Team#5 ASSIGNMENT# 1

Database Team based Assignment

Ishwarya Varadarajan 011549473
Mridula Krishnamurthy 010830209
Sowmya Viswanathan 011432668
Vishal Praanesh 011485175

SQLITE

PURCHASE ORDER MANAGEMENT SYSTEM:

The system stores the purchase order in a table that is created when a buyer(company) orders for commodities from its various vendors

The system has three tables namely PURCHASE_ORDR, VENDOR & PRODUCT.

PURCHASE_ORDR contains the details of the purchase order details that is created every time an order is placed to a vendor

VENDOR table contains the details of every vendor the company does business with
PRODUCT table contains the details of the products for which orders are placed

VIEW PROD_PRICE contains the columns such as UNIT_COST, ORDER_QTY to calculate and update total price of commodities based on the PURCHASE_ORDR tables

The above functionality is done automatically when a new row is inserted in the table PURCHASE_ORDR with the help of a TRIGGER TRG_PO which will update the PURCHASE_ORDR table with the TOTAL_COST with (UNIT_COST * ORDER_QTY) from PROD_PRICE VIEW

To create a TABLE named PURCHASE_ORDR:

```
CREATE TABLE PURCHASE_ORDR (

PO_NUMBER INT PRIMARY KEY NOT NULL,

ITEM_NUMBER INT NOT NULL,
ITEM_DESC CHAR (20) NOT NULL,

VENDOR_NUMBER INT NOT NULL,
```

ORDER_DATE TEXT NOT NULL,

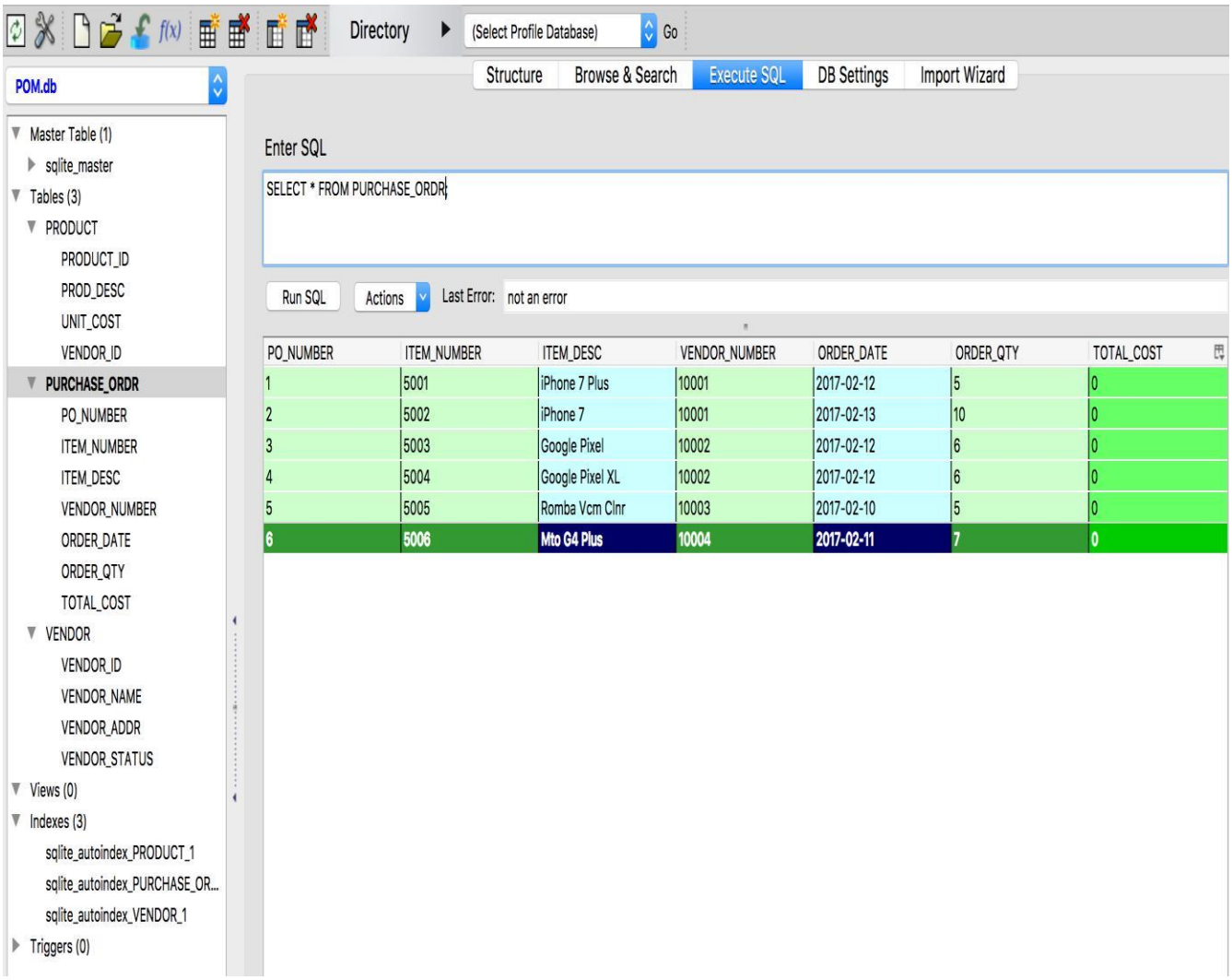
ORDER_QTY INT NOT NULL,

TOTAL_COST REAL);

To insert data into the table

INSERT INTO PURCHASE_ORDR (PO_NUMBER, ITEM_NUMBER, ITEM_DESC, VENDOR_NUMBER, ORDER_DATE, ORDER_QTY, TOTAL_COST)

VALUES (1, 5007, 'X's Digital Clock', 10005, '2017-02-17', 100, '0');



To create a TABLE called VENDOR:

CREATE TABLE VENDOR (

VENDOR_ID INT NOT NULL,

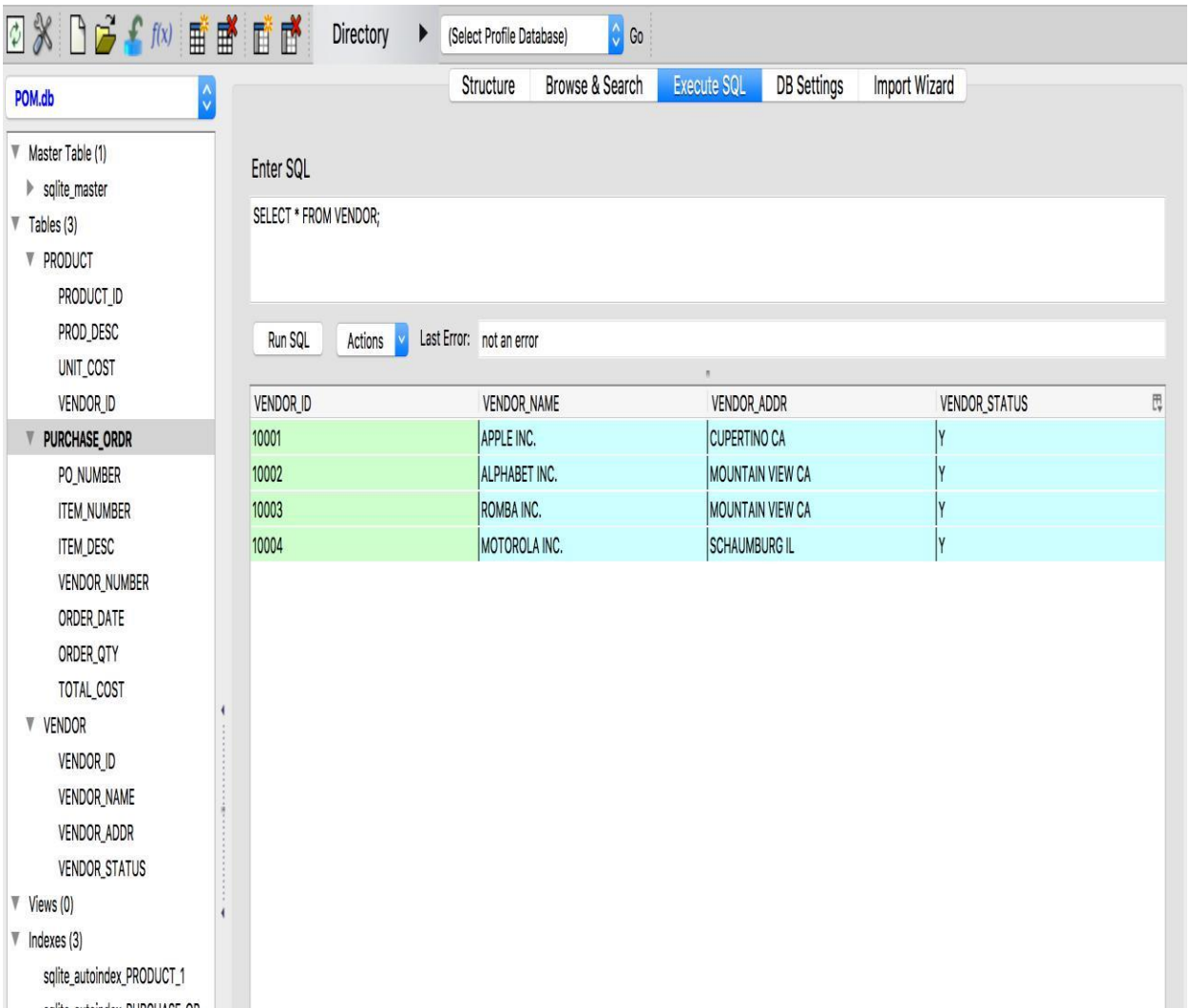
VENDOR_NAME CHAR (30) NOT NULL,

VENDOR_ADDR CHAR (30) NOT NULL,

VENDOR_STATUS CHAR (1) NOT NULL);

To insert data into the table

INSERT INTO VENDOR (VENDOR_ID, VENDOR_NAME, VENDOR_ADDR, VENDOR_STATUS) VALUES (10001, 'APPLE INC.', 'CUPERTINO CA', 'Y');



To create a TABLE called PRODUCT:

CREATE TABLE PRODUCT (

PRODUCT_ID INT NOT NULL,

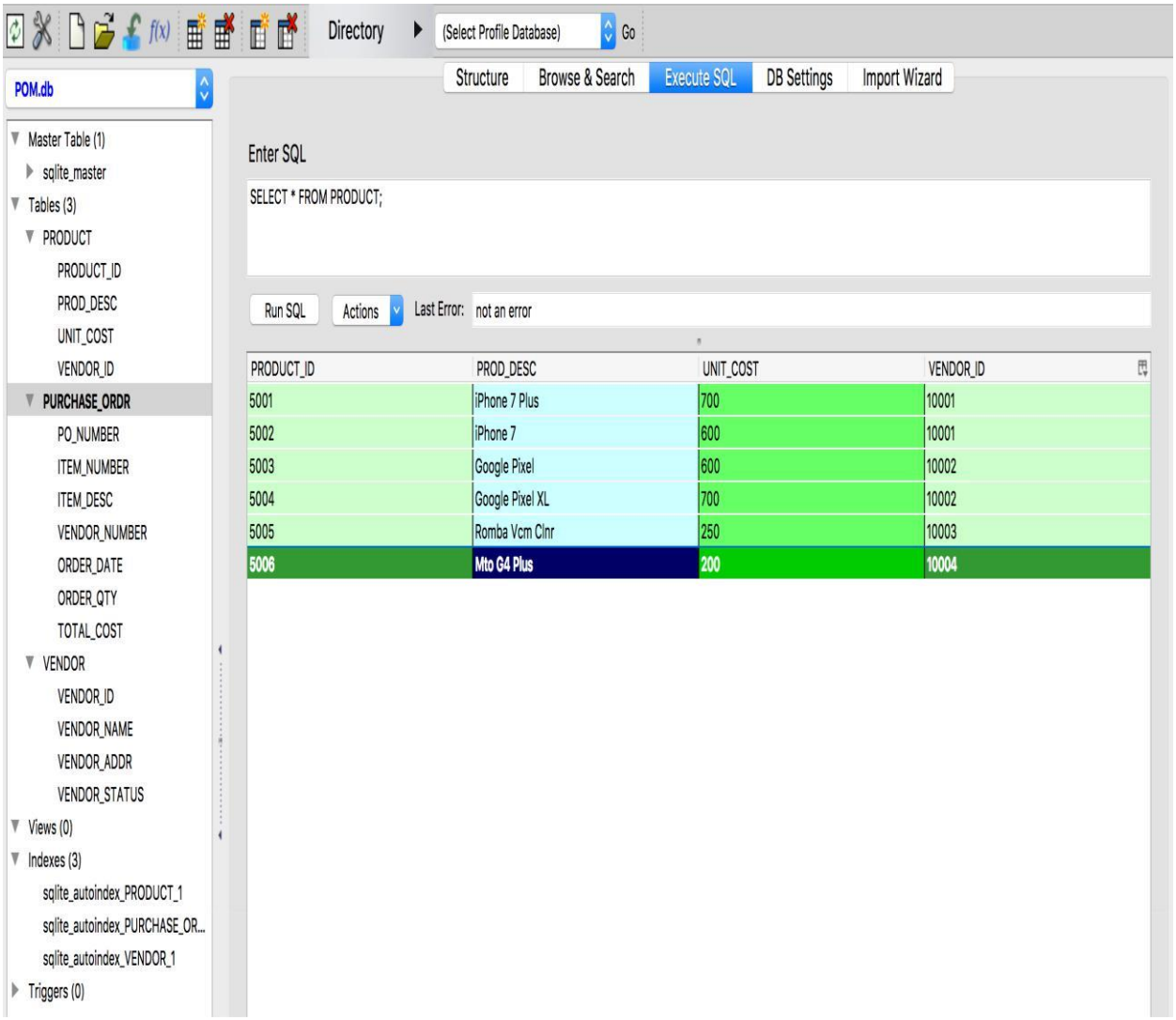
PROD_DESC CHAR (30) NOT NULL,

UNIT_COST REAL NOT NULL,

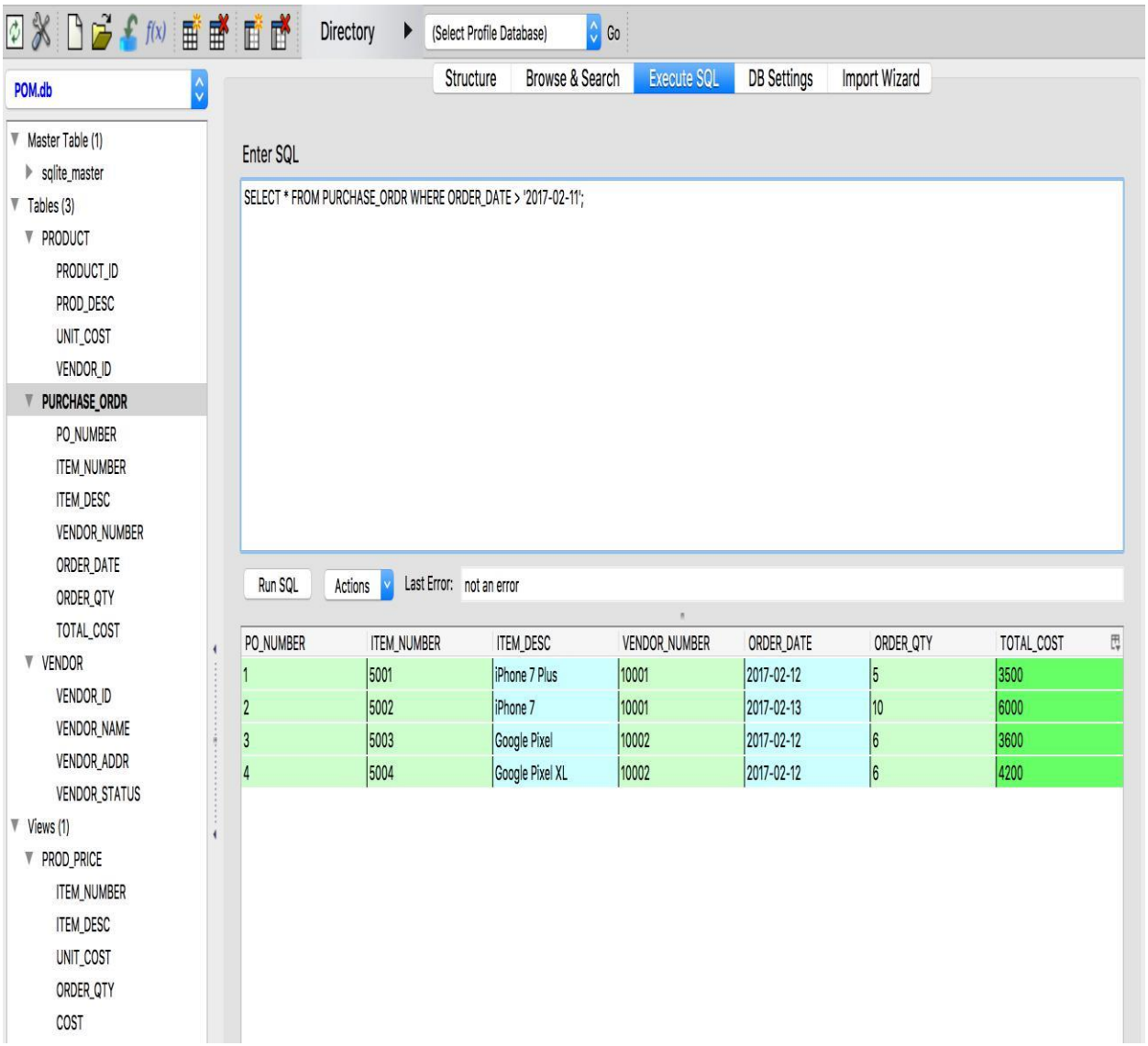
VENDOR_ID INT NOT NULL);

To insert data into the table

INSERT INTO PRODUCT (PRODUCT_ID, PROD_DESC, UNIT_COST, VENDOR_ID) VALUES (5001, 'iPhone 7 Plus', 700.00, 10001);

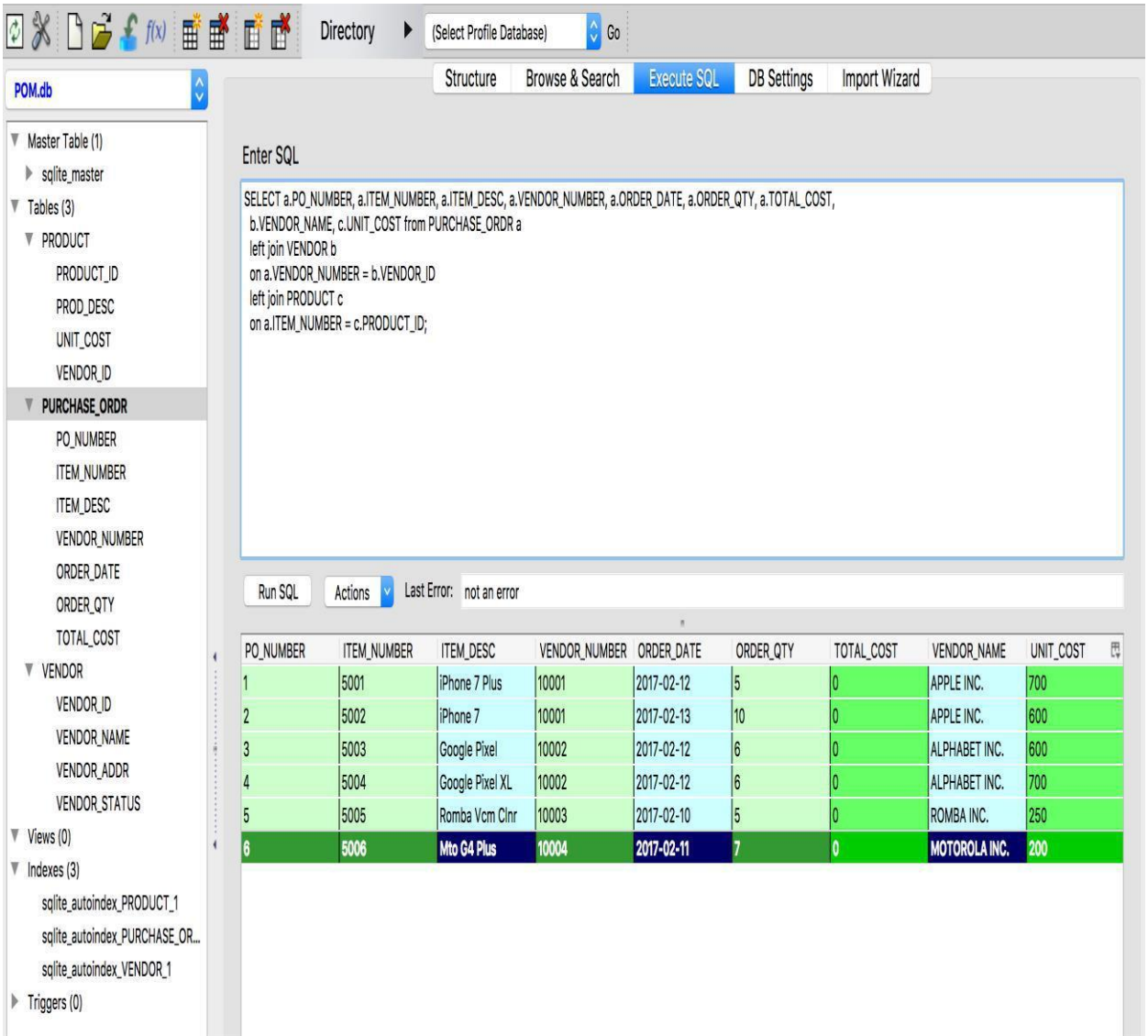


SELECT QUERY WITH A CONDITION

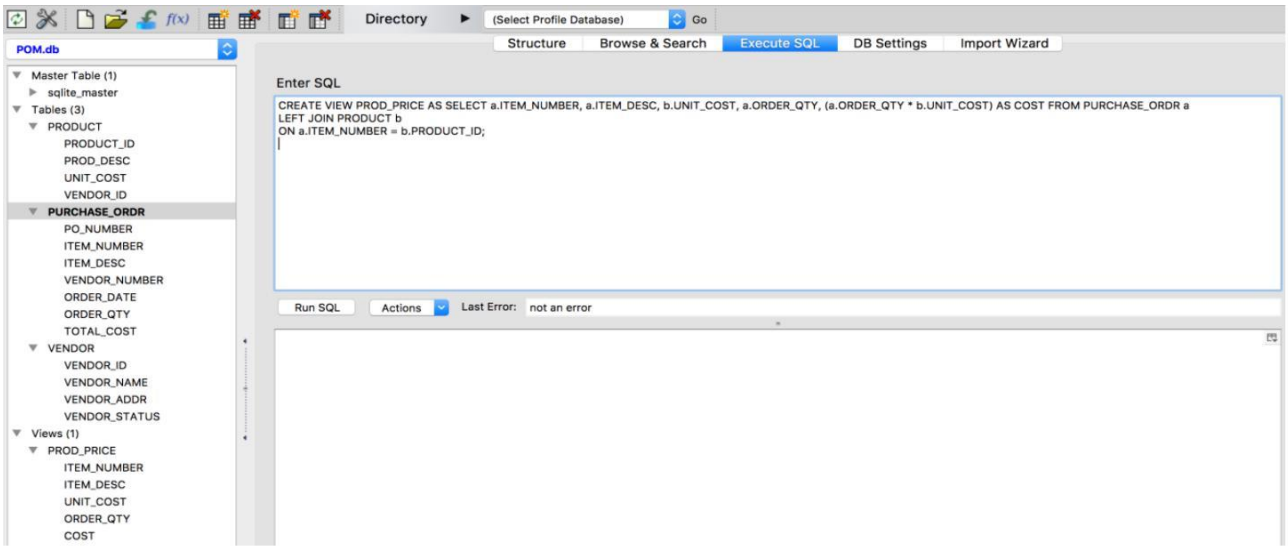


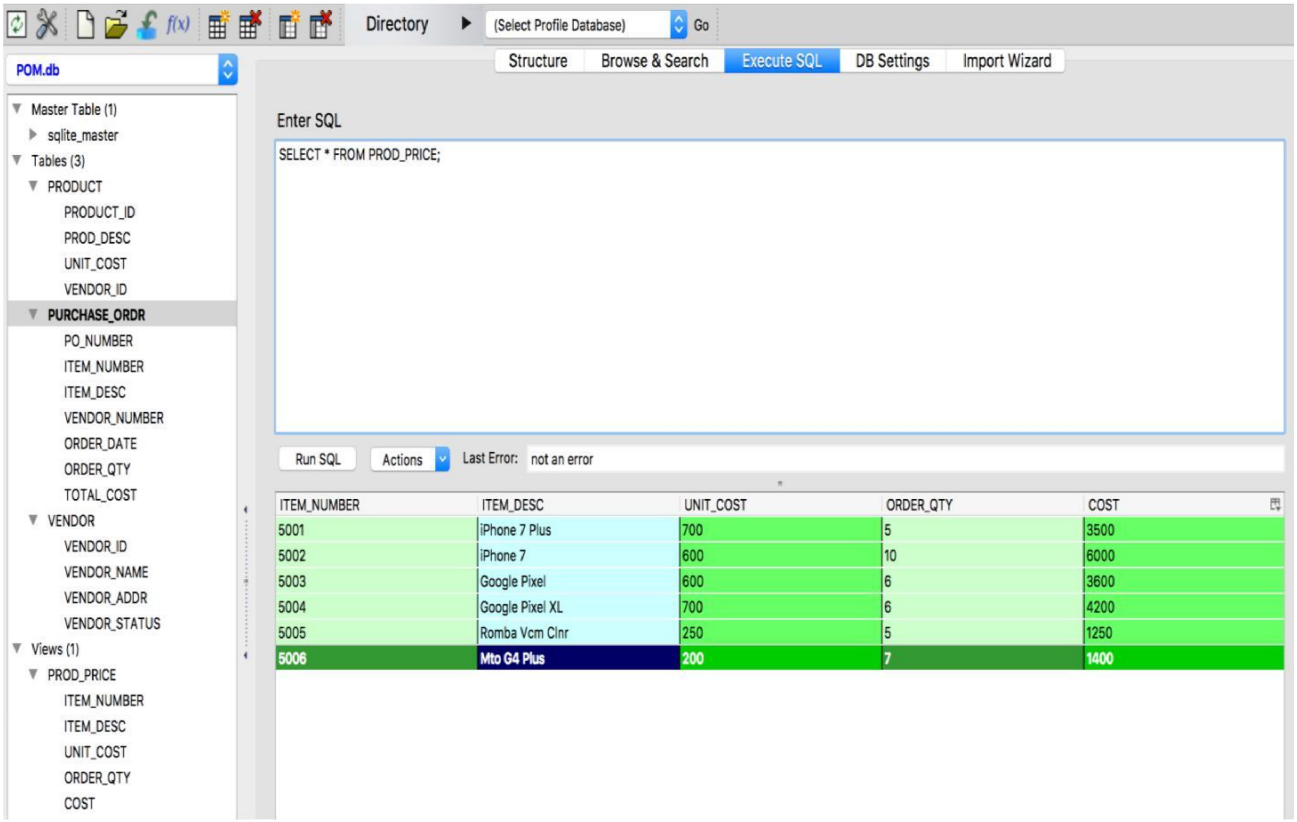
A query to LEFT JOIN a table with two or more tables

