neo4j-rest-client Documentation

Release 2.0.0

Javier de la Rosa

Sep 27, 2017

Contents

synopsis Object-oriented Python library to interact with Neo4j standalone REST server.

The main goal of neo4j-rest-client was to enable Python programmers already using Neo4j locally through python-embedded, to use the Neo4j REST server. So the syntax of neo4j-rest-client's API is fully compatible with python-embedded. However, a new syntax is introduced in order to reach a more pythonic style and to enrich the API with the new features the Neo4j team introduces.

CHAPTER 1

Getting started

The main class is `GraphDatabase`, exactly how in python-embedded:

```
>>> from neo4jrestclient.client import GraphDatabase
```

```
>>> gdb = GraphDatabase("http://localhost:7474/db/data/")
```

If `/db/data/` is not added, neo4j-rest-client will do an extra request in order to know the endpoint for data.

And now we are ready to create nodes and relationhips:

```
>>> alice = gdb.nodes.create(name="Alice", age=30)
```

```
>>> bob = gdb.nodes.create(name="Bob", age=30)
```

```
>>> alice.relationships.create("Knows", bob, since=1980)
```

Although using `labels` is usually easier:

```
>>> people = gdb.labels.create("Person")
```

```
>>> people.add(alice, bob)
```

```
>>> carl = people.create(name="Carl", age=25)
```

Now we can list and filter nodes according to the labels they are associated to:

```
>>> people.filter(Q("age", "gte", 30))
```

CHAPTER 2

Installation

Available through Python Package Index:

```
$ pip install neo4jrestclient
```

Contents:

neo4j-rest-client's documentation

synopsis Object-oriented Python library to interact with Neo4j standalone REST server.

The main goal of neo4j-rest-client was to enable Python programmers already using Neo4j locally through **python-embedded_**, to use the Neo4j REST server. So the syntax of neo4j-rest-client's API is fully compatible with python-embedded. However, a new syntax is introduced in order to reach a more pythonic style and to enrich the API with the new features the Neo4j team introduces.

Installation

Available through Python Package Index:

```
$ pip install neo4jrestclient
```

Or the old way:

```
$ easy_install neo4jrestclient
```

You can also install the development branch:

```
$ pip install git+https://github.com/versae/neo4j-rest-client.git
```

Getting started

The main class is `GraphDatabase`, exactly how in **python-embedded_**:

```
>>> from neo4jrestclient.client import GraphDatabase
```

```
>>> gdb = GraphDatabase("http://localhost:7474/db/data/")
```

If `/db/data/` is not added, neo4j-rest-client will do an extra request in order to know the endpoint for data.

And now we are ready to create nodes and relationhips:

```
>>> alice = gdb.nodes.create(name="Alice", age=30)
```

```
>>> bob = gdb.nodes.create(name="Bob", age=30)
```

```
>>> alice.relationships.create("Knows", bob, since=1980)
```

Although using `labels` is usually easier:

```
>>> people = gdb.labels.create("Person")
```

```
>>> people.add(alice, bob)
```

```
>>> carl = people.create(name="Carl", age=25)
```

Now we can list and filter nodes according to the labels they are associated to:

```
>>> people.filter(Q("age", "gte", 30))
```

Authentication

Authentication-based services like **Heroku_** are also supported by passing extra parameters:

```
>>> url = "http://<instance>.hosted.neo4j.org:7000/db/data/"
```

```
>>> gdb = GraphDatabase(url, username="username", password="password")
```

And when using certificates (both files must be in **PEM_** format):

```
>>> gdb = GraphDatabase(url, username="username", password="password",
                        cert_file='path/to/file.cert',
                        key_file='path/to/file.key')
```

Elements

Nodes

Due to the syntax is fully compatible with neo4j.py, the next lines only show the commands added and its differences.

