Data Mapping

The rust module will be responsible for querying from different datasources and transforming the data into a schema defined in a JSON config file. The JSON config file will be responsible to storing the configuration on where and how to fetch and data and what the resulting object should look like.

Example data mapping JSON file

```
{
    "attributes": [
        {
          "connection": "postgres",
          "query": "select * from users where id = '__PID__'",
          "expected_rows": "single",
          "select_attributes": {
            "fn": ["Type::String", "!ConvertName::firstname", "!Audit"],
            "ln": ["Type::String", "!ConvertName::lastname", "!Audit"],
            "currency": ["Type::String"],
            "age": ["Type::Number"]
          }
        },
        {
          "connection": "mysql",
          "query": "select * from org where user_id = '__PID__'",
          "expected_rows": "multiple",
          "select_attributes": {
            "manager": ["Type::String", "!ConvertName::managers"]
          }
        }
      ],
        "roles": [
          {
            "connection": "customers_db",
            "query": "select product from products where id =
'__PID__''",
            "entity": "ProductRole",
            "select_attribute": ["ReturnAttribute::product"]
          }
        ]
      }
```

- attributes - Holds mapping on which attribute to get from which connection. The end result will be a map of attributes
  - connection - name of the connection, this should map to the connection pool name for reference
  - query - raw query to be executed on the connection. Any values that are in `__PID__` must be substituted by a dynamic value before executing the query at runtime.
  - select_attributes - attributes to be selected from query.
    - select attributes is a object which key as the attribute name returned from query and value as a array list holding properties.
    - Select attribute properties will define characteristics of the attribute can have simple functions "attached" to it for transformation. Type name and value will be separated with `::` eg. `Type::String` There will be 2 types of attribute properties, static and dynamic. `Type` will be considered as static since every attribute will required it. Dynamic ones will be start with `!` and should map to some function for dynamic processing/calculation.
      - Type - define the type of the attribute
      - ConvertName - dynamic property which defines the name of the attribute in result object
  - expected_results
    - Single - 0 or 1 rows is expected
    - Multiple - 0, 1 or N rows are expected. Resulting object will be a array list
- Roles - This has similar format as attributes but the end result will be a list of Entity objects instead of a map.
  - Roles can have 0, 1 or N number of rows. If no rows are returned, returned a empty list.

    - Use the `entity` attributes in roles as type in the resulting object.

Result object

```
{
    "uid": {
        "type": "User",
        "id": "dipen"
    },
    "attrs": {
        "firstname": "Dipen",
        "lastname": "Javia",
        "currency": "USD",
        "age": 30,
        "managers": ["john", "mike"]
    },
    "parents": [
        {
            "type": "ProductRole",
            "id": "role1"
        },
        {
            "type": "ProductRole",
            "id": "role1"
        }
    ]
}
```

Attribute Properties

Name	Type	Notes
Type	Static	type of the attribute. Expected types • String • Number • Boolean • JSON

ConvertName	Dynamic	Convert the name of the attribute in the resulting object
Audit	Dynamic	Call a specific function passing in the context (eg. attribute value)
ReturnAttribute	Static	Used when evaluating roles.

Types of connections to be supported:

- Postgres
- Mysql
- MS SQL Server
- MongoDB
- DynamoDB

Each of these connections should support querying over non-encrypted and TLS/SSL.

General Requirements

1. Rust 1.7 or later must be used for the solution
2. Usage of serde_json to serialize and deserialize
3. Usage of sqlx connection pooling to maintain connections of each object.
4. Code must use async behavior for concurrency using tokio runtime
5. Usage of 3rd party client drivers for the database connectors
6. Usage of individual error message enums for proper error handling
7. Code should be modularized to support more connectors in future
8. Code should be split into different modules for an ease of maintenance
9. Appropriate structs and abstraction layer should be built so that the parsing of the rows is database agnostic and not each connector implements it's own way to parse data.