Table PURCHASE_ORDR is joined completely with tables VENDOR & PRODUCT based on PRODUCT_ID. Matching records based on the condition are selected for the view.



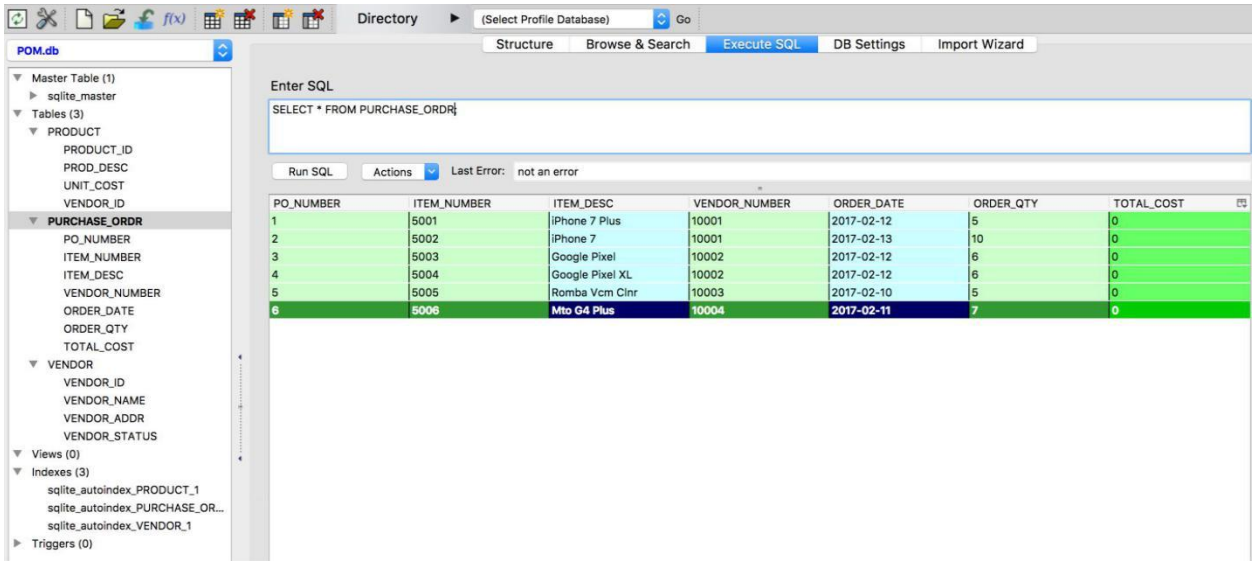
To create a VIEW named PROD_PRICE



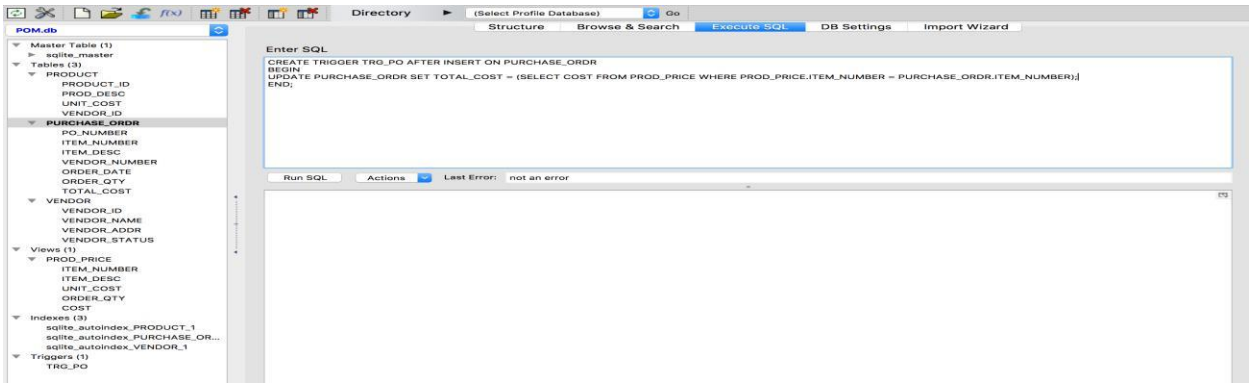


To UPDATE TABLE PURCHASE_ORDR

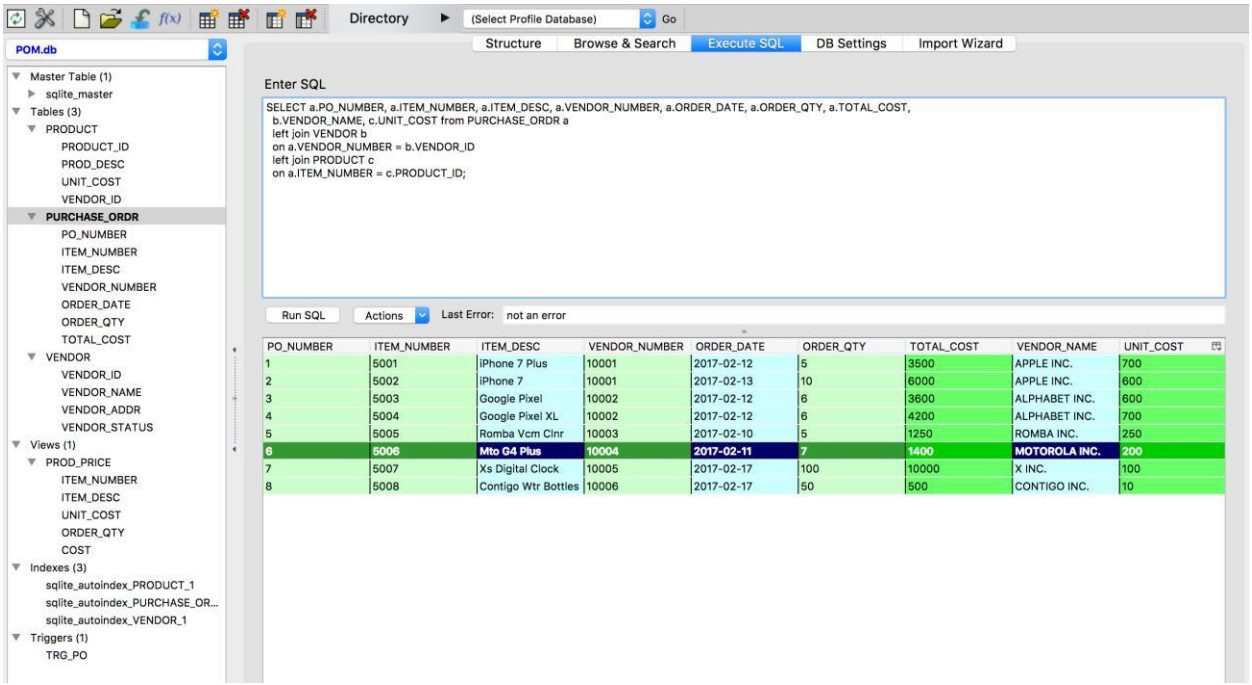
UPDATE PURCHASE_ORDR SET TOTAL_COST = (SELECT COST FROM PROD_PRICE WHERE PROD_PRICE.ITEM_NUMBER = PURCHASE_ORDR.ITEM_NUMBER);



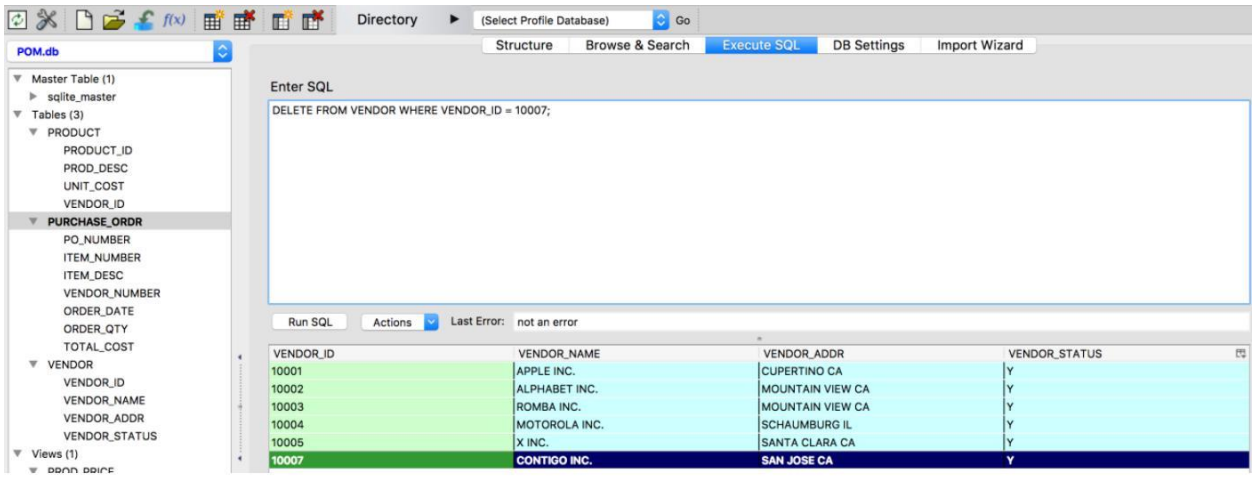
To create a trigger named TRG_PO

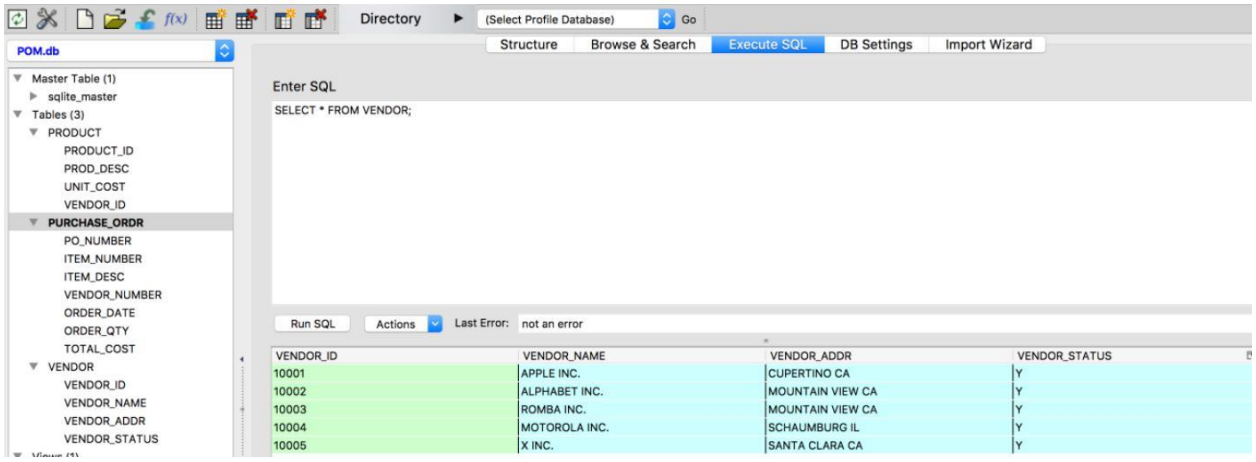


Automatically updates the PURCHASE_ORD table's column by calculating the TOTAL_COST as per the LOGIC set inside the BEGIN and END statements in the TRIGGER query.



To delete a row from a table

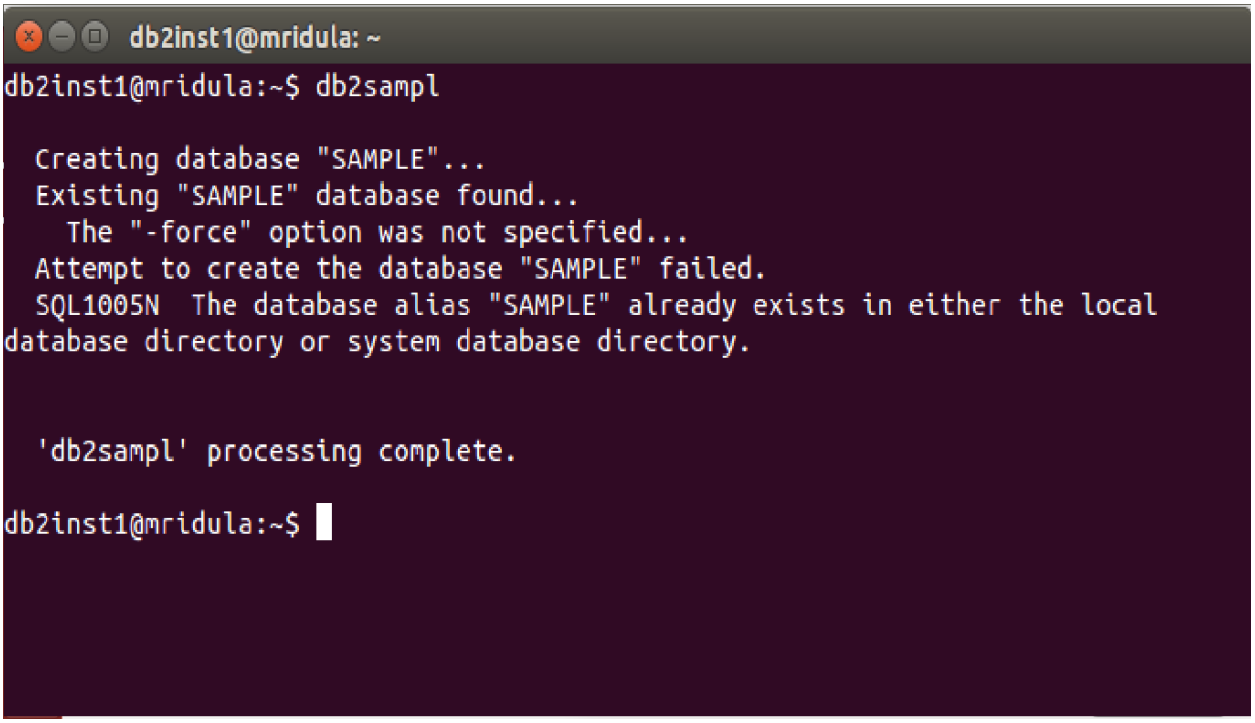




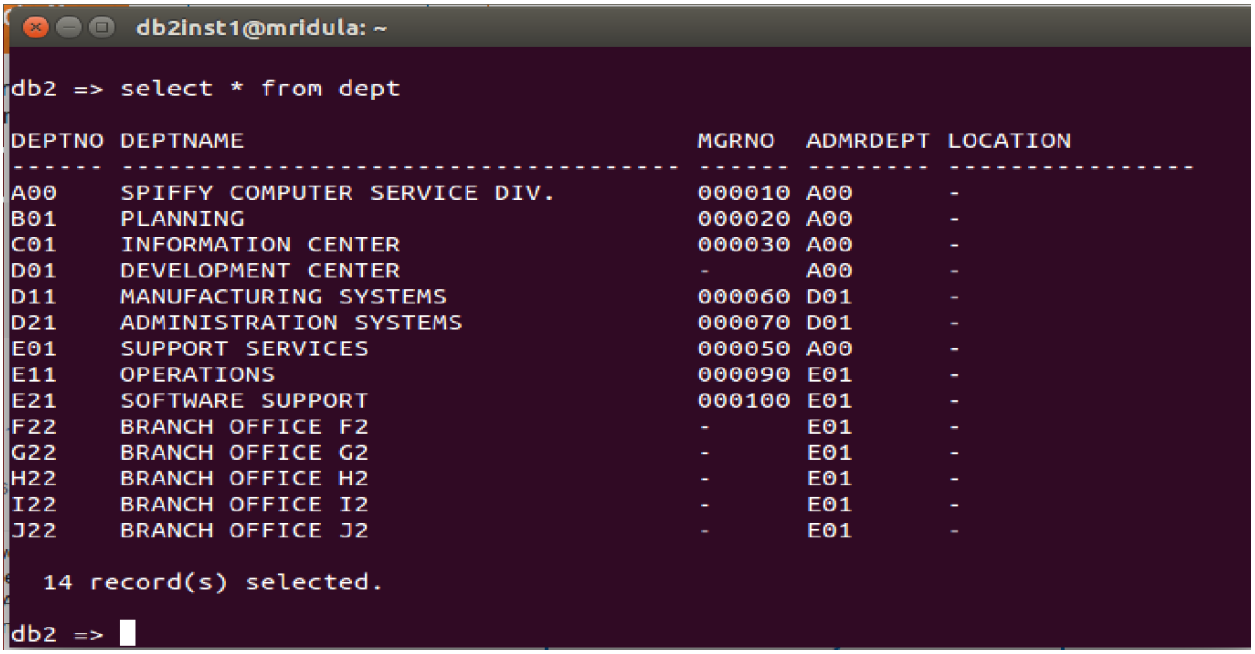
DB2 Express C

In this part of the assignment, we create a Sample database using ‘db2sampl’ command, which creates a database called ‘SAMPLE’, and adds some tables to it along with values filling the tables. Then, we run a sample query for one of these tables using ‘where’ clause and ‘group by’. Then, we generate the query explain plan using the ‘db2exfmt’ tool.

Creating Sample database using ‘db2sampl’



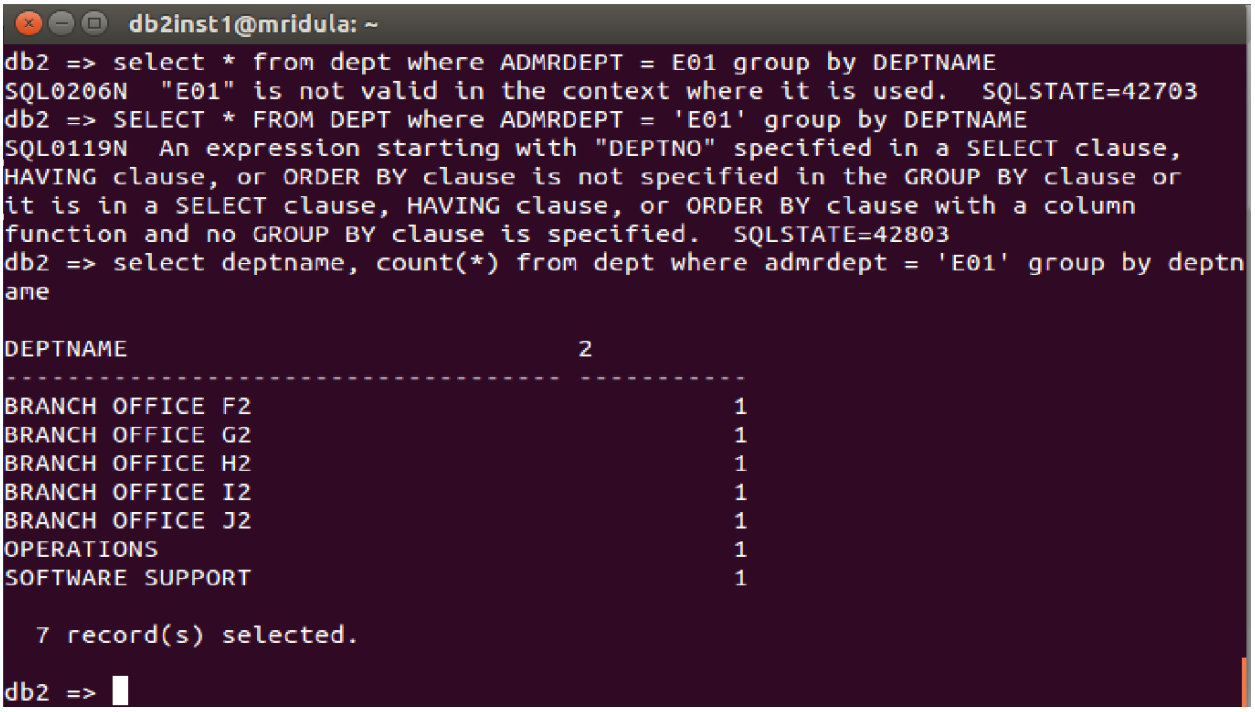
We are querying the ‘dept’ table within the database, which has the following rows and columns.