Creating a node:

```
>>> n = gdb.nodes.create()

# Equivalent to
>>> n = gdb.node()
```

Specify properties for new node:

```
>>> n = gdb.nodes.create(color="Red", width=16, height=32)

# Or
>>> n = gdb.node(color="Red", width=16, height=32)
```

Accessing node by id:

```
>>> n = gdb.node[14]

# Using the identifier or the URL is possible too
>>> n = gdb.nodes.get(14)
```

Accessing properties:

```
>>> value = n['key'] # Get property value

>>> n['key'] = value # Set property value

>>> del n['key']     # Remove property value

# Or the other way
>>> value = n.get('key', 'default') # Support 'default' values

>>> n.set('key', value)

>>> n.delete('key')
```

Besides, a Node object has other attributes:

```
>>> n.properties
{}

>>> n.properties = {'name': 'John'}
{'name': 'John'}

# The URL and the identifier assigned by Neo4j are added too
>>> n.id
14

>>> n.url
'http://localhost:7474/db/data/node/14'
```

Relationships

Create relationship:

```
>>> n1.Knows(n2)

# Or
```

```
>>> n1.relationships.create("Knows", n2) # Usefull when the name of
                                         # relationship is stored in a variable
```

Specify properties for new relationships:

```
>>> n1.Knows(n2, since=123456789, introduced_at="Christmas party")

# It's the same to
>>> n1.relationships.create("Knows", n2, since=123456789,
                                         introduced_at="Christmas party")
```

The creation returns a Relationship object, which has properties, setter and getters like a node:

```
>>> rel = n1.relationships.create("Knows", n2, since=123456789)

>>> rel.start
<Neo4j Node: http://localhost:7474/db/data/node/14>

>>> rel.end
<Neo4j Node: http://localhost:7474/db/data/node/32>

>>> rel.type
'Knows'

>>> rel.properties
{'since': 123456789}
```

Or you can create the relationship using directly from GraphDatabse object:

```
>>> rel = gdb.relationships.create(n1, "Hates", n2)

>>> rel
<Neo4j Relationship: http://localhost:7474/db/data/relationship/66>

>>> rel.start
<Neo4j Node: http://localhost:7474/db/data/node/14>

>>> rel.end
<Neo4j Node: http://localhost:7474/db/data/node/32>
```

Others functions over 'relationships' attribute are possible. Like get all, incoming or outgoing relationships (typed or not):

```
>>> rels = n1.relationships.all()
<Neo4j Iterable: Relationship>
```

In order improve the performance of the 'neo4jrestclient', minimizing the number of HTTP requests that are made, all the functions that should return list of objects like Nodes, Relationships, Paths or Positions, they actually return an Iterable object that extends the Python 'list' type:

```
>>> rels = n1.relationships.all()[:]
[<Neo4j Relationship: http://localhost:7474/db/data/relationship/35843>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/35840>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/35841>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/35842>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/35847>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/35846>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/35845>,
```

```
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/35844>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/11>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/10>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/9>]

>>> rels = n1.relationships.incoming(types=["Knows"])[:]
[<Neo4j Relationship: http://localhost:7474/db/data/relationship/35843>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/35840>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/11>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/10>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/9>]

>>> rels = n1.relationships.outgoing(["Knows", "Loves"])[:]
[<Neo4j Relationship: http://localhost:7474/db/data/relationship/35842>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/35847>]
```

There's a shortcut to access to the list of all relationships:

```
>>> rels = n1.relationships.all()[2]
<Neo4j Relationship: http://localhost:7474/db/data/relationship/47>
```

It's the same to:

```
>>> rels = n1.relationships[2]
<Neo4j Relationship: http://localhost:7474/db/data/relationship/47>
```

And:

```
>>> rels = n1.relationships.get(2)
<Neo4j Relationship: http://localhost:7474/db/data/relationship/47>
```

Labels

Labels are *tags* that you can associate a node to. A node can have more than one label and a label can *have* more than one node.

Add and remove labels to/from a node

For example, we can create a couple of nodes:

```
>>> alice = gdb.nodes.create(name="Alice", age=30)
```

```
>>> bob = gdb.nodes.create(name="Bob", age=30)
```

And then put the label `Person` to both of them:

```
>>> alice.labels.add("Person")
>>> bob.labels.add("Person")
```

You can also add more than one label at the time, or replace all the labels of a node:

```
>>> alice.labels.add(["Person", "Woman"])
>>> bob.labels = ["Person", "Man", "Woman"]
```

And remove labels in the same way:

```
>>> bob.labels.remove("Woman")
```

Although using `labels` from a `GraphDatabase` is usually easier:

```
>>> people = gdb.labels.create("Person")
```

```
>>> people.add(alice, bob)
```

The call for `gdb.labels.create` **does not** actually create the label until the first node is added.

