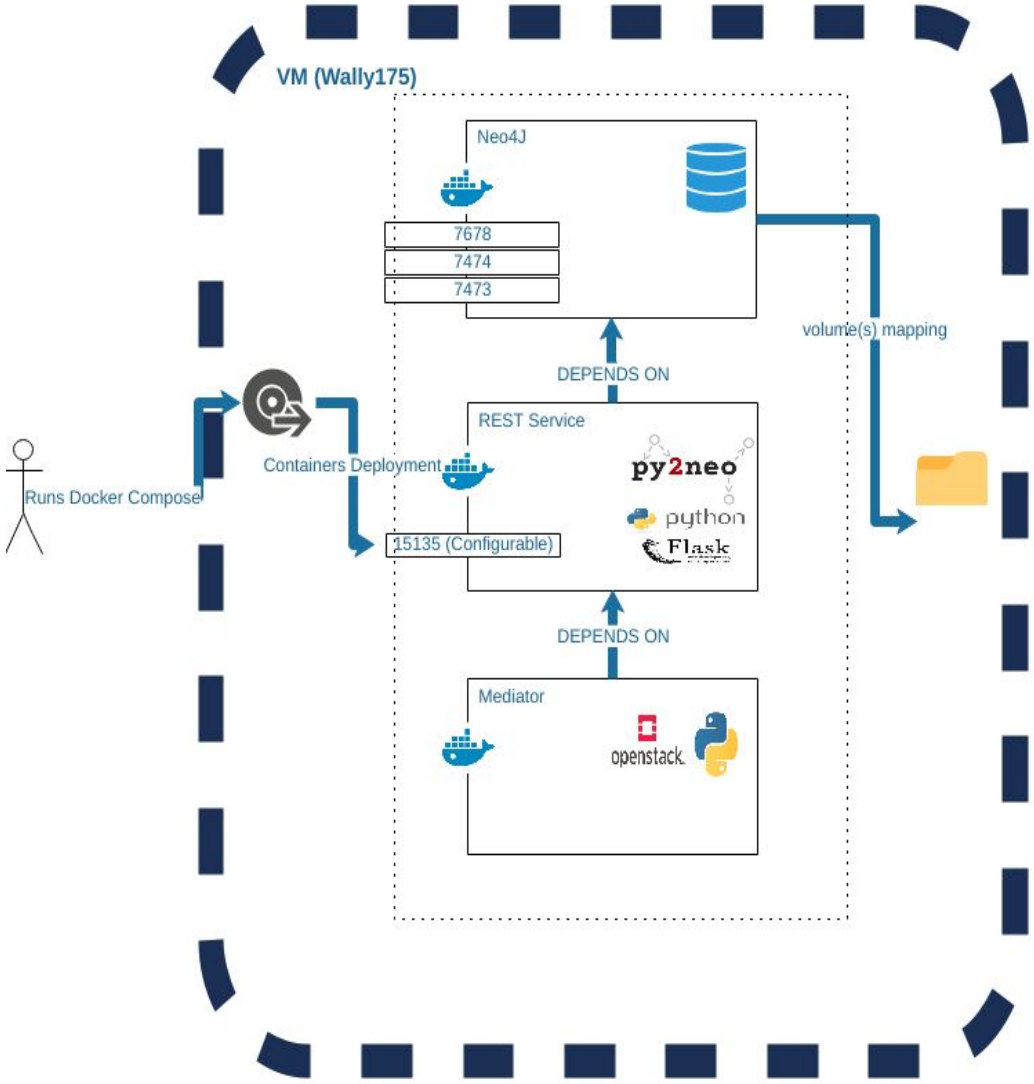
Storing OpenStack Infrastructure Information in Graph Database--Technical Documentation

System Architecture:



Components:

- **Neo4J Graph Database Container**

  It contains Neo4J database community edition of latest version. The data folder is mapped to local folder of the containing (Virtual) Machine (Wally175 in our case), so that the data can be retained in case the container crashes. The ports require to interact with the database are mapped to the host system.

- **REST Service**

  It exposes REST service that interacts with the Neo4J database to the client(s). It consists of two modules:

  - **Graph service API:**

    Responsible for CRUD (Create, Read, Update, Delete) operation on Neo4J Graph database. It uses Py2neo API for such operations.