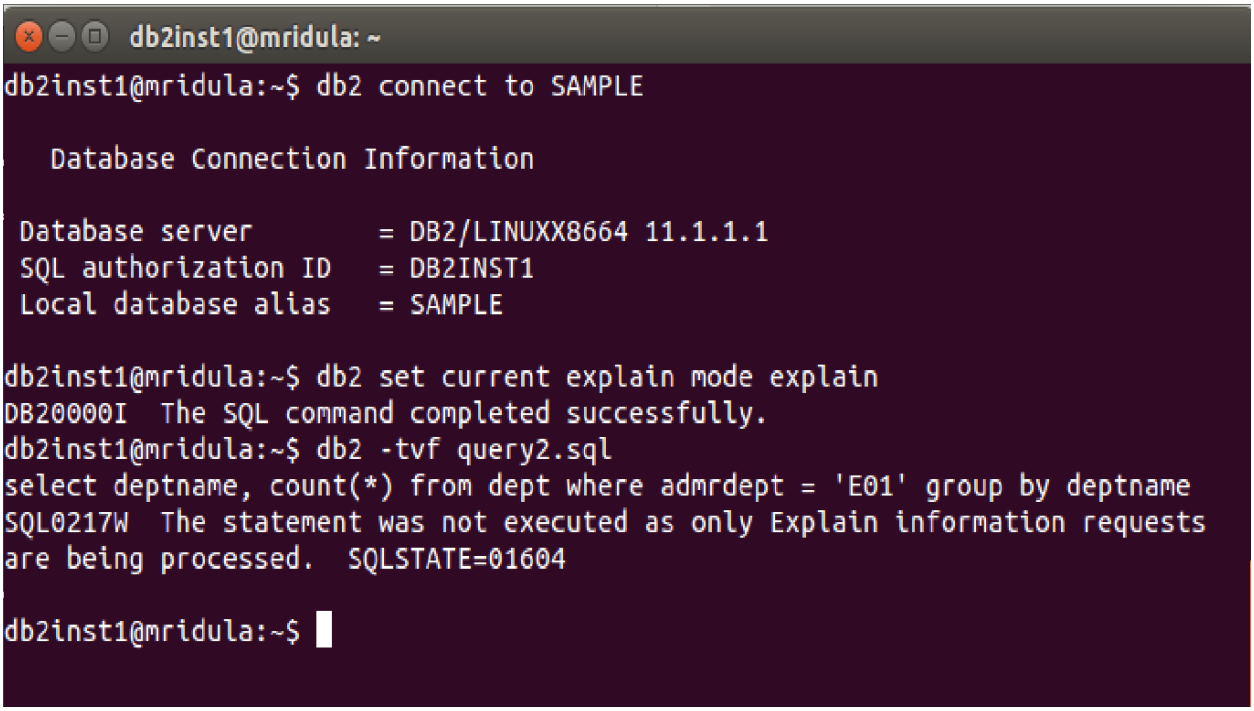
The query we are using is

“select deptname, count(*) from dept where admrdept = ‘E01’ group by deptname”.

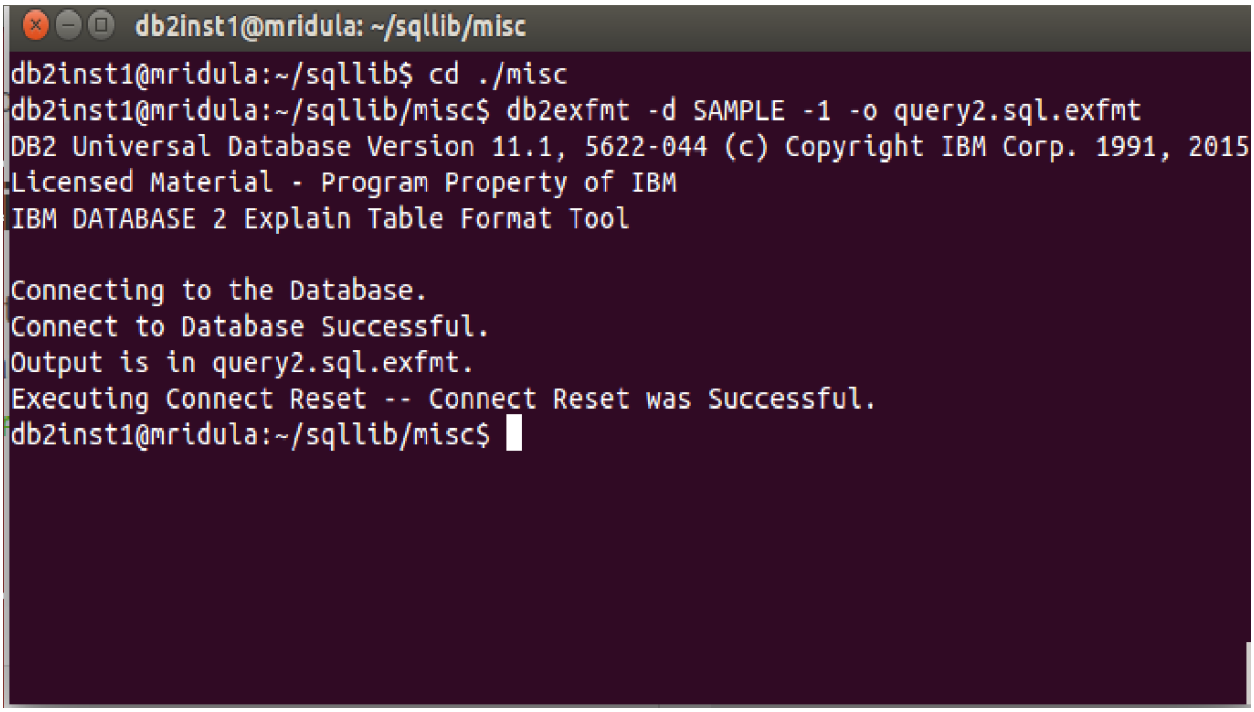
The result is a shown in the below capture.



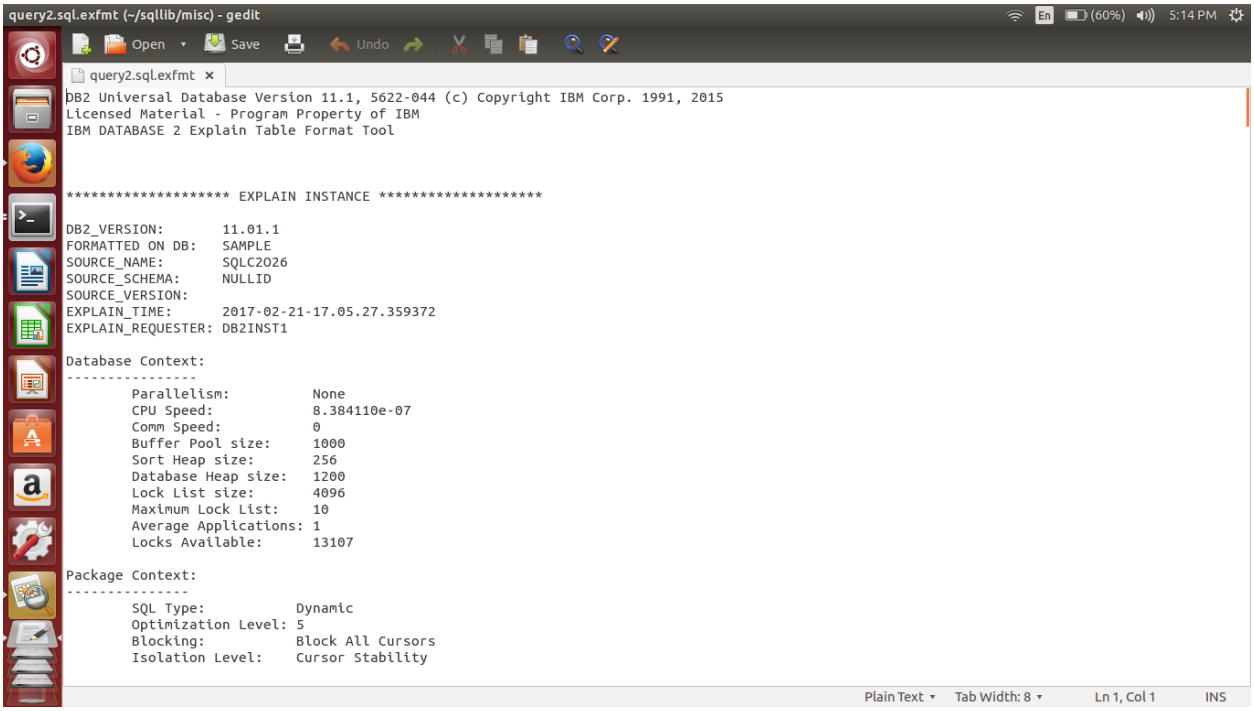
Then, we set the mode in the database to explain mode, so that the result of the query is converted to an explain plan. To generate this, we store the above query in a text file called ‘query2.sql’, and use this as an input to generate the plan.

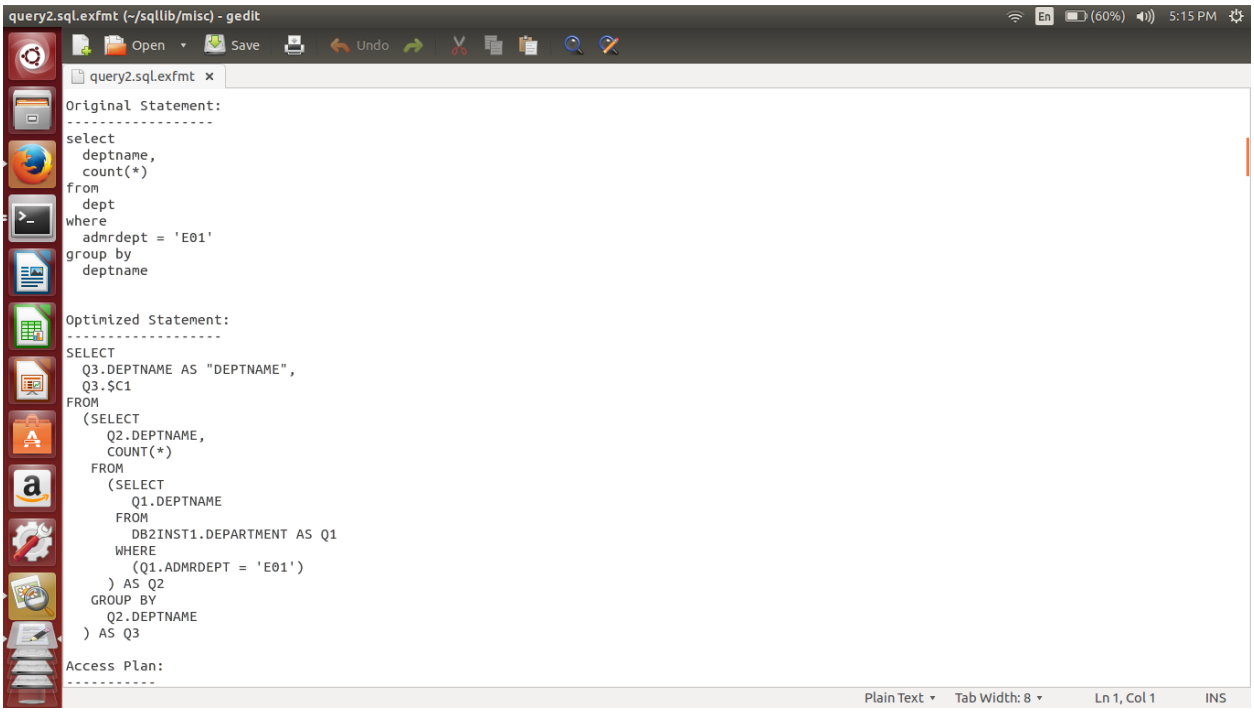


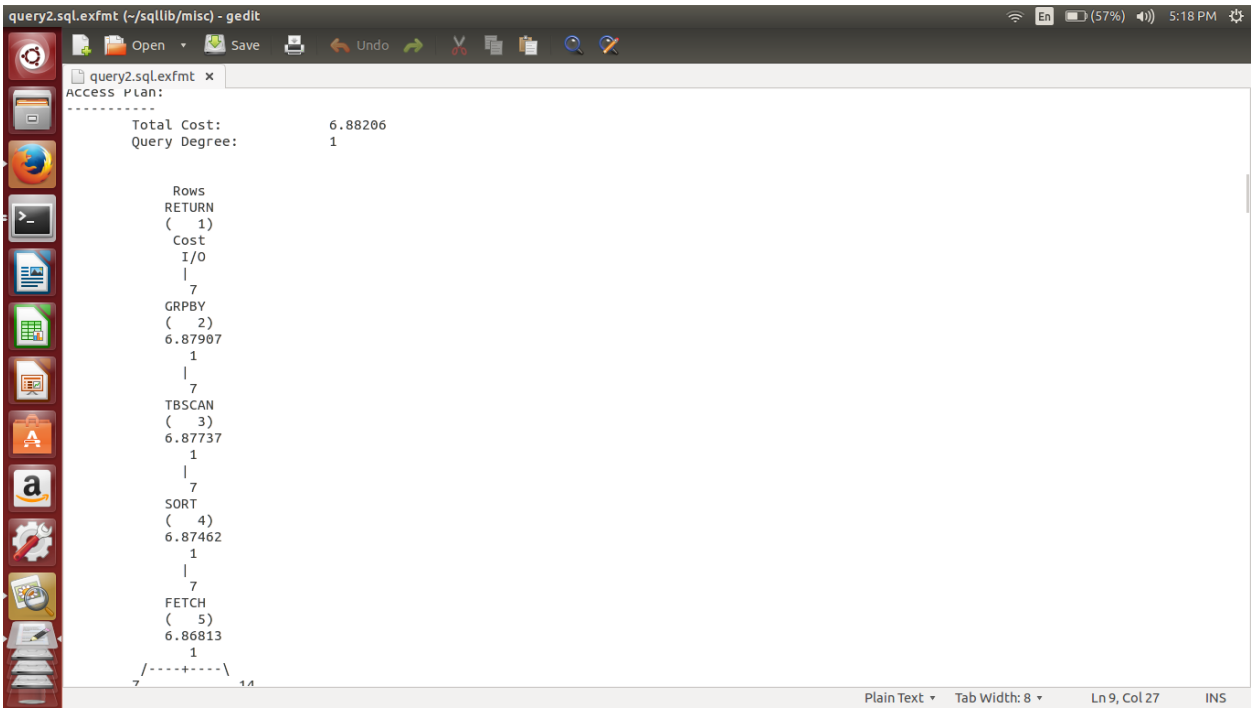
Then, to convert this into a human readable format, we use ‘db2exfmt’ tool. We give the ‘SAMPLE’ database as the input, and store the output in the file ‘query2.sql.exfmt’, where as the other options are taken as default.

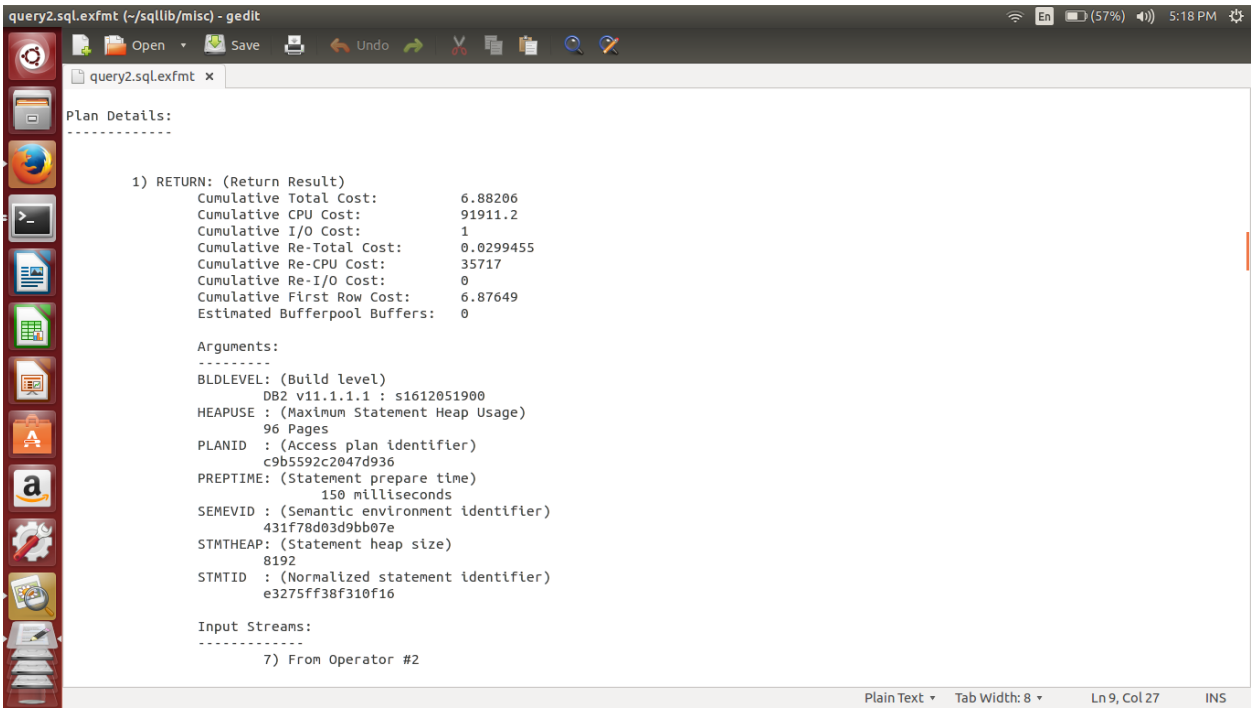


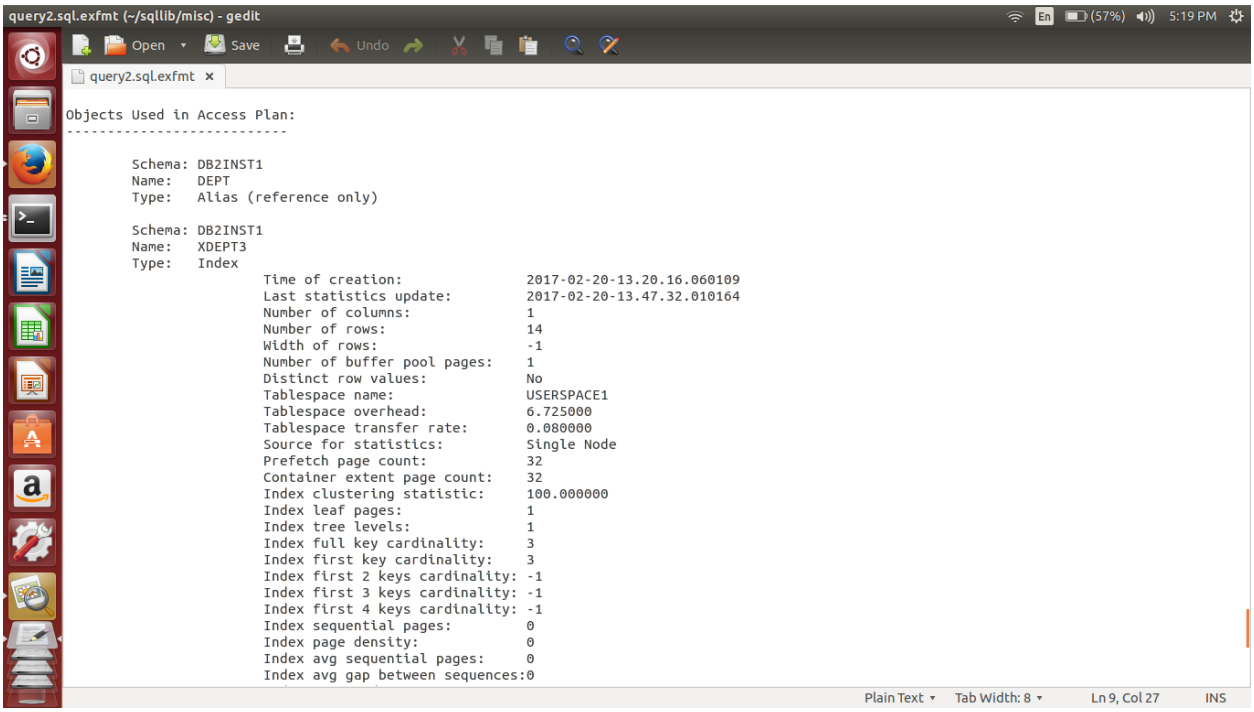
When the output file ‘query2.sql.exfmt’ is viewed, we can see the following in the explain plan file.






query2.sql.exfmt (~/sqllib/misc) - gedit
Open
Save
Undo
query2.sql.exfmt
Access Plan:
Total Cost: 6.88206
Query Degree: 1
Rows
RETURN
(1)
Cost
I/O
7
GRPBY
(2)
6.87907
1
7
TBSCAN
(3)
6.87737
1
7
SORT
(4)
6.87462
1
7
FETCH
(5)
6.86813
1
Plain Text
Tab Width: 8
Ln 9, Col 27
INS


query2.sql.exfmt (~/sqllib/misc) - gedit
Open
Save
Undo
query2.sql.exfmt
Plan Details:
1) RETURN: (Return Result)
Cumulative Total Cost: 6.88206
Cumulative CPU Cost: 91911.2
Cumulative I/O Cost: 1
Cumulative Re-Total Cost: 0.0299455
Cumulative Re-CPU Cost: 35717
Cumulative Re-I/O Cost: 0
Cumulative First Row Cost: 6.87649
Estimated Bufferpool Buffers: 0
Arguments:
BLDLEVEL: (Build level)
DB2 v11.1.1.1 : s1612051900
HEAPUSE : (Maximum Statement Heap Usage)
96 Pages
PLANID : (Access plan identifier)
c9b5592c2047d936
PREPTIME: (Statement prepare time)
150 milliseconds
SEMEVID : (Semantic environment identifier)
431f78d03d9bb07e
STMTHEAP: (Statement heap size)
8192
STMTID : (Normalized statement identifier)
e3275ff38f310f16
Input Streams:
7) From Operator #2
Plain Text
Tab Width: 8
Ln 9, Col 27
INS

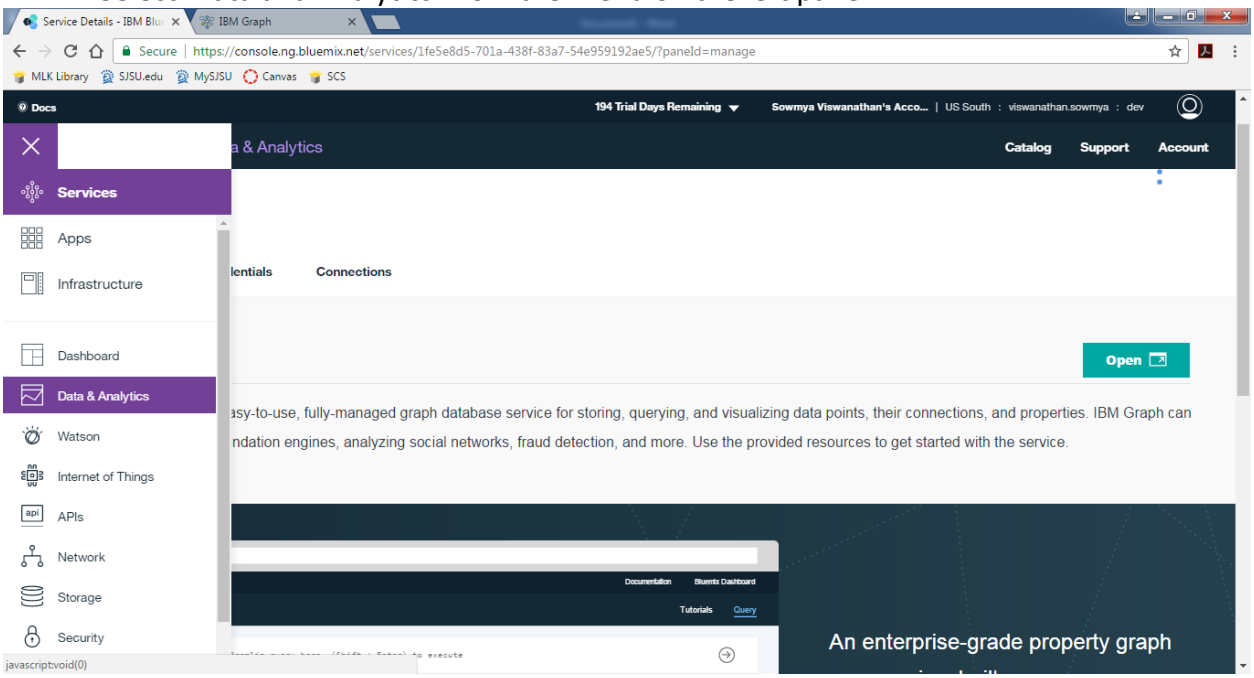

query2.sql.exfmt (~/sqllib/misc) - gedit
Open
Save
Undo
query2.sql.exfmt
Objects Used in Access Plan:
Schema: DB2INST1
Name: DEPT
Type: Alias (reference only)
Schema: DB2INST1
Name: XDEPT3
Type: Index
Time of creation: 2017-02-20-13.20.16.060109
Last statistics update: 2017-02-20-13.47.32.010164
Number of columns: 1
Number of rows: 14
Width of rows: -1
Number of buffer pool pages: 1
Distinct row values: No
Tablespace name: USERSPACE1
Tablespace overhead: 6.725000
Tablespace transfer rate: 0.080000
Source for statistics: Single Node
Prefetch page count: 32
Container extent page count: 32
Index clustering statistic: 100.000000
Index leaf pages: 1
Index tree levels: 1
Index full key cardinality: 3
Index first key cardinality: 3
Index first 2 keys cardinality: -1
Index first 3 keys cardinality: -1
Index first 4 keys cardinality: -1
Index sequential pages: 0
Index page density: 0
Index avg sequential pages: 0
Index avg gap between sequences:0
Plain Text
Tab Width: 8
Ln 9, Col 27
INS

