditySensor", unit: "Percentage", hist: 13,
live: "65d0be73"},
  %{room: "17 m²", type: "PIR", unit: "Unitless", hist: 24, live:
"5a3573c3"},
  %{room: "23 m²", type: "TemperatureSensor", unit: "Kelvin", hist: 37,
live: "b0bc97af"},
  %{room: "23 m²", type: "HumiditySensor", unit: "Percentage", hist: 22,
live: "06ef2490"},
  %{room: "23 m²", type: "PIR", unit: "Unitless", hist: 28, live:
"92d17015"}
]
```

```
sensors
|> Enum.map(fn sensor ->
  modality = Map.get(type2modality, sensor.type)

  query = """
    MATCH
      (t:Type {name: '#{sensor.type}'}),
      (u:Type {name: '#{sensor.unit}'}),
      (d:Type {name: 'Data'}),
      (m:Type {name: '#{modality}'}),
      (r:Room {area: '#{sensor.room}'})
    CREATE (tmp:#{sensor.type})-[:type]->(t)
    CREATE (tmp)-[:unit]->(u)
    CREATE (tmp)-[:data]->(:Data {hist: '#{sensor.hist}', live: '#
{sensor.live}'})-[:type]->(d)
    CREATE (tmp)-[:provides]->(modality:#{modality})-[:type]->(m)
    CREATE (r)-[:modality]->(modality)
  """

  Boltx.query!(conn, query)
end)
```

Queries

Room sizes:

```
Boltx.query!(conn, "MATCH (room:Room) RETURN room.area AS area")
|> result_as_table.()
```

Per-floor combinations of temperature and relative humidity data:

```
q = """
MATCH
    (r)<-[:contains]-(f),
    (r)-[:type]->(:Type {name: 'Room'}),
    (f)-[:type]->(:Type {name: 'Floor'}),
    (r)-[:modality]->(rhum_modality)<-[:provides]-(hum),
    (rhum_modality)-[:type]->(:Type {name: 'RelativeHumidity'}),
    (r)-[:modality]->(temp_modality)<-[:provides]-(temp),
    (temp_modality)-[:type]->(:Type {name: 'Temperature'})
RETURN
    f.name AS floor, hum, temp
"""

response =
  Boltx.query!(conn, q)
  |> result_as_table.()
```

Type tree:

```
response =
  Boltx.query!(conn, "MATCH (t:Type)-[:subtypeof]->(parent:Type) RETURN
t.name, parent.name")

response.results
|> Enum.map(fn %{"t.name" => name, "parent.name" => parent} -> {name,
parent} end)
|> result_as_tree.()
```

Final Words

The boltx package, used here, implements several versions of the bolt protocol needed to communicate with Neo4j. Breaking changes were introduced to this protocol in version 5. So, there might still be a few minor issues left.

```
q = """
MATCH
(sensor:TemperatureSensor),
(sensor)-[:unit]->(:Type {name: 'Kelvin'}),
(sensor)-[:provides]->(modality:Temperature)<- [:modality]-(room)


RETURN
    sensor,room
"""


response =
  Boltx.query!(conn, q)
  |> result_as_table.()
```

Here is how we use a query in neo4j, we first get all temperatureSensors contained inside the graph, then we grab the sensors only using kelvin as their unit,

after this we go from this sensor, to modalities through the provides relation ship from here we identify the room based on the modality relationship tp the room and then display room.

```
q = """
MATCH
  (building:Building),
  (floor:Floor)

RETURN floor, building

"""


response =
  Boltx.query!(conn, q)
  |> result_as_table.()
```

```
q = """
CREATE (:Type {name: 'Actuator'})

"""


response =
  Boltx.query!(conn, q)
  |> result_as_table.()
```

```
q = """
CREATE (:hvadController {room: 'roomB'})


"""


response =
  Boltx.query!(conn, q)
  |> result_as_table.()
```

```
q = """
MATCH
(room:Room) - [:modality] -> (modality:Thermostat),
(actuator:TemperatureActuator) - [:provides] -> (modality)


RETURN room,modality,actuator
"""


response =
  Boltx.query!(conn, q)
  |> result_as_table.()
```

```
q = """
MATCH (n)
WHERE ID(n) = 41
DETACH DELETE n

"""


response =
  Boltx.query!(conn, q)
  |> result_as_table.()
```