We can also check if a node already has a specific label:

```
>>> "Animal" in bob.labels
False
```

List, get and filter

One common use case for labels is to list all the nodes that are under the same label. The most basic way to do it is by using the `.all()` method once we assign a label to a variable:

```
>>> person = gdb.labels.get("Person")
>>> person.all()
```

Or get those nodes that has a certain pair property name and value:

```
>>> person.get(age=25)
```

Can list and filter nodes according to the labels they are associated to by using the `Q` objects provided by neo4j-rest-client:

```
>>> from neo4jrestclient.query import Q
>>> people.filter(gdb.Q("age", "gte", 30))
```

Indices

The original neo4j.py currently did not provide support for the new index component. However, the current syntax for indexing is now compliant with the python-embedded API, and hopefully more intuitive:

```
>>> i1 =  gdb.nodes.indexes.create("index1")

>>> i2 =  gdb.nodes.indexes.create("index2", type="fulltext", provider="lucene")

>>> gdb.nodes.indexes
{u'index2': <Neo4j Index: http://localhost:7474/db/data/index/node/index2>,
 u'index1': <Neo4j Index: http://localhost:7474/db/data/index/node/index1>}

>>> gdb.nodes.indexes.get("index1")
<Neo4j Index: http://localhost:7474/db/data/index/node/index1>
```

You can query and add elements to the index like a 3-dimensional array or using the convenience methods:

```
>>> i1["key"]["value"]
[]

>>> i1.get("key")["value"]
[]

>>> i1.get("key", "value")
[]

>>> i1["key"]["value"] = n1

>>> i1.add("key", "value", n2)

>>> i1["key"]["value"][:]
[<Neo4j Node: http://localhost:7474/db/data/node/1>,
 <Neo4j Node: http://localhost:7474/db/data/node/2>]
```

Advanced queries are also supported if the index is created with the type *fulltext* (*lucene* is the default provider) by entering a Lucene query:

```
>>> n1 = gdb.nodes.create(name="John Doe", place="Texas")

>>> n2 = gdb.nodes.create(name="Michael Donald", place="Tijuana")

>>> i1 = gdb.nodes.indexes.create(name="do", type="fulltext")

>>> i1["surnames"]["doe"] = n1

>>> i1["places"]["Texas"] = n1

>>> i1["surnames"]["donald"] = n2

>>> i1["places"]["Tijuana"] = n2

>>> i1.query("surnames", "do*")[:]
[<Neo4j Node: http://localhost:7474/db/data/node/295>,
 <Neo4j Node: http://localhost:7474/db/data/node/296>]
```

...or by using the DSL described by lucene-querybuilder to support boolean operations and nested queries:

```
>>> i1.query(Q('surnames','do*') & Q('places','Tijuana'))[:]
[<Neo4j Node: http://localhost:7474/db/data/node/295>]
```

Deleting nodes from an index:

```
>>> i1.delete("key", "values", n1)

>>> i1.delete("key", None, n2)
```

And in order to work with indexes of relationships the instructions are the same:

```
>>> i3 =  gdb.relationships.indexes.create("index3")
```

For deleting an index just call 'delete' with no arguments:

```
>>> i3.delete()
```

Queries

Since the Cypher plugin is not a plugin anymore, neo4j-rest-client is able to run queries and returns the results properly formatted:

```
>>> q = """start n=node(*) return n"""
```

```
>>> result = gdb.query(q=q)
```

Returned types

This way to run a query will return the results as RAW, i.e., in the same way the REST interface get them. However, you can always use a `returns` parameter in order to perform custom castings:

```
>>> q = """start n=node(*) match n-[r]-() return n, n.name, r"""
```

```
>>> results = gdb.query(q, returns=(client.Node, unicode, client.Relationship))
```

```
>>> results[0]
[<Neo4j Node: http://localhost:7474/db/data/node/14>,
u'John Doe',
<Neo4j Relationship: http://localhost:7474/db/data/relationship/47>]
```