IBM GRAPH DB

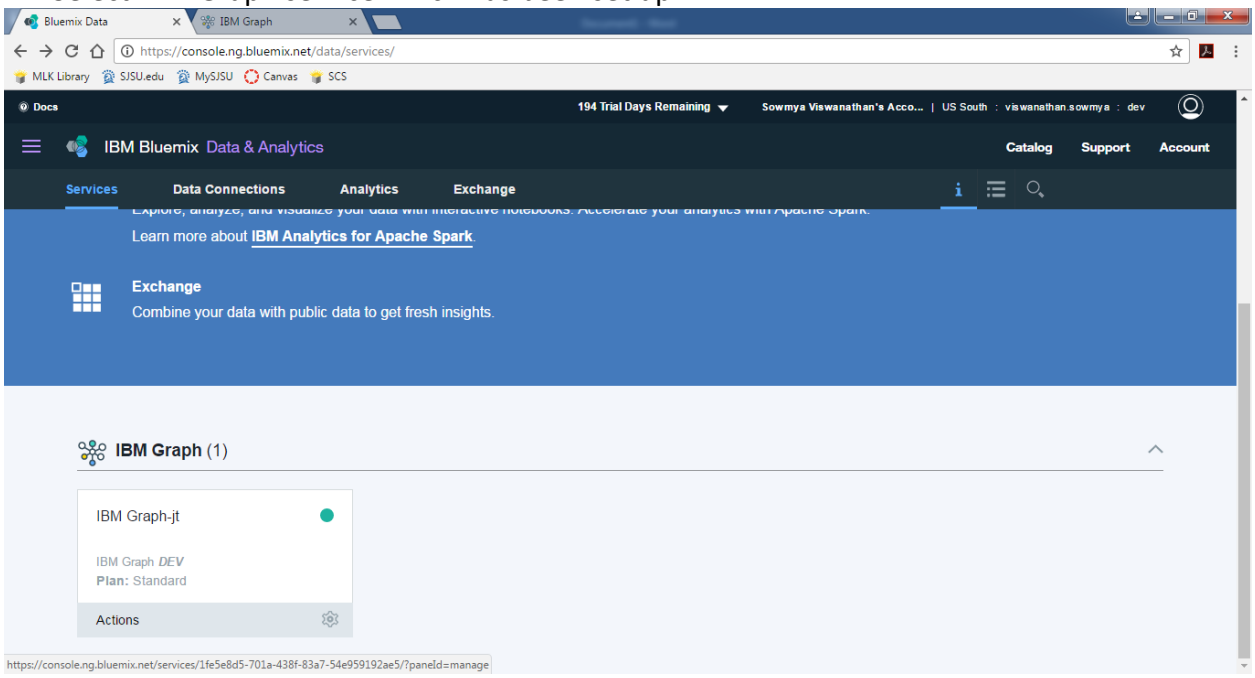
IBM Graph provides an HTTP API for manipulating and querying graphs. Requests and responses use the JSON format.

Graph DB set up and UI features:

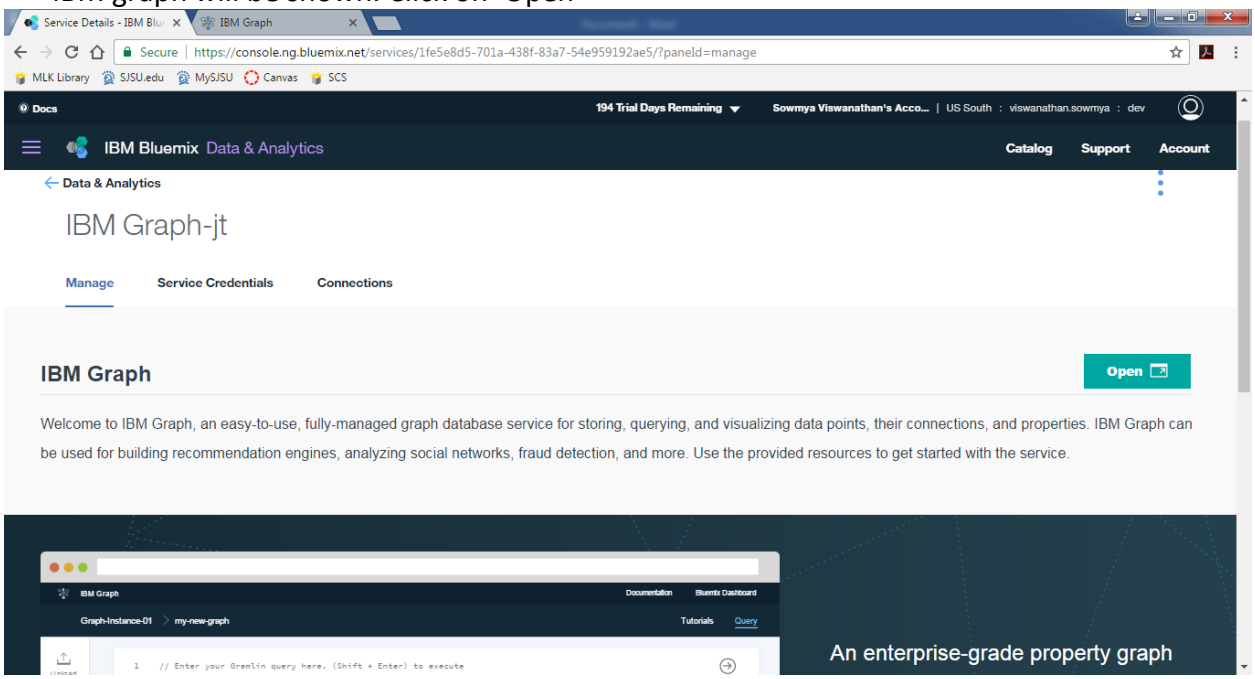
- Select 'Data and Analytics' from the Menu on the left pane.



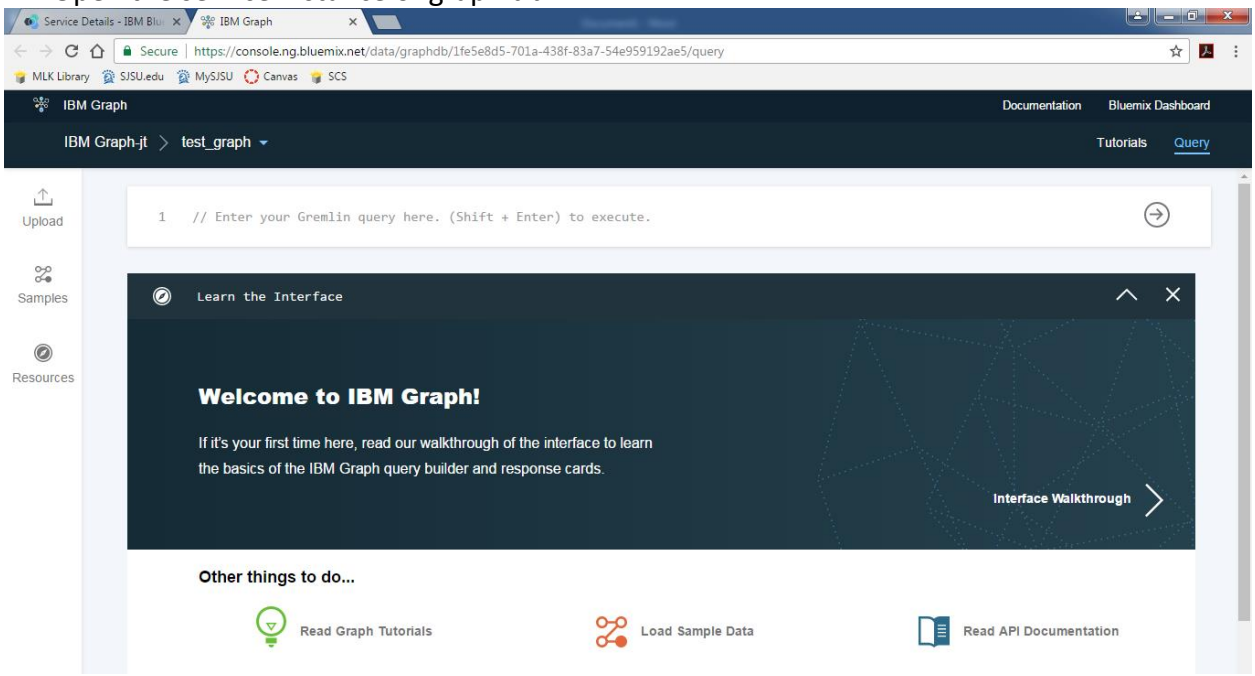
- Select IBM Graph service which has been set up



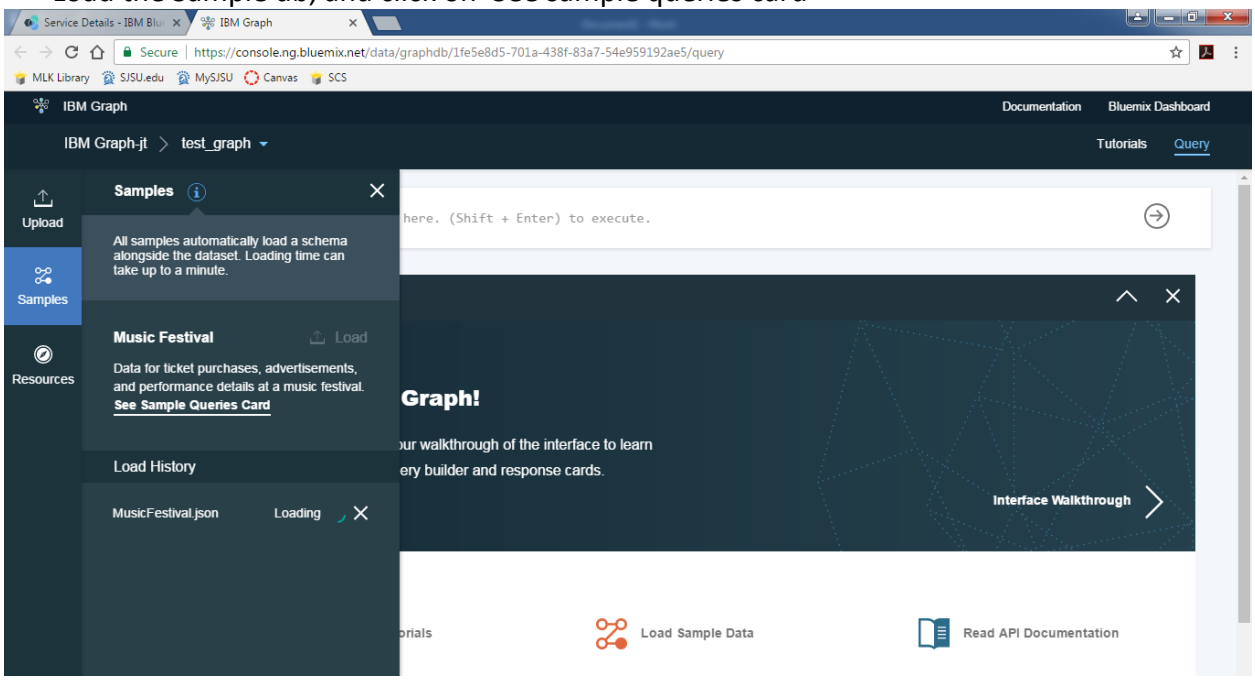
- IBM graph will be shown. Click on ‘Open’



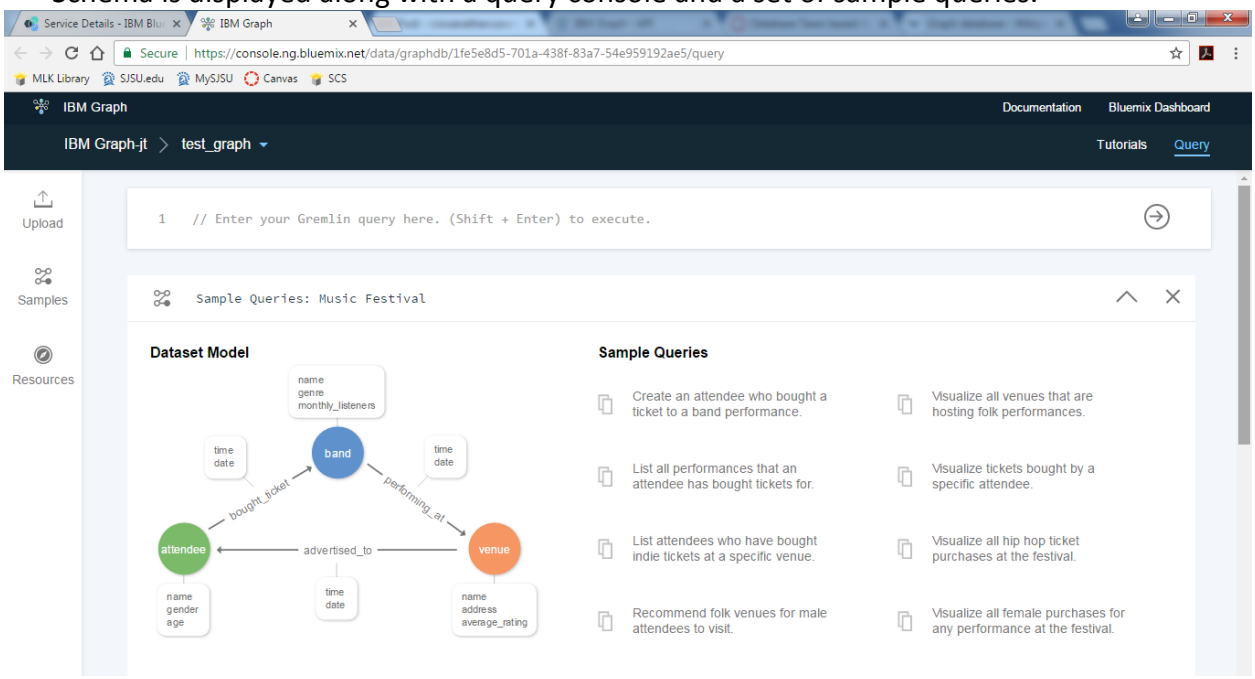
- Open the service instance of graph db.



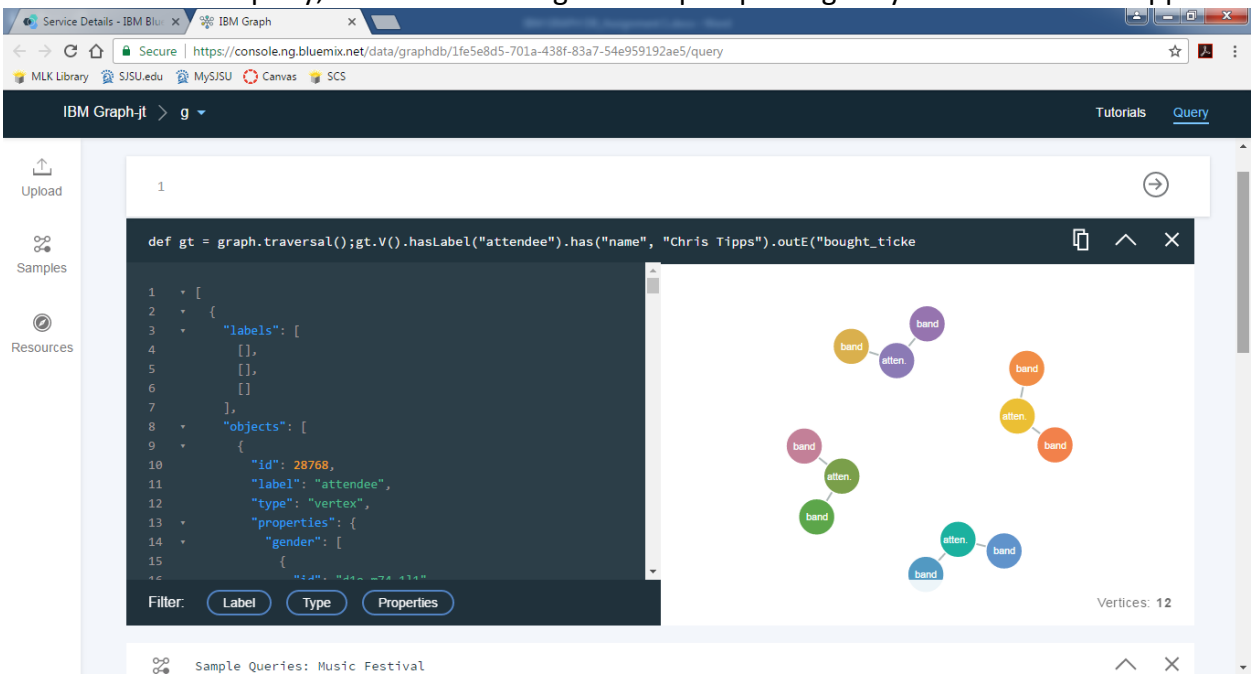
- Load the Sample db, and click on 'See sample queries card'



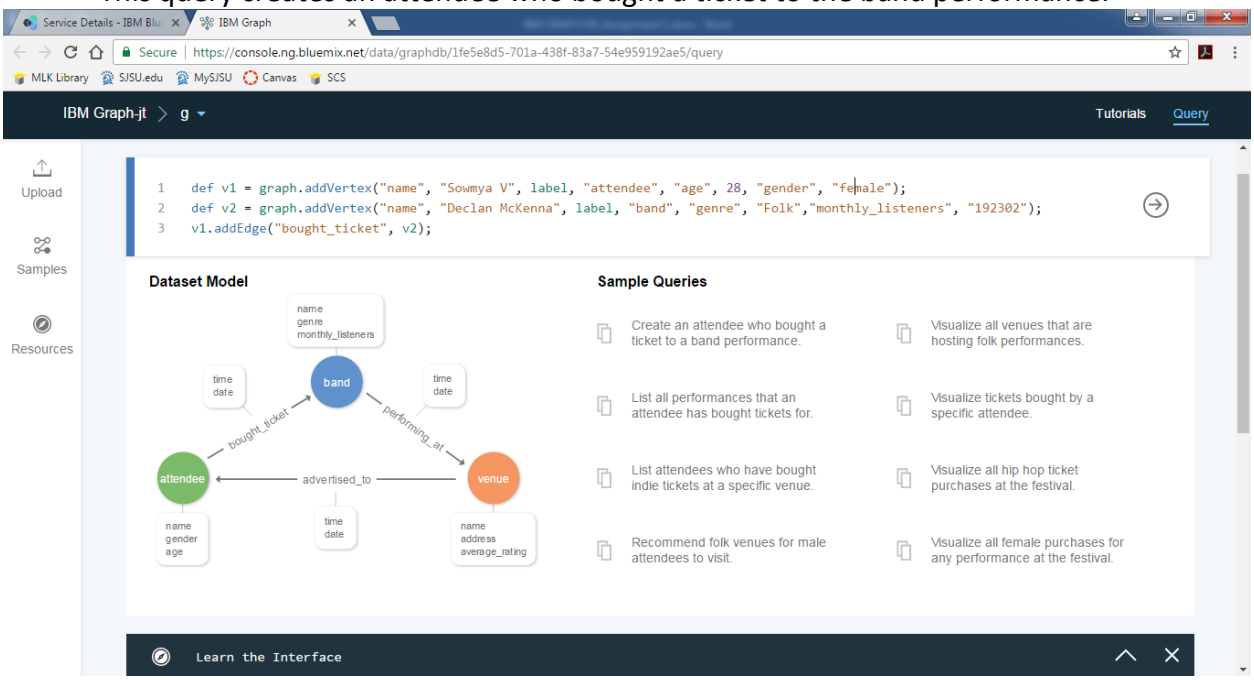
- Schema is displayed along with a query console and a set of sample queries.



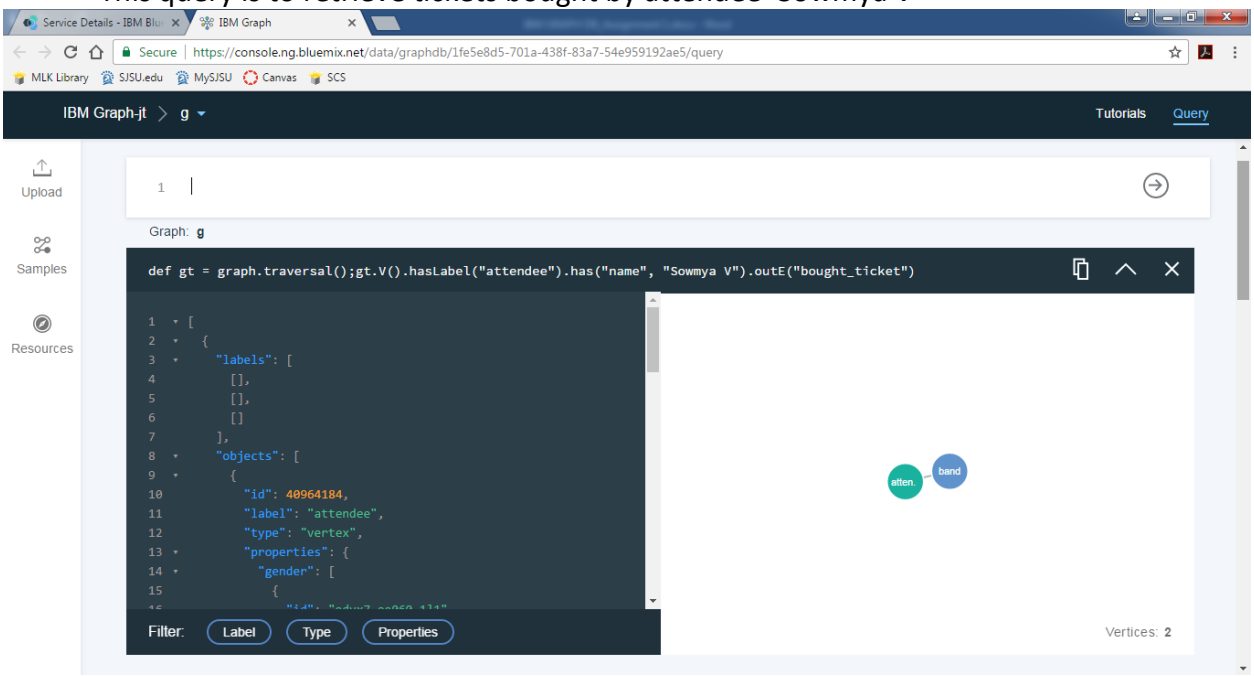
- **Query**: def gt = graph.traversal();gt.V().hasLabel("attendee").has("name", "Chris Tipps").outE("bought_ticket").inV().hasLabel("band").has("genre","Hip Hop").path(); The above query, lists all tickets of genre: 'Hip Hop' bought by attendee - Chris Tipps



- Query: def v1 = graph.addVertex("name", "Sowmya V", label, "attendee", "age", 28, "gender", "female"); def v2 = graph.addVertex("name", "Declan McKenna", label, "band", "genre", "Folk","monthly_listeners", "192302"); v1.addEdge("bought_ticket", v2);
  This query creates an attendee who bought a ticket to the band performance.