    Py2neo:

    Py2neo is a well-maintained high-level API for Python to interact with Neo4J graph database. It provides abstraction to interact with the graph database with or without using Cypher query language.

    For more information about Py2neo API, please see their official documentation: https://py2neo.org

  - **Graph service resource:**

    Exposes REST endpoints to enable the users to use Graph service API independently from platform they use.

    It uses Flask API to expose different methods over HTTP. By default, it is configured with port **15135**. However, it can be configured with another port via an environment variable.

- **Mediator**

  It is a client module of the REST service. It queries and fetches data from OpenStack and sends it to the REST service to write it on the graph database. It contains three Python modules:

  - **Graph service interface schema (Optional)**

    It provides common schema for a client and the graph service API. It allows user to create objects of node, node attributes, relationship, and relationship attributes with the properties that are required by Neo4J. This is an optional module. Its use can be avoided if the required properties are provided (e.g. name).

  - **OpenStack-REST service Mediator**

    It uses Openstack Querier module to get the OpenStack's data, subscribes for OpenStack event(s), creates a format using graph service interface schema, and calls the REST service.

  - **OpenStack Querier**

    It uses OpenStack APIs (Nova, Neutron, Glance, Cinder, etc) for Python to query OpenStack resources and oslo_messaging API to subscribe for OpenStack's events. It subscribes to several events that could occur in

OpenStack in order to immediately update the representation in the graph database.

Deployment:

Three docker containers are deployed using docker-compose file. The three containers are:

- *Neo4J*
- *REST service*
- *Mediator*

*Deployment instructions are given in the **README** file associated with the github repository.*

Graph Data Structure:

A ***dependency graph***, which is a directed graph representing dependencies of several objects towards each other, is created as a result of running the end-to-end workflow.

Openstack API for Python doesn't explicitly give information about OpenStack components dependencies. To avoid this limitation, we defined a configuration file in which we can define OpenStack components and their dependencies along with attributes that point to the dependent component instances.

An example of the graph is:

<TODO: Insert Image>

The above given graph is based on the following configurations:

```
{
 "SERVERS": {
   "name_attr": "name",
   "id_keys": ["id"],
   "data": [],
   "RELATIONSHIPS": [
    {
      "source_attr_name": "OS-EXT-AZ:availability_zone",
      "is_source_attr_name_regex": false,
      "target_node_type": "AVAILABILITY_ZONES",
      "target_node_attr_name": "name",
      "is_target_attr_name_regex": false,
```

```
      "target_value_data_type": "string",
      "relationship_name": "DEPENDS_ON",
      "relationship_attrs": {
        "STATUS": "ACTIVE"
      }
    },
    {
      "source_attr_name": "OS-EXT-SRV-ATTR:host",
      "is_source_attr_name_regex": false,
      "target_node_type": "HYPERVISORS",
      "target_node_attr_name": "service___host",
      "is_target_attr_name_regex": false,
      "target_value_data_type": "string",
      "relationship_name": "RUNS_ON",
      "relationship_attrs": {
        "STATUS": "ACTIVE"
      }
    },
    {
      "source_attr_name": "key_name",
      "is_source_attr_name_regex": false,
      "target_node_type": "KEY_PAIRS",
      "target_node_attr_name": "_info___keypair___name",
      "is_target_attr_name_regex": false,
      "target_value_data_type": "string",
      "relationship_name": "USES",
      "relationship_attrs": {
        "STATUS": "ACTIVE"
      }
    },
    {
      "source_attr_name": "flavor___id",
      "is_source_attr_name_regex": false,
      "target_node_type": "FLAVORS",
      "target_node_attr_name": "id",
      "is_target_attr_name_regex": false,
      "target_value_data_type": "string",
      "relationship_name": "USES",
      "relationship_attrs": {
        "STATUS": "ACTIVE"
      }
    }
  ]
 }
}
```

The system (mediator component) subscribes to several events that are likely to occur in OpenStack. Upon occurrence of those events, the graph is updated to reflect changes made in the OpenStack.
If no event occurred in the time interval of 10 minutes, the mediator component will check for the changes to see if any unsubscribed event has made changes to the components of OpenStack that are shown in the graph.