Or pass a custom function:

```
>>> is_john_doe = lambda x: x == "John Doe"
```

```
>>> results = gdb.query(q, returns=(client.Node, is_john_doe, client.Relationship))
>>> results[0]
[<Neo4j Node: http://localhost:7474/db/data/node/14>,
True,
<Neo4j Relationship: http://localhost:7474/db/data/relationship/47>]
```

If the length of the elements is greater than the casting functions passed through the `returns` parameter, the *RAW* will be used instead of raising an exception.

Sometimes query results include lists, as it happens when using `COLLECT` or other collection functions, `neo4j-rest-client` is able to handle these cases by passing lists or tuples in the *results* list. Usually these lists contain items of the same type, so passing only one casting function is enough, as all the items are treated the same way.

```
>>> a = gdb.nodes.create()
>>> [a.relationships.create("rels", gdb.nodes.create()) for x in range(3)]
[<Neo4j Relationship: http://localhost:7474/db/data/relationship/43>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/44>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/45>]
>>> q = """match (a)--(b) with a, collect(b) as bs return a, bs limit 1"""
```

```
>>> gdb.query(q, returns=(client.Node, [client.Node, ]))[0]
[<Neo4j Node: http://localhost:7474/db/data/node/31>,
 [<Neo4j Node: http://localhost:7474/db/data/node/29>,
  <Neo4j Node: http://localhost:7474/db/data/node/28>,
  <Neo4j Node: http://localhost:7474/db/data/node/30>]]
```

```
>>> gdb.query(q, returns=(client.Node, (client.Node, )))[0]
[<Neo4j Node: http://localhost:7474/db/data/node/31>,
 (<Neo4j Node: http://localhost:7474/db/data/node/29>,
  <Neo4j Node: http://localhost:7474/db/data/node/28>,
  <Neo4j Node: http://localhost:7474/db/data/node/30>)]
```

```
>>> gdb.query(query, returns=[client.Node, client.Iterable(client.Node)])[0]
[<Neo4j Node: http://localhost:7474/db/data/node/3672>,
 <listiterator at 0x7f6958c6ff50>]
```

However, if you know in advance how many elements are going to be returned as the result of a collection function, you can always customize the casting functions:

```
>>> gdb.query(q, returns=(client.Node, (client.Node, lambda x: x["data"], client.Node␣
↪)))[0]
[<Neo4j Node: http://localhost:7474/db/data/node/31>,
 (<Neo4j Node: http://localhost:7474/db/data/node/29>,
  {u'tag': u'tag1'},
  <Neo4j Node: http://localhost:7474/db/data/node/30>)]
```

Query statistics

Extra information about the execution of a each query is stored in the property *stats*.

```
>>> query = "MATCH (n)--() RETURN n LIMIT 5"
>>> results = gdb.query(query, data_contents=True)
>>> results.stats
{u'constraints_added': 0,
 u'constraints_removed': 0,
 u'contains_updates': False,
 u'indexes_added': 0,
 u'indexes_removed': 0,
 u'labels_added': 0,
 u'labels_removed': 0,
 u'nodes_created': 0,
 u'nodes_deleted': 0,
 u'properties_set': 0,
 u'relationship_deleted': 0,
 u'relationships_created': 0}
```