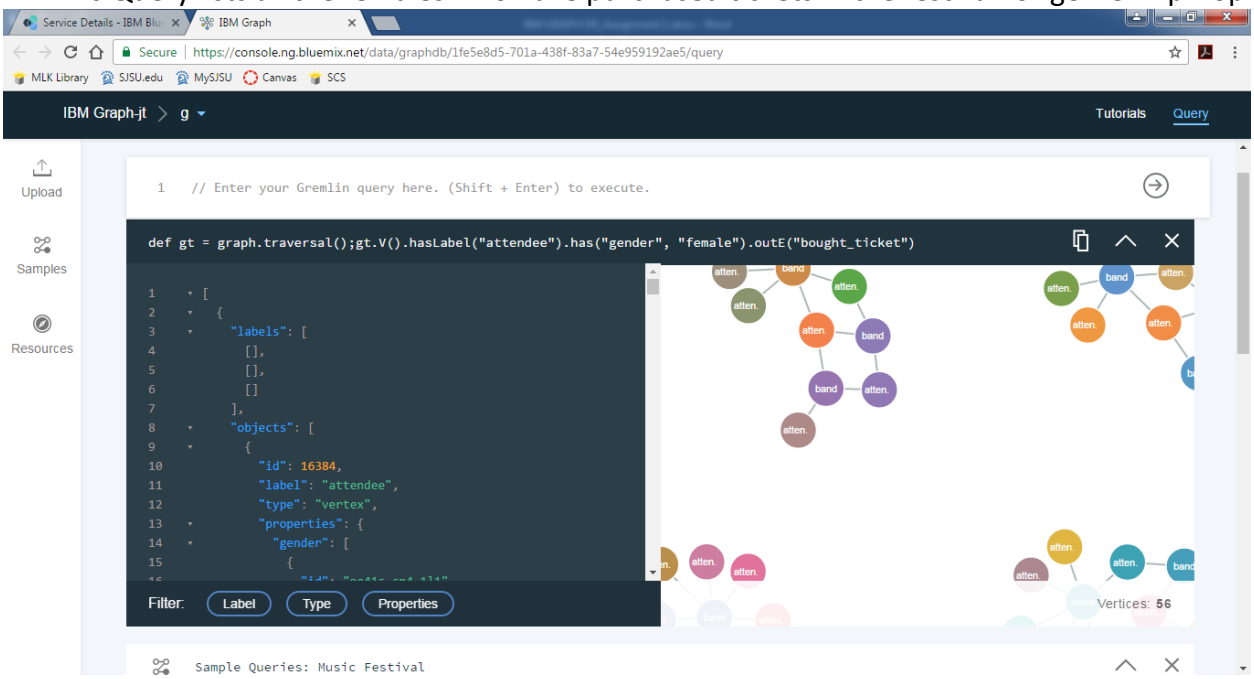


- Query: def gt = graph.traversal(); gt.V().hasLabel("attendee").has("name", "Sowmya V").outE("bought_ticket").inV().hasLabel("band").path();
  This query is to retrieve tickets bought by attendee ‘Sowmya V’

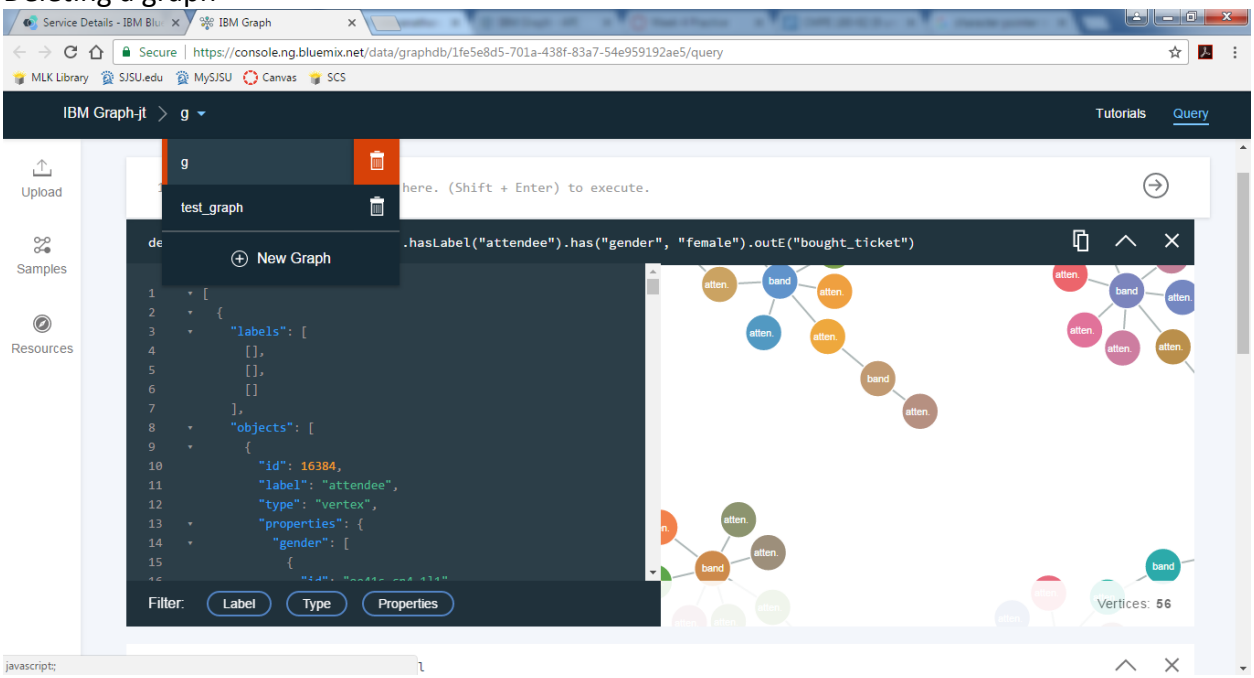


- Query: def gt = graph.traversal();gt.V().hasLabel("attendee").has("gender","female").outE("bought_ticket").inV().hasLabel("band").has("genre", "Hip Hop").path();

This Query lists all the females who have purchased tickets in the festival for genre ‘Hip Hop’



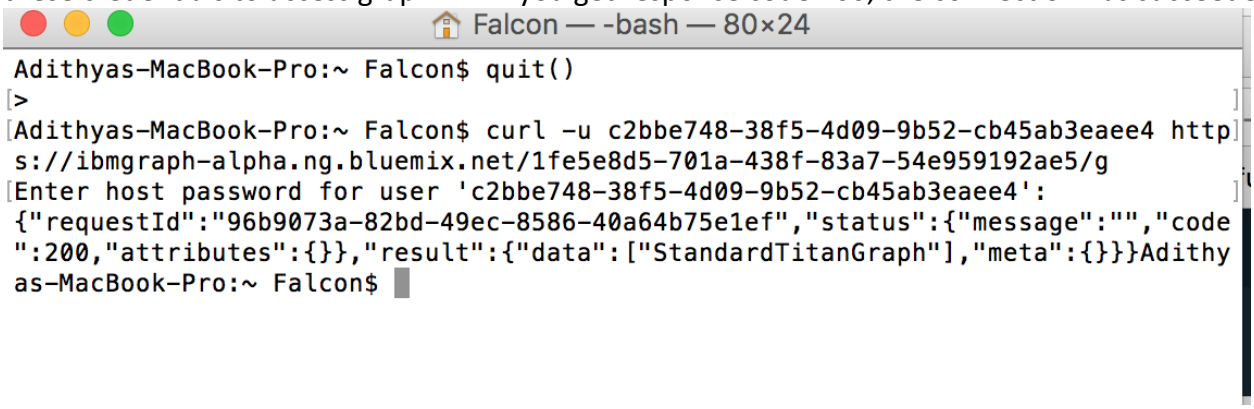
Deleting a graph



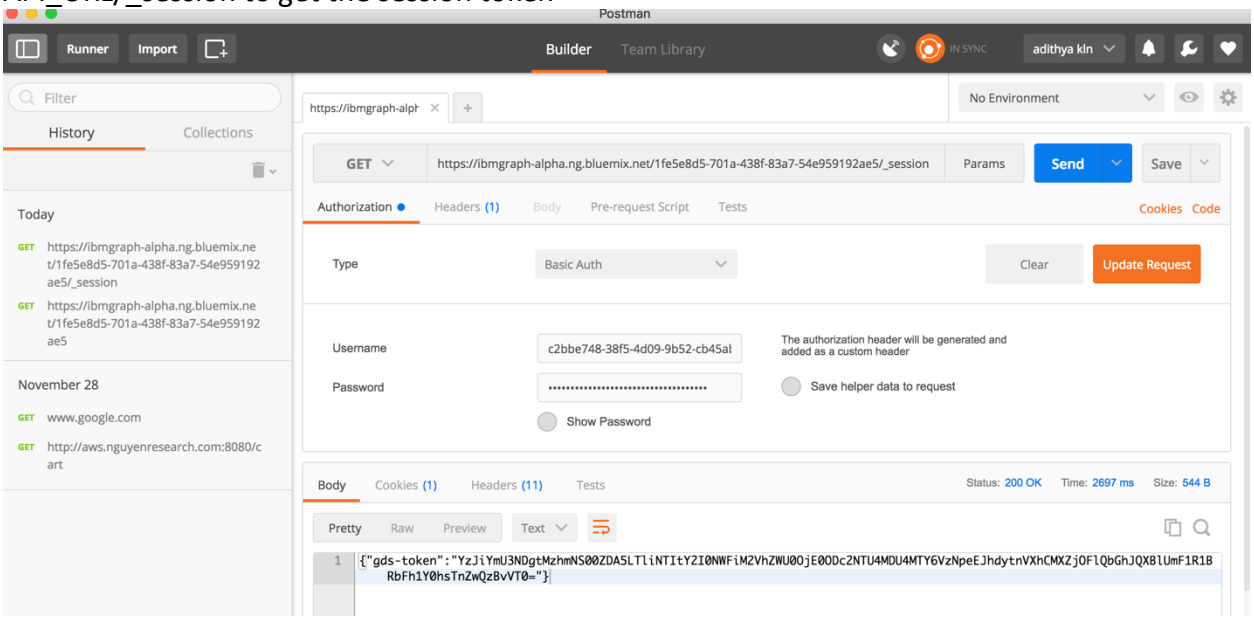
Using APIs to perform operations on database

1. Authentication – Session API

From the Security tab in the IBM Graph service, you can find the API_URL, username and password. We need these credentials to access graph DB. If you get response code 200, the connection has succeeded

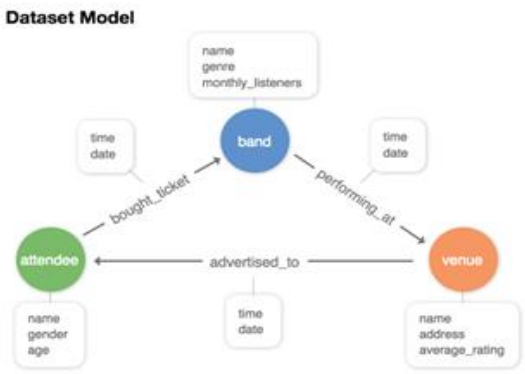


Now generate token to establish connection. We fire a GET request with the same credentials to the API_URL/_session to get the session token



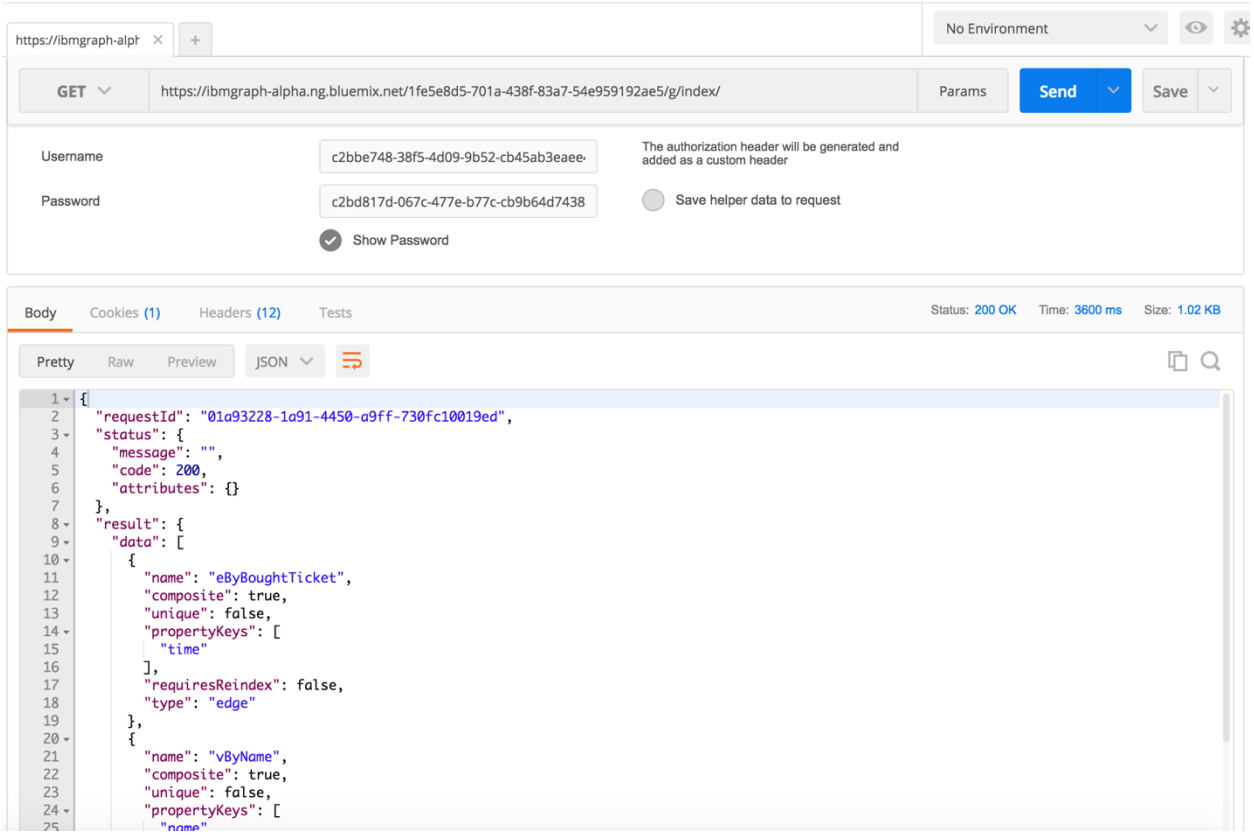
This session will be used to query schema at API_URL/g/

A graphical equivalent and the json schema received is as below:



```
"result": {
   "data": [
    {
     "propertyKeys": [
      {
       "name": "address",
       "dataType": "String",
       "cardinality": "SINGLE"
      },
      {
       "name": "average_rating",
       "dataType": "Float",
       "cardinality": "SINGLE"
      },
      {
       "name": "name",
       "dataType": "String",
       "cardinality": "SINGLE"
      },
      {
       "name": "gender",
       "dataType": "String",
       "cardinality": "SINGLE"
      },
      {
       "name": "age",
       "dataType": "Integer",
       "cardinality": "SINGLE"
      },
      {
       "name": "genre",
       "dataType": "String",
       "cardinality": "SINGLE"
      },
      {
       "name":
"monthly_listeners",
       "dataType": "String",
       "cardinality": "SINGLE"
      },
      {
       "name": "date",
       "dataType": "String",
       "cardinality": "SINGLE"
      },
      {
       "name": "time",
       "dataType": "String",
       "cardinality": "SINGLE"
      }
     ],
     "vertexLabels": [
      {
       "name": "person"
      },
      {
       "name": "Attendee"
      },
      {
       "name": "Band"
      },
      {
       "name": "Venue"
      }
     ],
     "edgeLabels": [
      {
       "name":
"bought_ticket",
       "directed": true,
       "multiplicity": "MULTI"
      },
      {
       "name":
"advertised_to",
       "directed": true,
       "multiplicity": "MULTI"
      },
      {
       "name":
"performing_at",
       "directed": true,
       "multiplicity": "MULTI"
      }
     ],
     "vertexIndexes": [
      {
       "name": "vByName",
       "composite": true,
       "unique": false,
       "propertyKeys": [
        "name"
       ],
       "requiresReindex": false,
       "type": "vertex"
      },
      {
       "name": "vByGender",
       "composite": true,
       "unique": false,
       "propertyKeys": [
        "gender"
       ],
       "requiresReindex": false,
       "type": "vertex"
      },
      {
       "name": "vByGenre",
       "composite": true,
       "unique": false,
       "propertyKeys": [
        "genre"
       ],
       "requiresReindex": false,
       "type": "vertex"
      }
     ],
     "edgeIndexes": [
      {
       "name": "eByBoughtTicket",
       "composite": true,
       "unique": false,
       "propertyKeys": [
        "time"
       ],       "requiresReindex": false,
       "type": "edge"
      }   ]    }   ]
```

2. Index APIs
   a. Get existing indices:



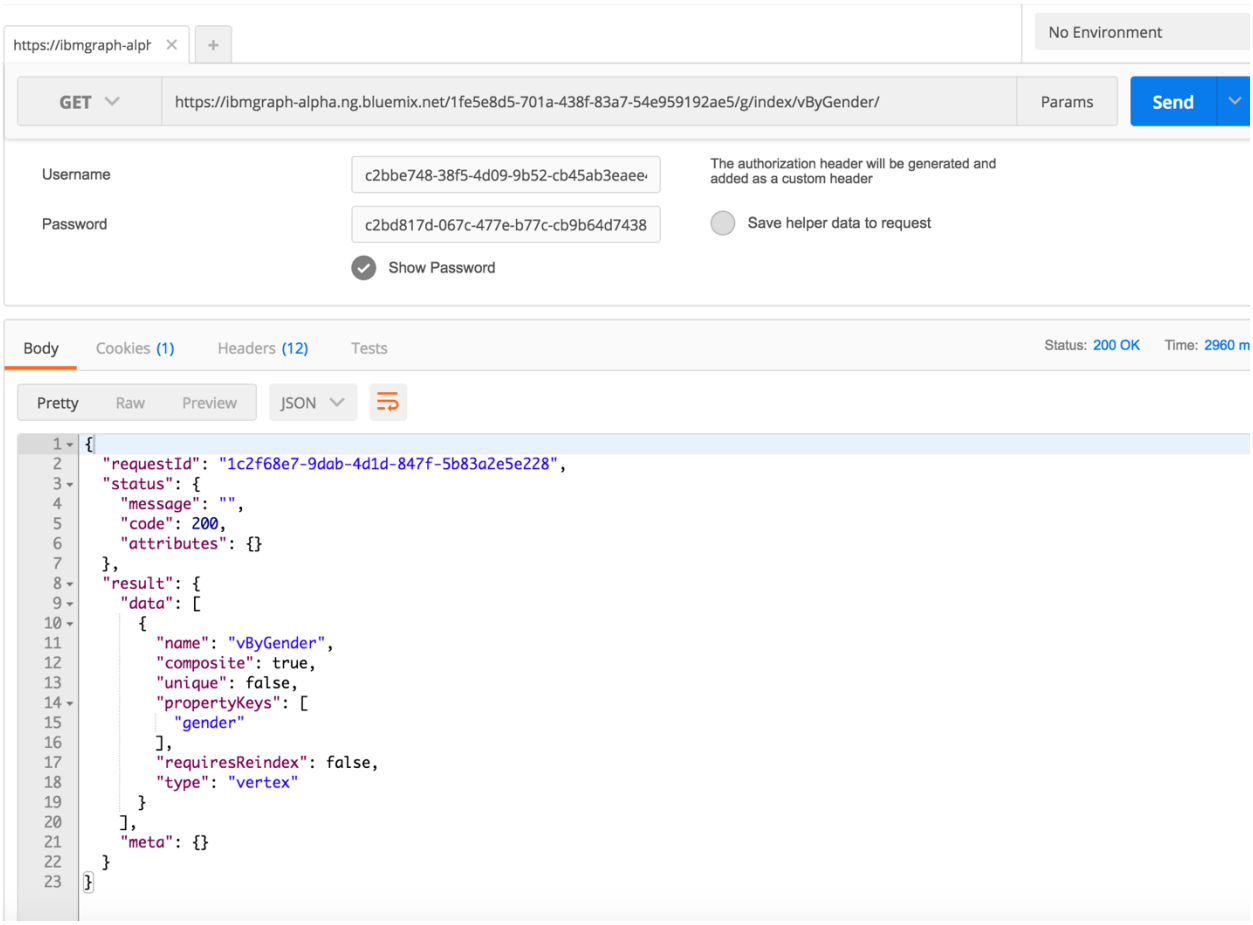
response:


```
{
 "requestId": "01a93228-1a91-4450-a9ff-730fc10019ed",
 "status": {
  "message": "",
  "code": 200,
  "attributes": {}
 },
 "result": {
  "data": [
   {
    "name": "eByBoughtTicket",
    "composite": true,
    "unique": false,
    "propertyKeys": [
     "time"
    ],
    "requiresReindex": false,
    "type": "edge"
   },
   {
    "name": "vByName",
    "composite": true,
    "unique": false,
    "propertyKeys": [
     "name"
    ],
    "requiresReindex": false,
    "type": "vertex"
   },
   {
    "name": "vByGender",
    "composite": true,
    "unique": false,
    "propertyKeys": [
     "gender"
    ],
    "requiresReindex": false,
    "type": "vertex"
   },
   {
    "name": "vByGenre",
    "composite": true,
```

```
  "unique": false,
  "propertyKeys": [
   "time"
  ],
  "requiresReindex": false,
  "type": "edge"
 },
 {
  "name": "vByName",
  "composite": true,
  "unique": false,
  "propertyKeys": [
   "name"
  ],
  "requiresReindex": false,
  "type": "vertex"
 },
```

```
   "unique": false,
   "propertyKeys": [
    "genre"
   ],
   "requiresReindex": false,
   "type": "vertex"
  }
 ],
 "meta": {}
}
}
```

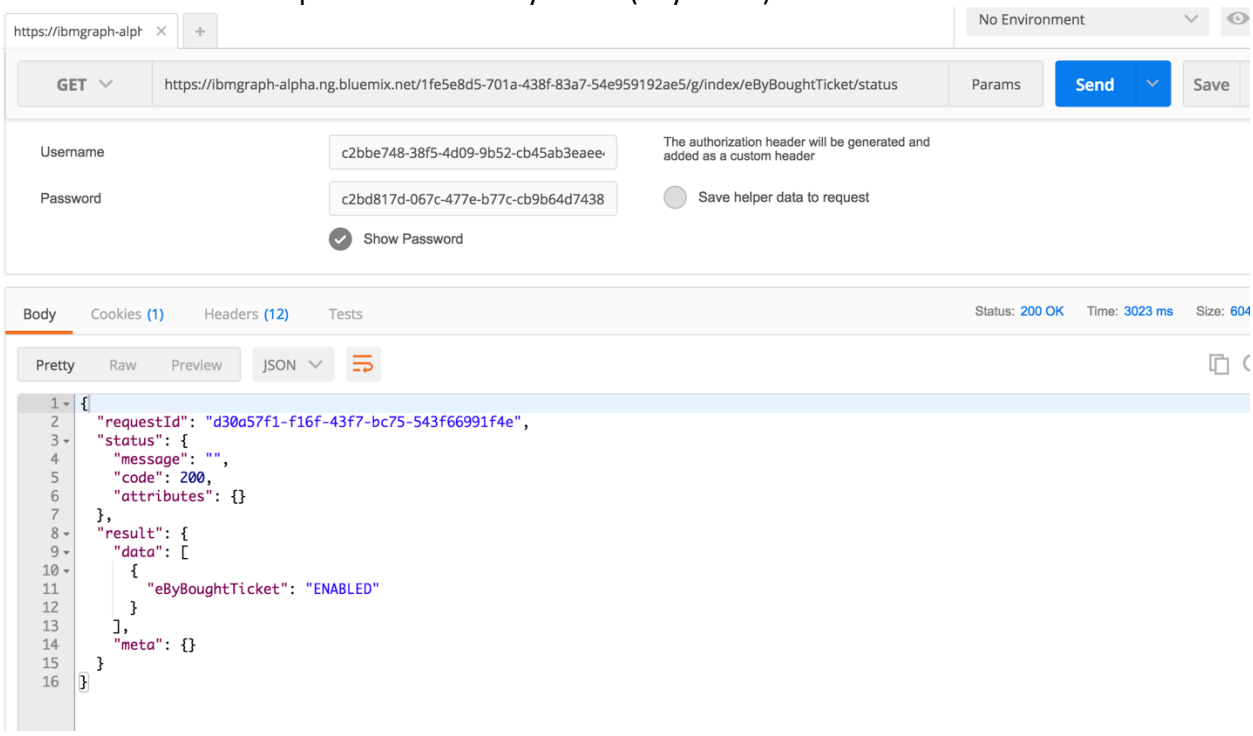

b. GET a specific index by value



response:


```
{
 "requestId": "1c2f68e7-9dab-4d1d-847f-5b83a2e5e228",
 "status": {
  "message": "",
  "code": 200,
  "attributes": {}
 },
 "result": {
  "data": [
   {
    "name": "vByGender",
    "composite": true,
    "unique": false,
    "propertyKeys": [
     "gender"
    ],
    "requiresReindex": false,
    "type": "vertex"
   }
  ],
  "meta": {}
 }
}
```


c. GET status of a particular index by name (vByName) to see if it's enabled or disabled:



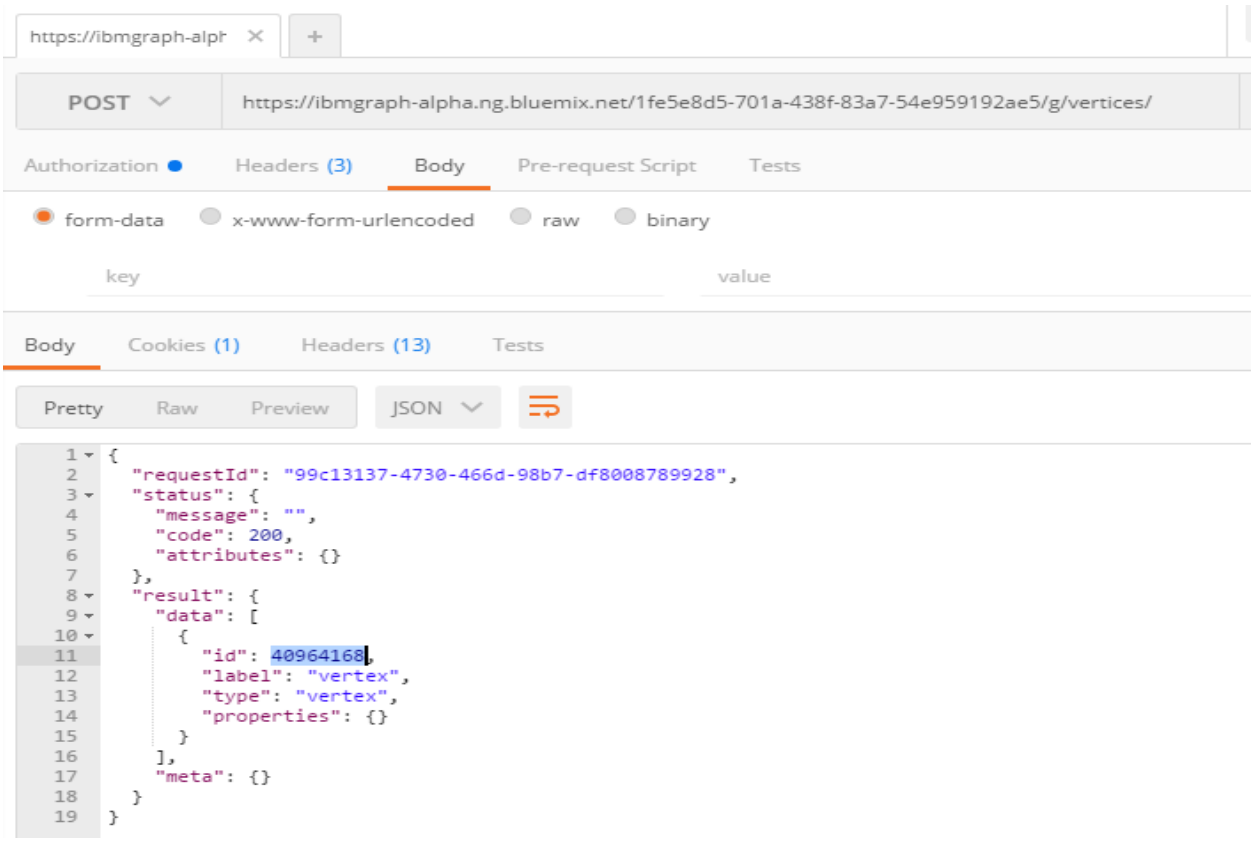
response:


```
{
  "requestId": "d30a57f1-f16f-43f7-bc75-543f66991f4e",
  "status": {
    "message": "",
    "code": 200,
    "attributes": {}
  },
  "result": {
    "data": [
      {
        "eByBoughtTicket": "ENABLED"
      }
    ],
    "meta": {}
  }}
```

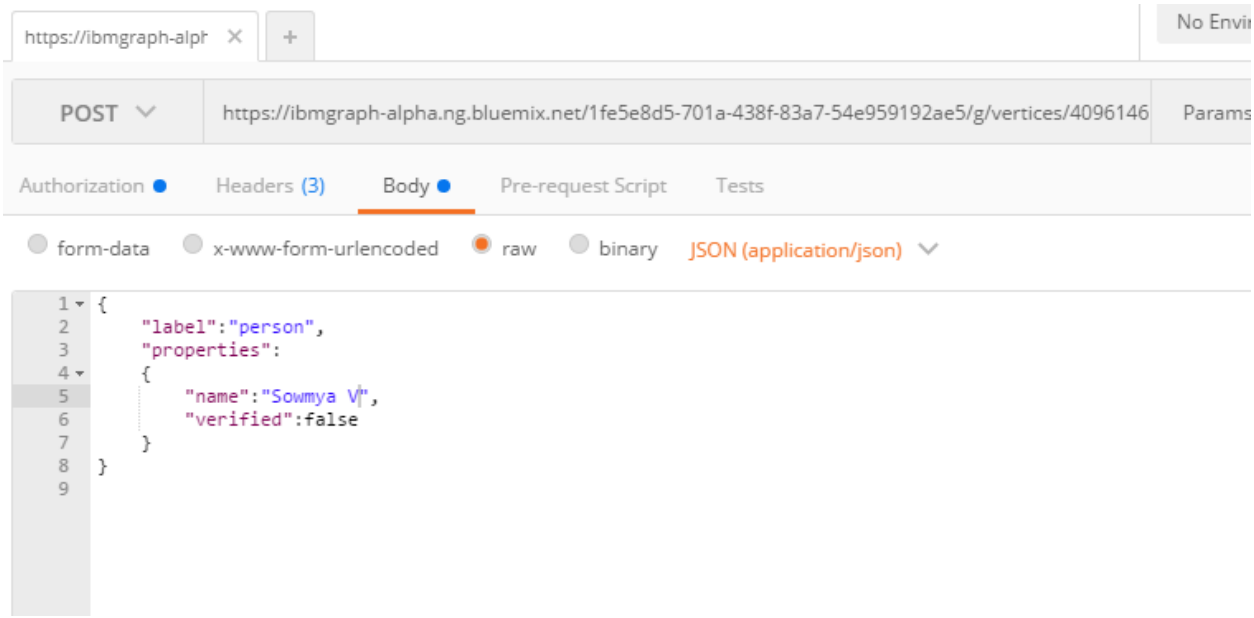

d. DELETE an existing index

Check with GET if this index exists

3. Vertex APIs
   a. POST method to create a new vertex. In below example, a vertex is created with id 40964168 but with no properties as we need to specify a property in payload to assign one.



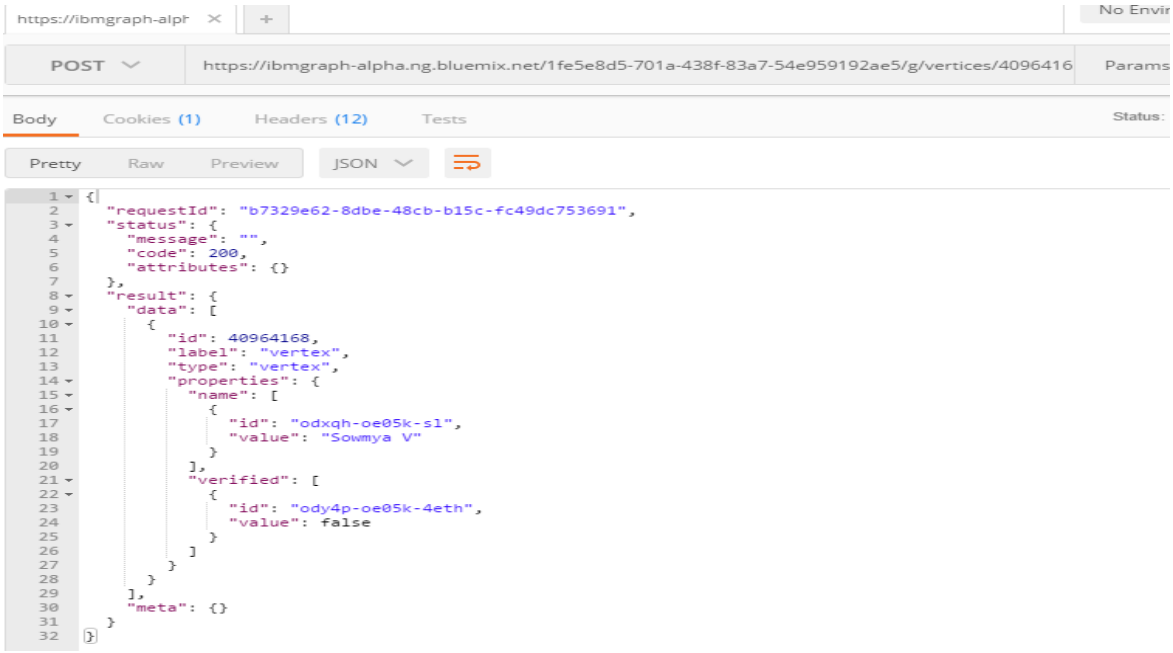
b. POST method with payload containing properties:



Request:

{

```
        "label":"person",
        "properties":
        {
                "name":"Sowmya V",
                "verified":false
        }
}
```

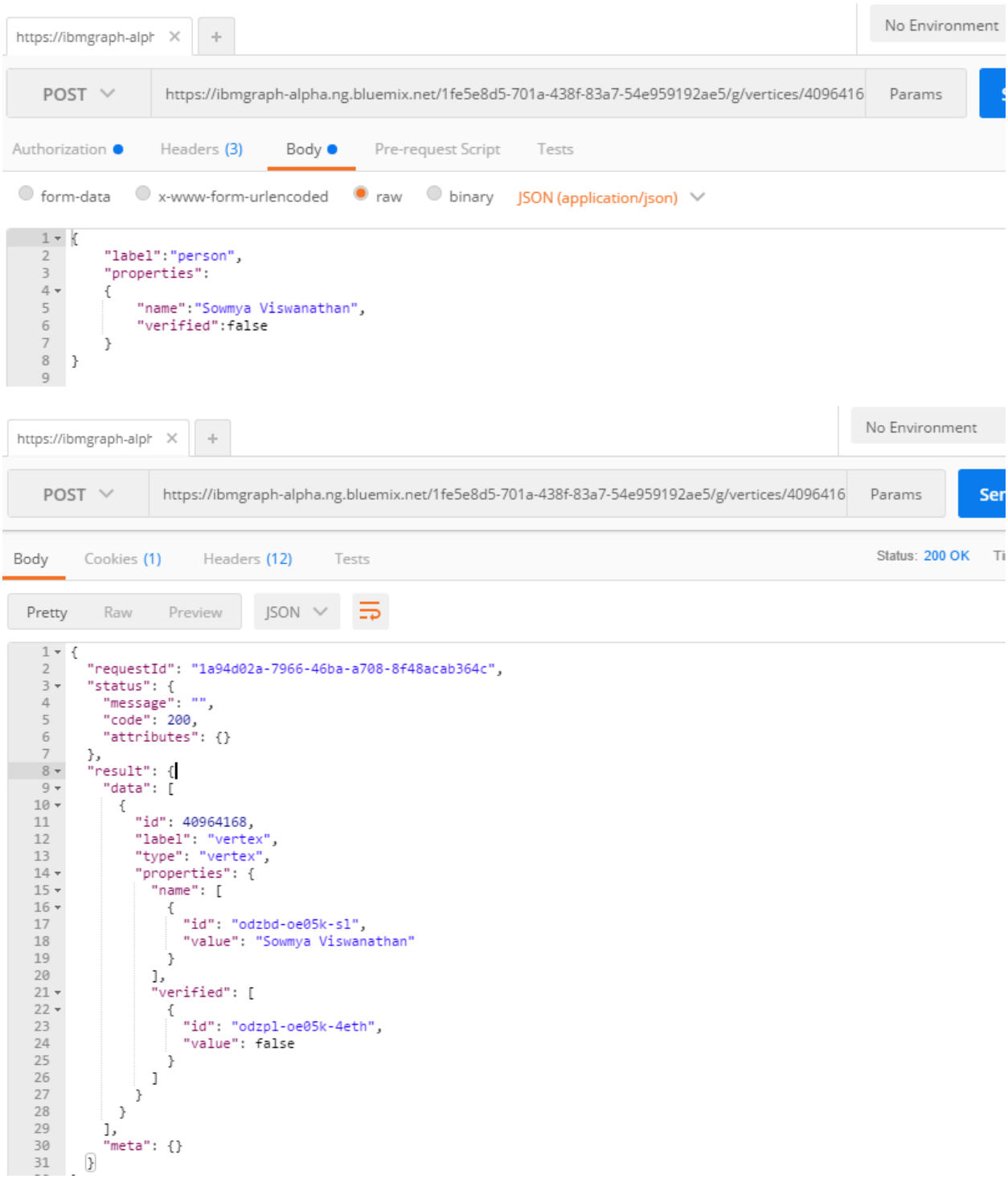


Response:


```
{
  "requestId": "b7329e62-8dbe-48cb-b15c-fc49dc753691",
  "status": {
    "message": "",
    "code": 200,
    "attributes": {}
  },
  "result": {
    "data": [
      {
        "id": 40964168,
        "label": "vertex",
        "type": "vertex",
        "properties": {
          "name": [
            {
              "id": "odxqh-oe05k-sl",
              "value": "Sowmya V"
            }
          ],
          "verified": [
            {
              "id": "ody4p-oe05k-4eth",
              "value": false
            }
          ]
        }
      }
    ],
    "meta": {}
  }
}
```

```
"type": "vertex",
"properties": {
 "name": [
 ],
 "meta": {}
}
}
```

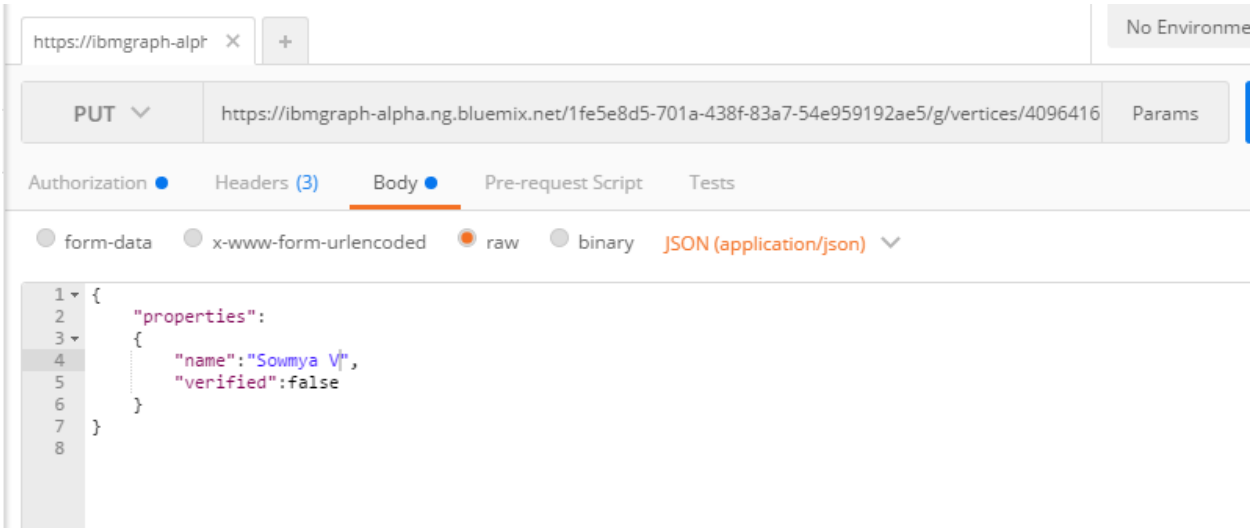

c. Update the properties using POST

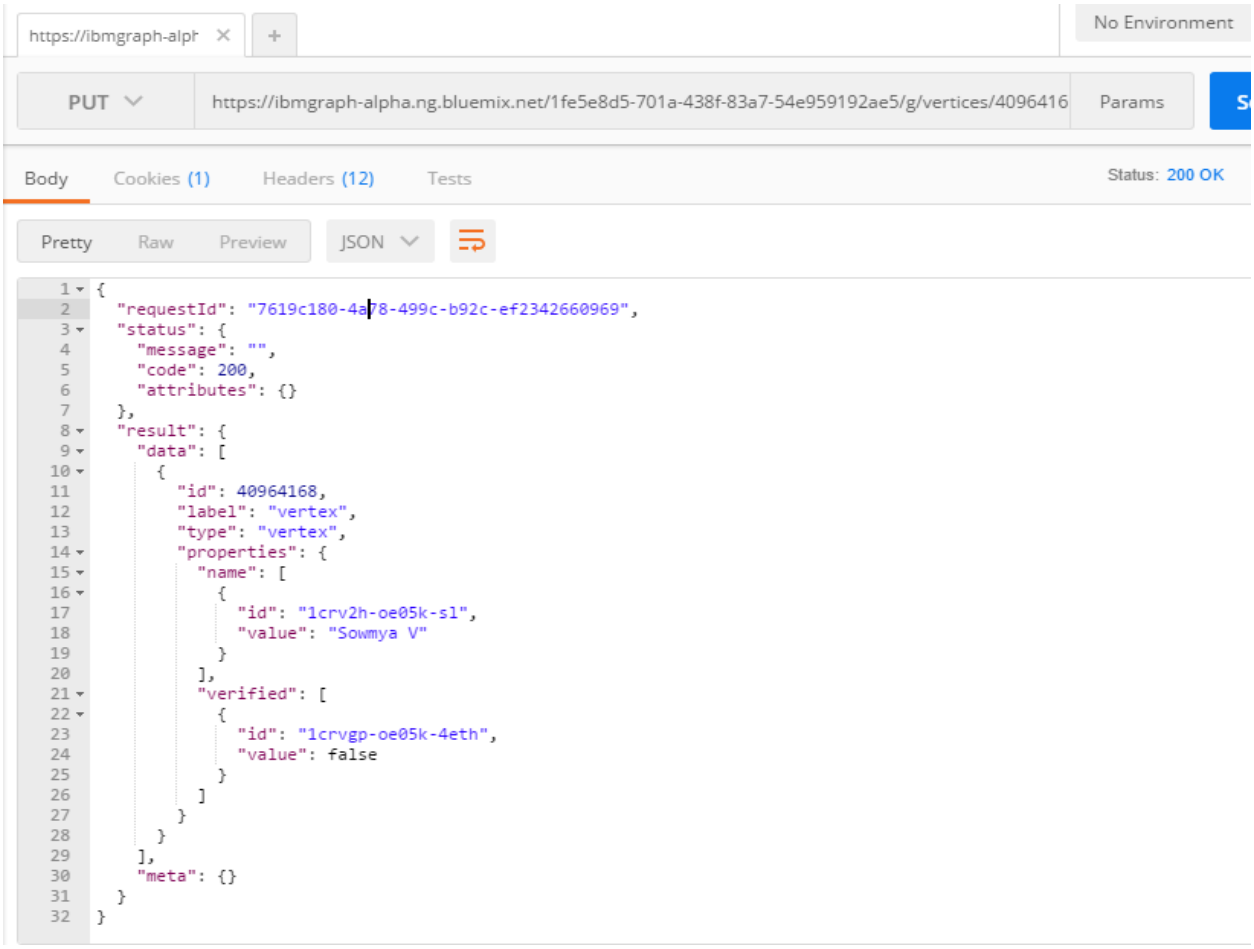


Response:


```
{
  "requestId":      "1a94d02a-7966-46ba-a708-
8f48acab364c",
  "status": {
    "message": "",
    "code": 200,
    "attributes": {}
  },
  "result": {
    "data": [
      {
        "id": 40964168,
        "label": "vertex",
        "type": "vertex",
        "properties": {
          "name": [
            {
"id": "odzbd-oe05k-sl",
              "value": "Sowmya Viswanathan"
            }
          ],
          "verified": [
            {
              "id": "odzpl-oe05k-4eth",
              "value": false
            }
          ]
        }
      }
    ],
    "meta": {}
  }
}
```