Graph and row data contents

The Neo4j REST API is able to provide the results of a query in other two formats that might be useful when redering. To enable this option (which is the default only when running inside a IPython Notebook), you might pass an extra parameter to the query, *data_contents*. If set to *True*, it will populate the properties *.rows* as a list of rows, and *.graph* as a graph representation of the result.

```
>>> query = "MATCH (n)--() RETURN n LIMIT 5"
>>> results = gdb.query(query, data_contents=True)
>>> results.rows
[[{u'name': u'M\xedchael Doe', u'place': u'T\xedjuana'}],
 [{u'name': u'J\xf3hn Doe', u'place': u'Texa\u015b'}],
 [{u'name': u'Rose 0'}],
 [{u'name': u'William 0'}],
```

```
 [{u'name': u'Rose 1'}]]
>>> results.graph
  [{u'nodes': [{u'id': u'3',
    u'labels': [],
    u'properties': {u'name': u'M\xedchael Doe', u'place': u'T\xedjuana'}}],
  u'relationships': []},
 {u'nodes': [{u'id': u'2',
    u'labels': [],
    u'properties': {u'name': u'J\xf3hn Doe', u'place': u'Texa\u015b'}}],
  u'relationships': []},
 {u'nodes': [{u'id': u'45',
    u'labels': [],
    u'properties': {u'name': u'Rose 0'}}],
  u'relationships': []},
 {u'nodes': [{u'id': u'44',
    u'labels': [],
    u'properties': {u'name': u'William 0'}}],
  u'relationships': []},
 {u'nodes': [{u'id': u'47',
    u'labels': [],
    u'properties': {u'name': u'Rose 1'}}],
  u'relationships': []}]
```

If only one of the represenations is needed, *data_contents* can be either *constants.DATA_ROWS* or *constants.DATA_GRAPH*.

Filters

On top of Queries feature, there are some filtering helpers for nodes, relationships and both indices. First thing you need is to define *Q* objects:

```
>>> from neo4jrestclient.query import Q
```

```
>>> Q("name", istartswith="william")
```

Once a lookup is defined, you may call the *filter* method over all the nodes or the relationships:

```
>>> gdb.nodes.filter(lookup)
[<Neo4j Node: http://localhost:7474/db/data/node/14>]
```

Or just a list of elements identifiers, *Node*'s or *Relationship*'s:

```
>>> nodes = []
```

```
>>> for i in range(2):
...:     nodes.append(gdb.nodes.create(name="William %s" % i))
```

```
>>> lookup = Q("name", istartswith="william")
```

```
>>> williams = gdb.nodes.filter(lookup, start=nodes)
```

Lookups

The syntax for lookups is very similar to the one used in Django, but does include other options:

Q(property_name, lookup=match)

The next list shows all the current lookups supported:

- *exact*, performs exact string comparation.
- *iexact*, performs exact string comparation, case insesitive.
- *contains*, checks if the property is contained in the string passed.
- *icontains*, as *contains* but case insesitive.
- *startswith*, checks if the property starts with the string passed.
- *istartswith*, as *startswith* but case insesitive.
- *endswith*, checks if the property ends with the string passed.
- *iendswith*, as *endswith* but case insesitive.
- *regex*, performs regular expression matching agains the string passed.
- *iregex*, as *regex* but case insesitive.
- *gt*, check if the property is greater than the value passed.
- *gte*, check if the property is greater than or equal to the value passed.
- *lt*, check if the property is lower than the value passed.
- *lte*, check if the property is lower than or equal to the value passed.
- *in*, , check if the property is in a list of elements passed.
- *inrange*,‘an alias for *in*.
- *isnull*, checks if the property is null, passing *True*, or not, passing *False*.
- *eq*, performs equal comparations.
- *equals*, an alias *eq*.
- *neq*, performs not equal comparations.
- *notequals*, an alias *neq*.

Also, in order to be compliant with Cypher syntax prior to Neo4j 2.0, you can add a *nullable* parameter to set if the lookup must be don using *!* or *?*. By default, all lookups are nullable. After Neo4j 2.0, *nullable* options is no longer supported since the operators *!* and *?* are not longer in Neo4j.

```
>>> lookup = Q("name", istartswith="william", nullable=True)
```

```
>>> lookup
n.`name`! =~ (?i)william.*
```

```
>>> lookup = Q("name", istartswith="william", nullable=False)
```

```
>>> lookup
n.`name`? =~ (?i)william.*
```

```
>>> lookup = Q("name", istartswith="william", nullable=None)
```

```
>>> lookup
n.`name` =~ (?i)william.*
```