d. PUT method to delete previous updates and update it with the new value

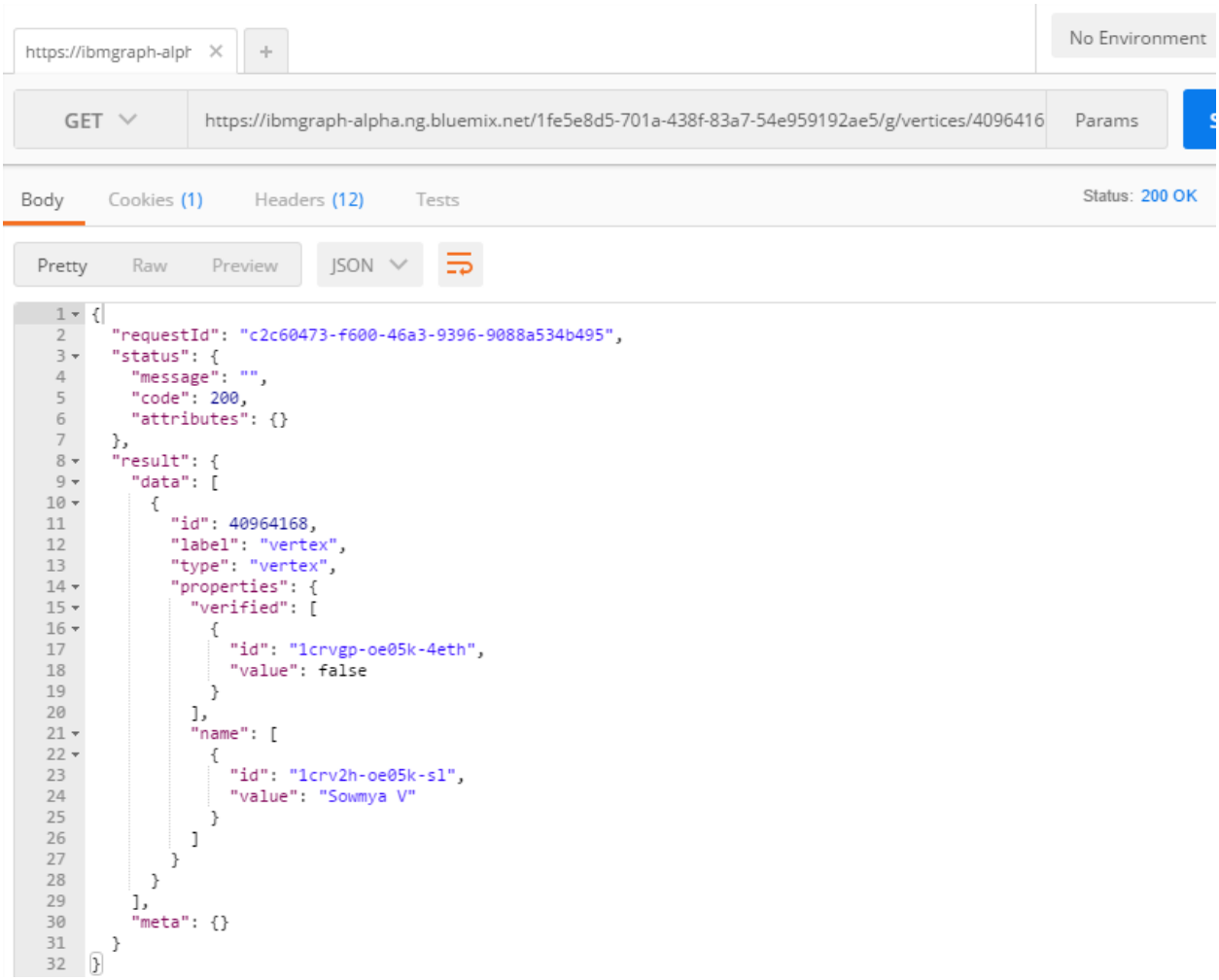




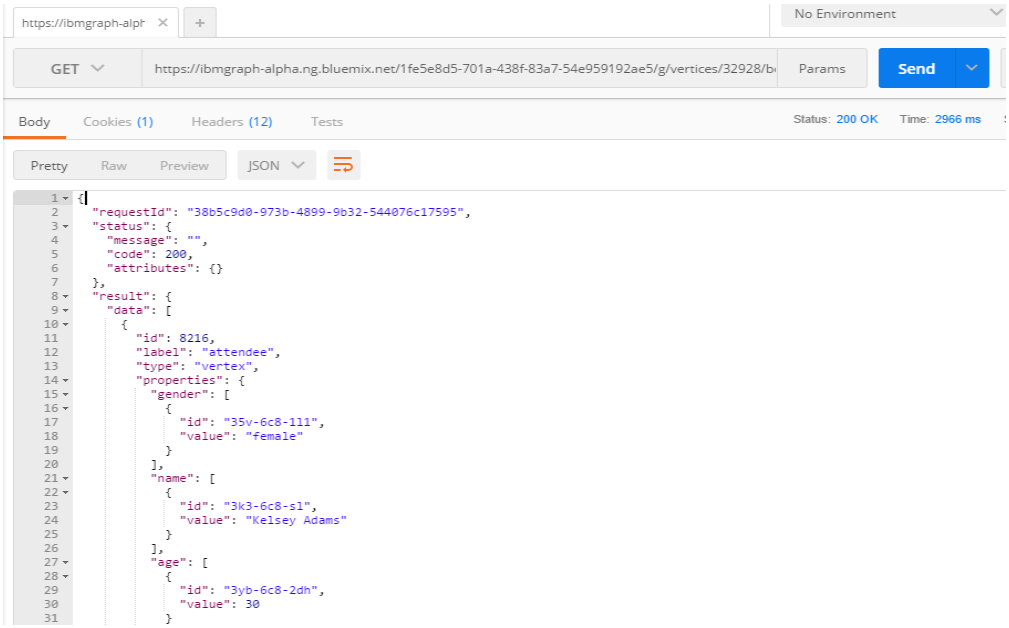
Response:


```
{
  "requestId": "7619c180-4a78-499c-b92c-ef2342660969",
  "status": {
    "message": "",
    "code": 200,
    "attributes": {}
  },
  "result": {
    "data": [
      {
        "id": 40964168,
        "label": "vertex",
        "type": "vertex",
        "properties": {
          "name": [
            {
              "id": "1crv2h-oe05k-sl",
              "value": "Sowmya V"
            }
          ],
          "verified": [
            {
              "id": "1crvgp-oe05k-4eth",
              "value": false
            }
          ]
        }
      }
    ],
    "meta": {}
  }
}
```

GET call for the same id. The last made changes are available



e. Check the out and in vertices to a particular vertex. Below id mentioned is that of a 'venue' in db.



Response:

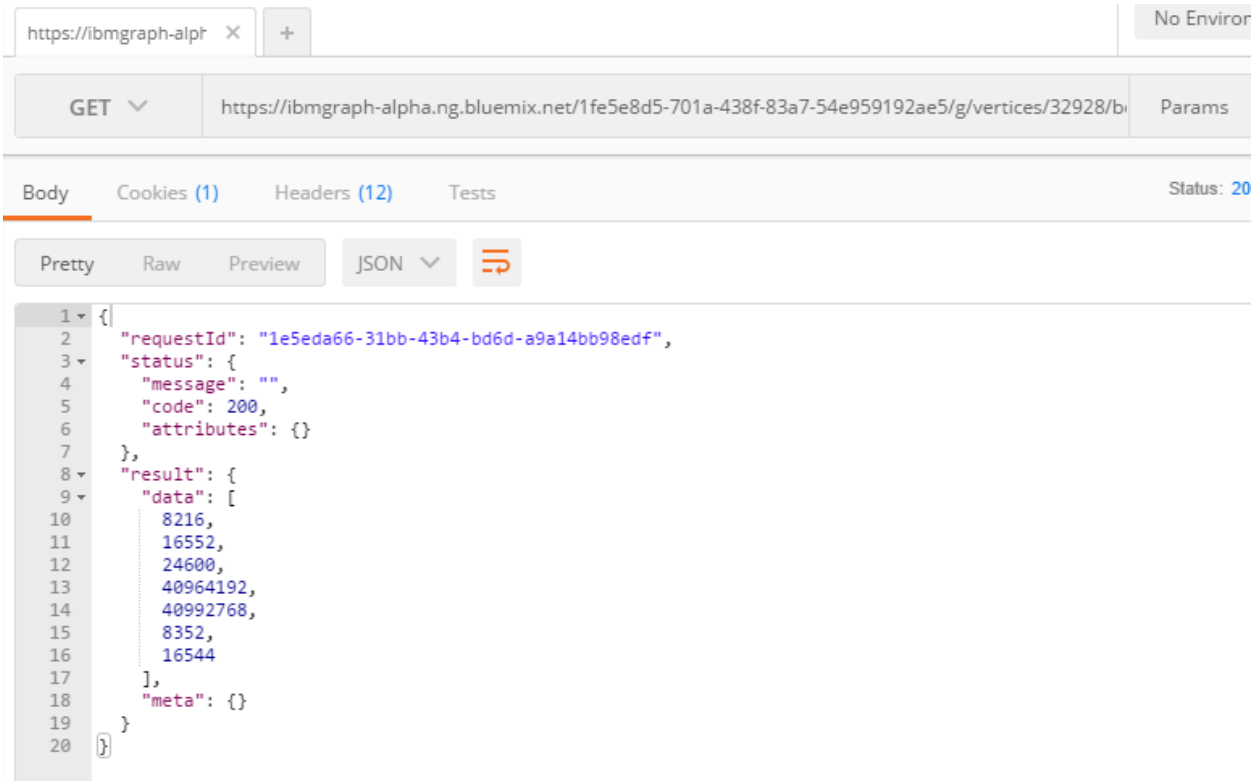
Contains both bands(2) and attendees(5)


```
{
  "requestId": "38b5c9d0-973b-4899-9b32-544076c17595",
  "status": {
    "message": "",
    "code": 200,
    "attributes": {}
  },
  "result": {
    "data": [
      {
        "id": 8216,
        "label": "attendee",
        "type": "vertex",
        "properties": {
          "gender": [
            {
              "id": "35v-6c8-111",
              "value": "female"
            }
          ],
          "name": [
            {
              "id": "3k3-6c8-sl",
              "value": "Kelsey Adams"
            }
          ],
          "age": [
            {
              "id": "3yb-6c8-2dh",
              "value": 30
            }
          ]
        }
      },
      {
        "id": 24600,
        "label": "attendee",
        "type": "vertex",
        "properties": {
          "gender": [
            {
              "id": "b2b-izc-1l1",
              "value": "Kelly Furough"
            }
          ],
          "name": [
            {
              "id": "odyj0-oe068-sl",
              "value": "Chromeo"
            }
          ],
          "age": [
            {
              "id": "odyx8-oe068-2dh",
              "value": 24
            }
          ]
        }
      },{
        "id": 40992768,
        "label": "attendee",
        "type": "vertex",
        "properties": {
          "name": [
            {
              "id": "3kk-6g0-sl",
              "value": "Techno/EDM"
            }
          ],
          "genre": [
            {
              "id": "3ys-6g0-35x",
              "value": "Techno/EDM"
            }
          ]
        }
      },
      {
```

```
        {
          "id": "35v-6c8-1l1",
          "value": "female"
        }
      ],
      "name": [
        {
          "id": "3k3-6c8-sl",
          "value": "Kelsey Adams"
        }
      ],
      "age": [
        {
          "id": "3yb-6c8-2dh",
          "value": 30
        }
      ]
    }
  },{
    "id": 16552,
    "label": "attendee",
    "type": "vertex",
    "properties": {
      "gender": [
        {
          "id": "74l-crs-1l1",
          "value": "male"
        }
      ],
      "name": [
        {
          "id": "7it-crs-sl",
          "value": "Phillip Frye"
        }
        {
          "value": "female"
        }
      ],
      "name": [
        {
          "id": "bgj-izc-sl",
          "value": "Lindsay Bluth"
        }
      ],
      "age": [
        {
          "id": "bur-izc-2dh",
          "value": 25
        }
      ] }
  },
{
    "id": 40964192,
    "label": "attendee",
    "type": "vertex",
    "properties": {
      "gender": [
        {
          "id": "ody4s-oe068-1l1",
          "value": "female"
        }
      ],
      "label": "attendee",
      "type": "vertex",
      "properties": {
        "gender": [
          {
            "id": "1cs8w0-oem80-1l1",
            "value": "female"
          }
        ],
        "name": [
          {
            "id": "1cs9a8-oem80-sl",
            "value": "Katherine Jordan"
          }
        ],
        "age": [
          {
            "id": "1cs9og-oem80-2dh",
            "value": 25
          }
        ]
      }
    },
    {
      "id": 8352,
      "label": "band",
      "type": "vertex",
      "properties": {
        "monthly_listeners": [
          {
            "id": "36c-6g0-3yd",
            "value": "692991"
          }
        ],
        "id": 16544,
        "label": "band",
        "type": "vertex",
        "properties": {
          "monthly_listeners": [
            {
              "id": "74k-crk-3yd",
              "value": "1583612"
            }
          ],
          "name": [
            {
              "id": "7is-crk-sl",
              "value": "Jake Bugg"
            }
          ],
          "genre": [
            {
              "id": "7x0-crk-35x",
              "value": "Folk"
            }
          ]
        }
      }
    ],
    "meta": {}
  }
}
```

],

f. GET command to retrieve IDs of both the in and out connections at a particular vertex. Use ‘bothIds’

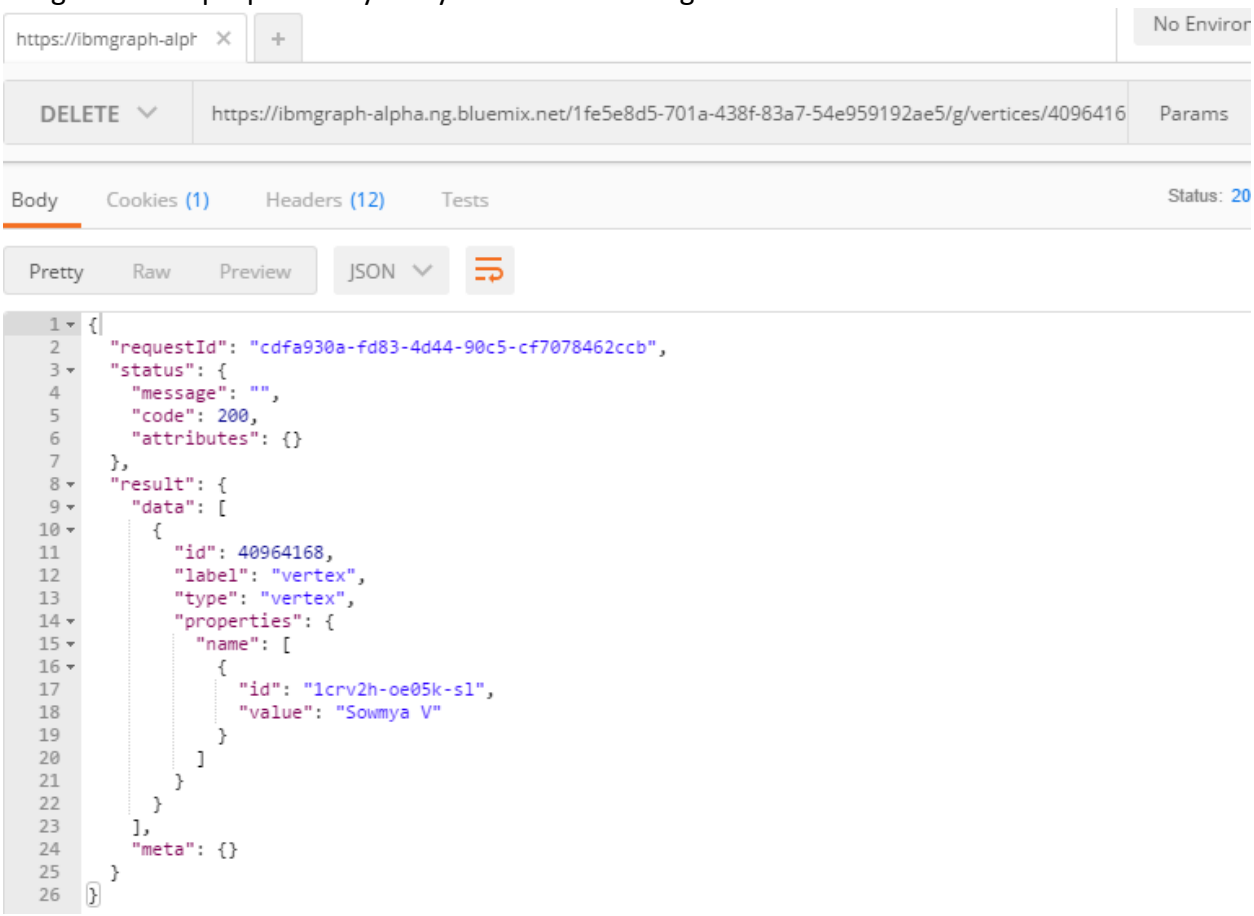


Response:


```
{
 "requestId": "1e5eda66-31bb-43b4-bd6d-a9a14bb98edf",
 "status": {
  "message": "",
  "code": 200,
  "attributes": {}
 },
 "result": {
  "data": [
   8216,
   16552,
   24600,
   40964192,
   40992768,
   8352,
   16544
  ],
  "meta": {}
 }
```

}

g. DELETE properties by a key for a vertex. Using id: 40964168 created earlier.



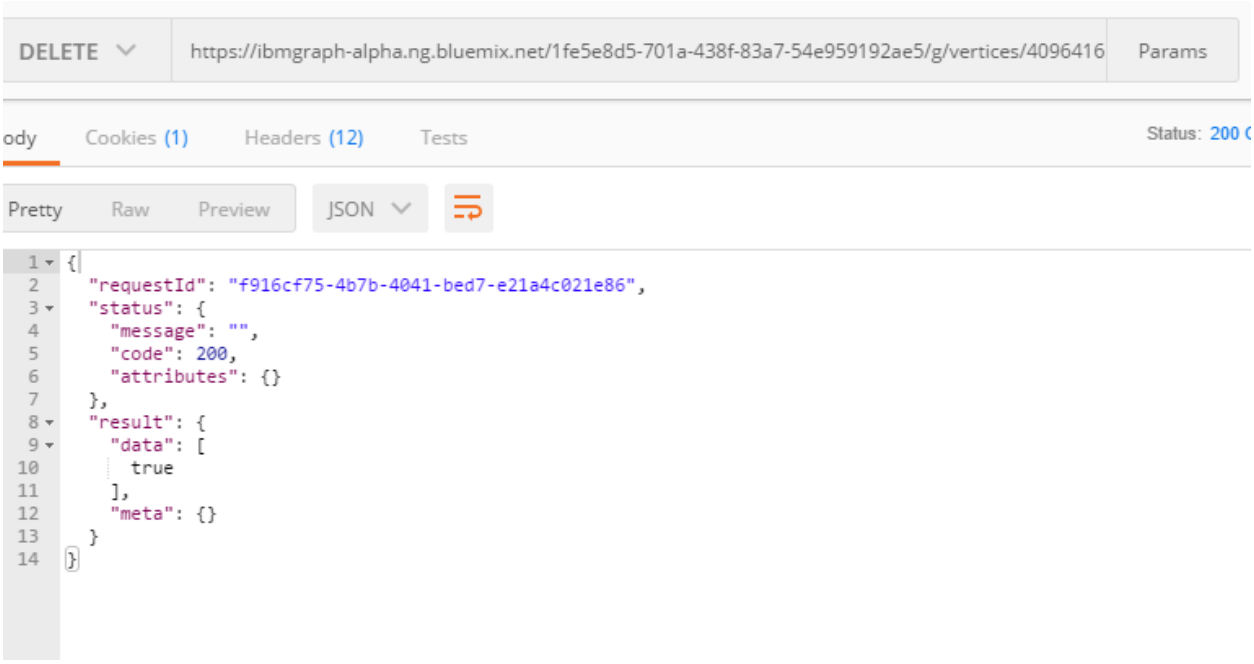
Request: https://ibmgraph-alpha.ng.bluemix.net/1fe5e8d5-701a-438f-83a7-54e959192ae5/g/vertices/40964168?verified

Response:

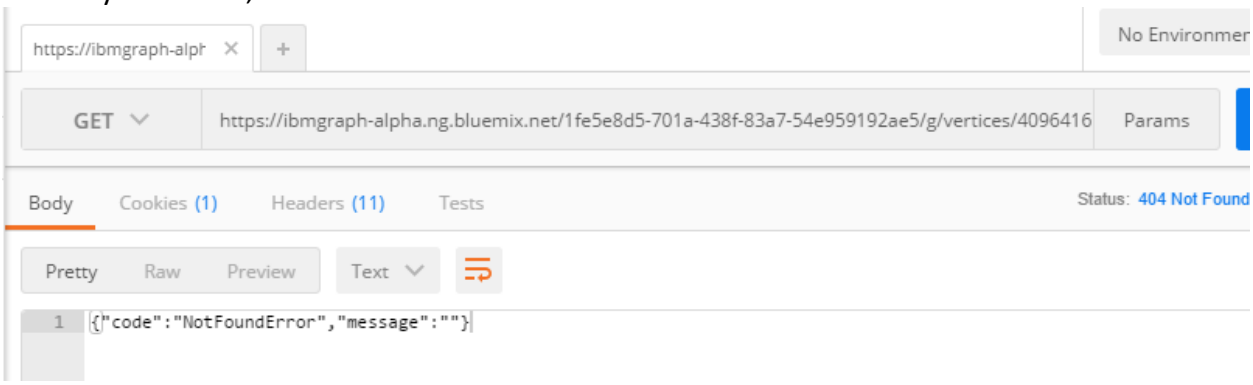
{
"requestId": "cdfa930a-fd83-4d44-90c5-cf7078462ccb",
"status": {
"message": "",
"code": 200,
"attributes": {} },
"result": {
"data": [
{
"id": 40964168,
"label": "vertex",
"type": "vertex",
"properties": {
"name": [{

```
    "id": "1crv2h-oe05k-sl",
    "value": "Sowmya V"
  }       ]      }    }   ],   "meta": {}  }}
```


To DELETE the vertex, send the vertex ID without any property.

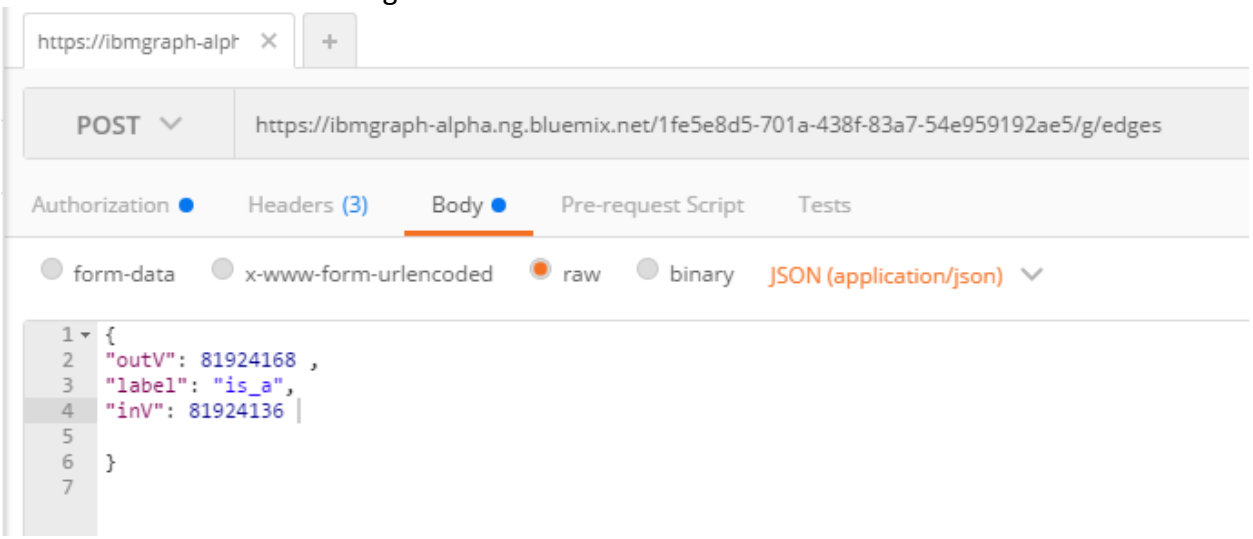


To verify if deleted, use the GET command



4. Edge APIs

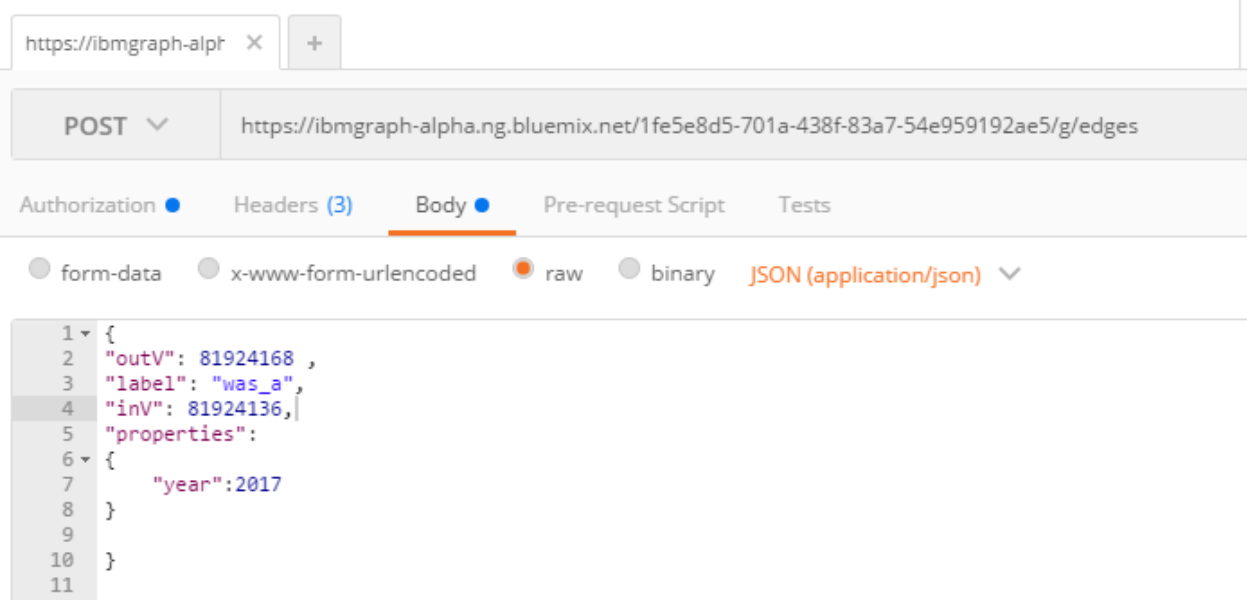
a. POST method to create an edge. iD: 81924168 and 81924136



Response:


```
{
 "requestId": "0eb34e32-2b12-4ec4-b417-afd4565f2cd6",
 "status": {
  "message": "",
  "code": 200,
  "attributes": {}
 },
 "result": {
  "data": [
   {
    "id": "2pjo5l-1crx3c-8sut-1crx2g",
    "label": "is_a",
    "type": "edge",
    "inVLabel": "person",
    "outVLabel": "person",
    "inV": 81924136,
    "outV": 81924168
   }
  ],
  "meta": {}
 }
}
```

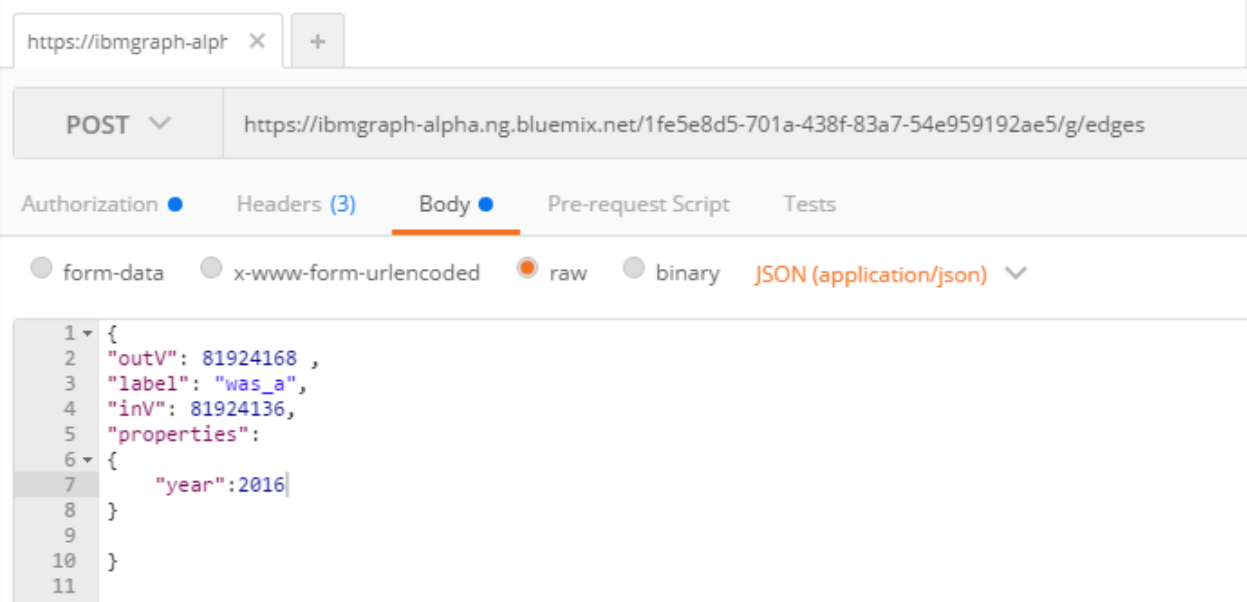
b. Adding a property to the above edge using ‘was_’ label and using POST



Response:


```
{
 "requestId": "f206a92f-ce2d-42db-aa42-ace834647786",
 "status": {
  "message": "",
  "code": 200,
  "attributes": {}
 },
 "result": {
  "data": [
   {
    "id": "3dy4uh-1crx3c-azv9-1crx2g",
    "label": "was_a",
    "type": "edge",
    "inVLabel": "person",
    "outVLabel": "person",
    "inV": 81924136,
    "outV": 81924168,
    "properties": {
     "year": 2017
    }
   }
  ],
  "meta": {}
 }
}
```

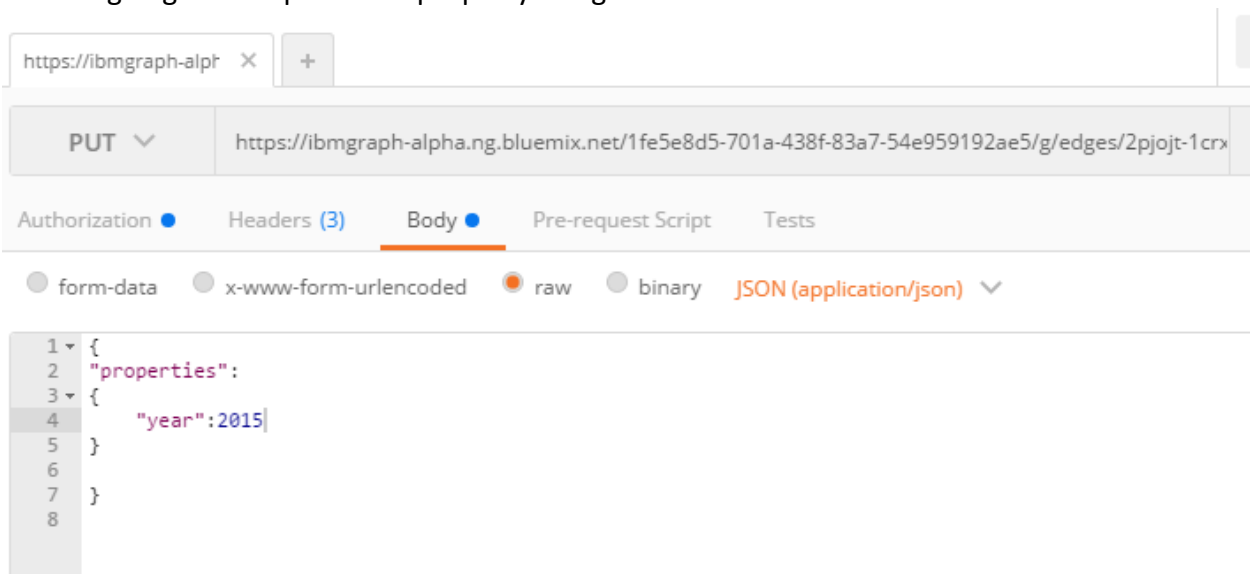
c. Updating the property value to a new value



Response:


```
{
 "requestId": "e44d4fa6-7883-4f90-b18b-f6dd7de64bc7",
 "status": {
  "message": "",
  "code": 200,
  "attributes": {}
 },
 "result": {
  "data": [
   {
    "id": "2pjojt-1crx3c-azv9-1crx2g",
    "label": "was_a",
    "type": "edge",
    "inVLabel": "person",
    "outVLabel": "person",
    "inV": 81924136,
    "outV": 81924168,
    "properties": {
     "year": 2016
    }
   }
  ],
  "meta": {}
 }
}
```

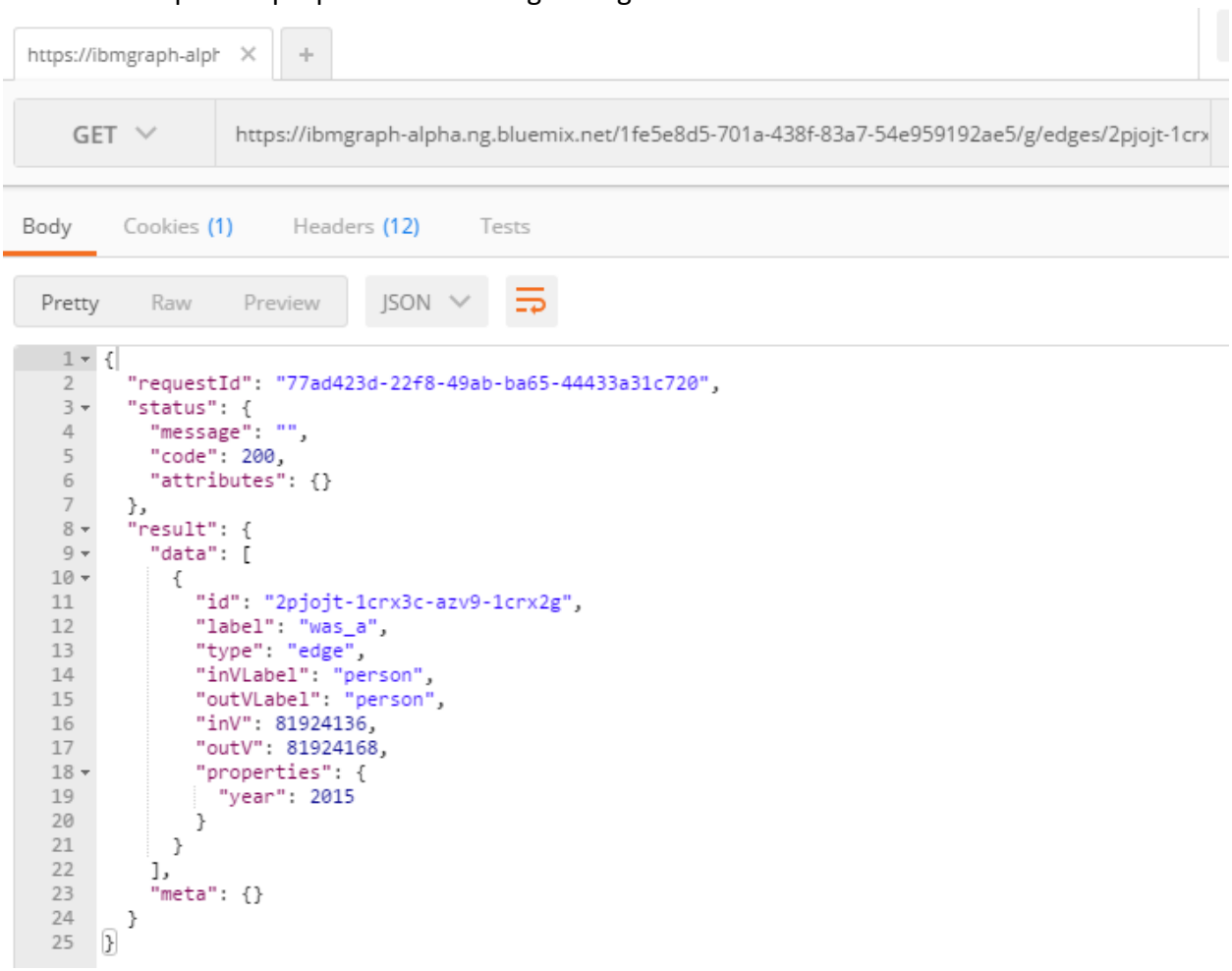
d. Using edge id to update the property using PUT



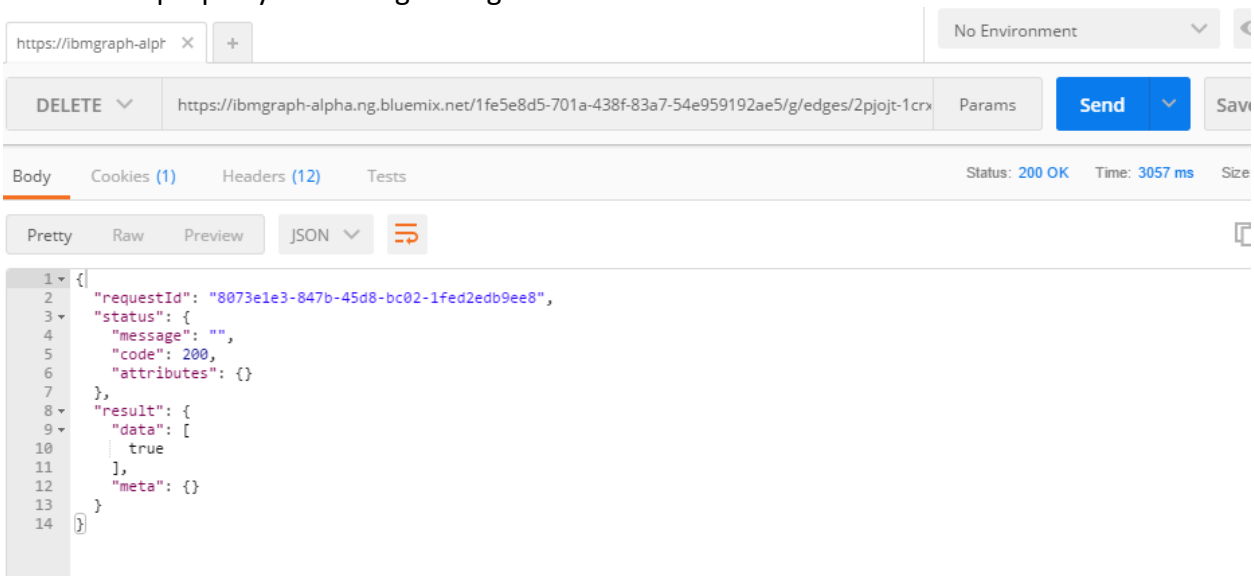
Response:


```
{
 "requestId": "6717d1c6-dab1-4ea4-82d2-5c412ae802d1",
 "status": {
  "message": "",
  "code": 200,
  "attributes": {}
 },
 "result": {
  "data": [
   {
    "id": "2pjojt-1crx3c-azv9-1crx2g",
    "label": "was_a",
    "type": "edge",
    "inVLabel": "person",
    "outVLabel": "person",
    "inV": 81924136,
    "outV": 81924168,
    "properties": {
     "year": 2015
    }
   }
  ],
  "meta": {}
 }
}
```

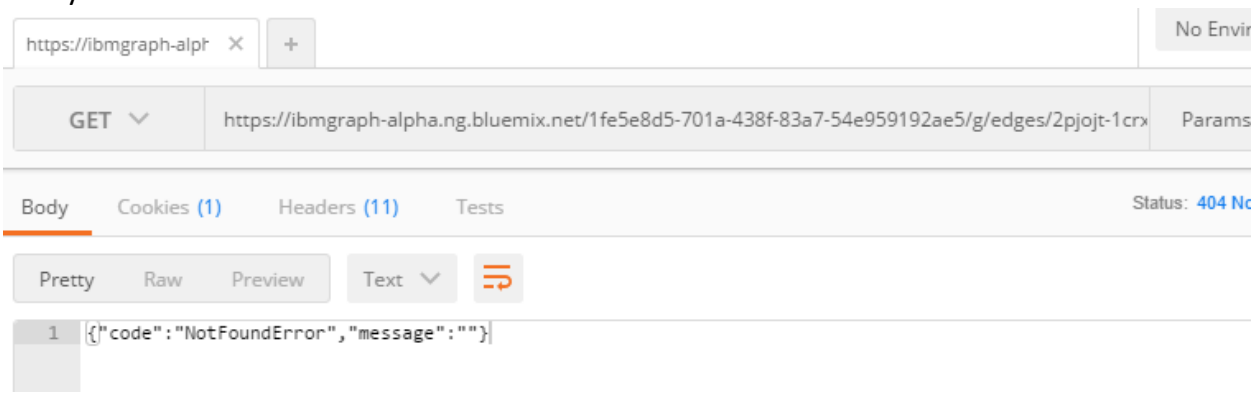
e. GET the updated properties of the edge using its id



DELETE the property of the edge using its ID

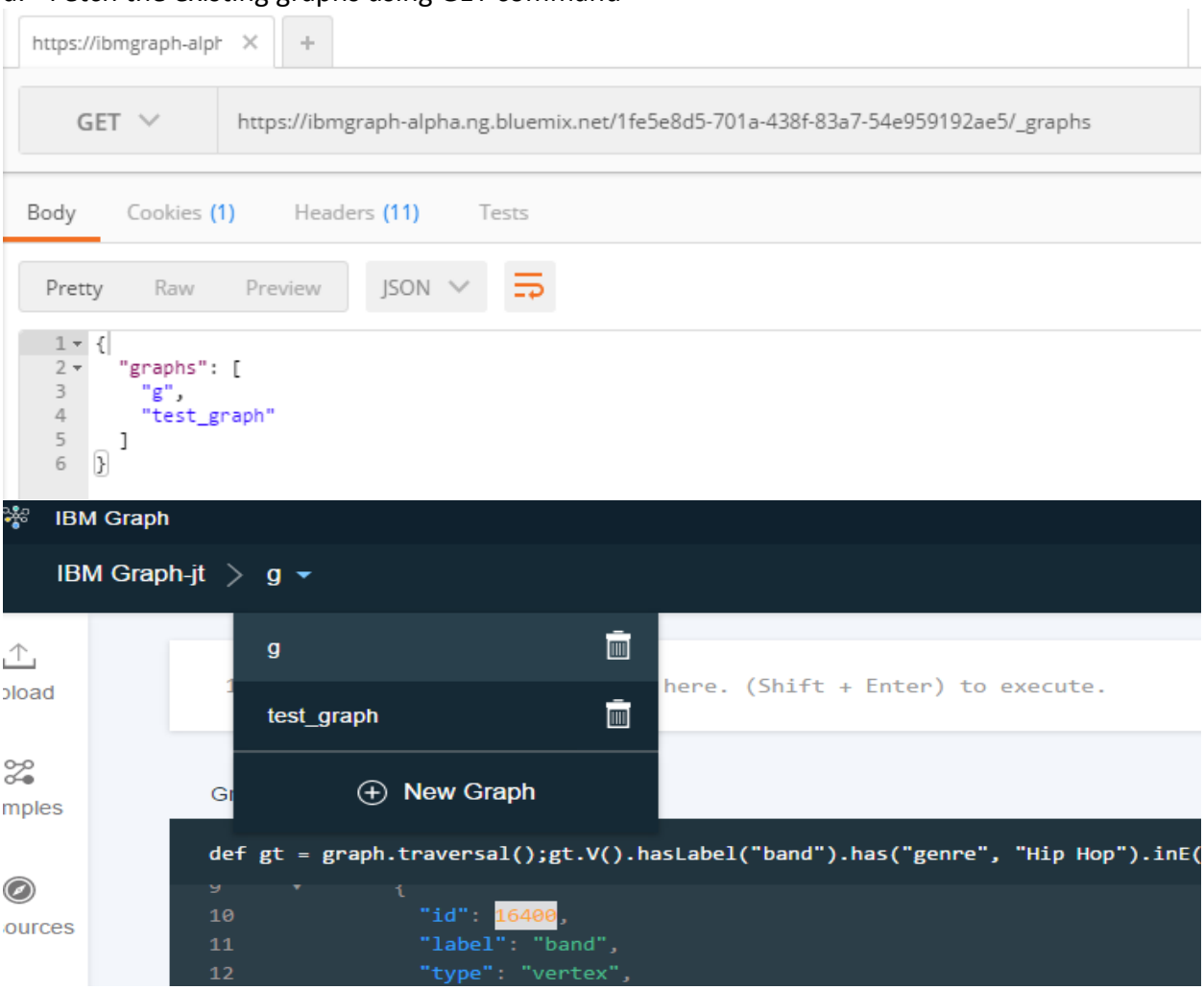


Verify if its deleted

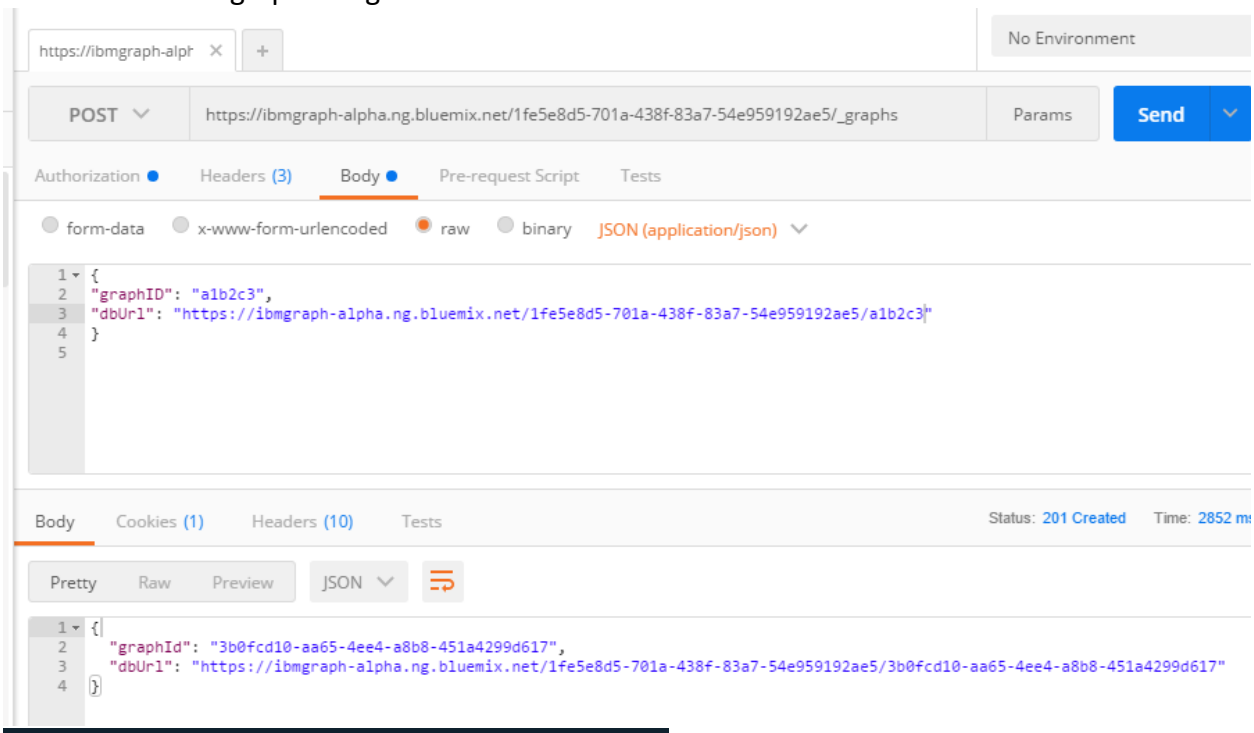


5. Graph APIs

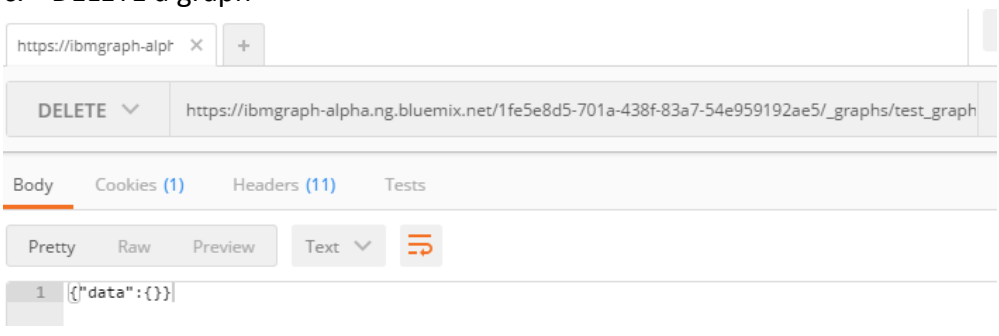
a. Fetch the existing graphs using GET command



b. Create a new graph using POST method



c. DELETE a graph



Graph is deleted