There is support for complex lookups as well:

```
>>> lookups = (Q("name", exact="James")
   ...:  & (Q("surname", startswith="Smith") & ~Q("surname", endswith="e")))
( n.`name`! = James AND ( n.`surname`! =~ Smith.* AND NOT ( n.`surname`! =~ .*1 ) ) )
```

Ordering

There is an feature to set the order by which the elements will be returned, using the Cypher option *order by*. The syntax is a tuple: the first element is the property name to order by, the second one the type of ordering, *constants.ASC* for ascending, and *constants.DESC* for descending. A set of orderings can be used:

```
>>> gdb.nodes.filter(lookup).order_by("code", constants.DESC)
```

Indices

Indices also implement the *filter* method, so you can use an index as a start, or just invoke the method to filter the elements:

```
>>> old_loves = gdb.relationships.filter(lookup, start=index)
```

```
>>> old_loves = gdb.relationships.filter(lookup, start=index["since"])
```

So, the next would be the same:

```
>>> old_loves = index.filter(lookup)
```

```
>>> old_loves = index.filter(lookup, key="since")
```

```
>>> old_loves = index["since"].filter(lookup)
```

However, it is not possible yet to pass a value for the index using the common dictionary syntax. Instead, you may use the *value* parameter:

```
>>> old_loves = index.filter(lookup, key="since", value=1990)
```

Slicing

In addition, all filters implement lazy slicing, so the query is not run until the results are going to be retrieved. However, there is not still support for transactions:

```
>>> lookup = Q("name", istartswith="william")
```

```
>>> results = gdb.nodes.filter(lookup)  # Not query executed yet
```

```
>>> len(restuls)  # Here the query is executed
12
```

If the elements of the filter have been already retrieved from the server, the slicing is then run against the local version. If not, the *slice* is transformed into *limit* and *skip* options before doing the request.

```
>>> results = gdb.nodes.filter(lookup)  # Not query executed yet
```

```
>>> restuls[1:2]  # The Cypher query is limited using limit and skip
[<Neo4j Node: http://localhost:7474/db/data/node/14>]
```

```
>>> len(results)  # The Cypher query is sent again to the server
12
```

Traversals

The traversals framework is supported too with the same syntax of neo4j.py, but with some added issues.

Regular way:

```
>>> n1.relationships.create("Knows", n2, since=1970)
<Neo4j Relationship: http://localhost:7474/db/data/relationship/36009>

>>> class TraversalClass(gdb.Traversal):
   ...:     types = [
   ...:         client.All.Knows,
   ...:     ]
   ...:

>>> [traversal for traversal in TraversalClass(n1)]
[<Neo4j Node: http://localhost:7474/db/data/node/15880>]
```

Added way (the types of relationships are 'All', 'Incoming', 'Outgoing'):

```
>>> n1.relationships.create("Knows", n2, since=1970)
<Neo4j Relationship: http://localhost:7474/db/data/relationship/36009>

>>> n1.traverse(types=[client.All.Knows])[:]
[<Neo4j Node: http://localhost:7474/db/data/node/15880>]
```

For getting a paginated traversal is only needed one of the next parameters: 'paginated' to enable the pagination, 'page_size' to set the size of returned page, and 'time_out' to establish the lease time that the server will wait for. After set any of this parameters, the traversal call will return an iterable object of traversals called 'PaginatedTraversal':

```
>>> pages = n1.traverse(types=[client.All.Knows], stop=stop, page_size=5)

>>> pages
<PaginatedTraversal object at 0x25a5150>

>>> [n for n in [traversal for traversal in pages]]
[<Neo4j Node: http://localhost:7474/db/data/node/15880>]
```

Extensions

The server plugins are supported as extensions of GraphDatabase, Node or Relationship objects:

```
>>> gdb.extensions
{u'GetAll': <Neo4j ExtensionModule: [u'get_all_nodes',
                                     u'getAllRelationships']>}
>>> gdb.extensions.GetAll
<Neo4j ExtensionModule: [u'get_all_nodes', u'getAllRelationships']>

>>> gdb.extensions.GetAll.getAllRelationships()[:]

[<Neo4j Relationship: http://localhost:7474/db/data/relationship/0>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/1>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/2>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/3>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/4>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/5>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/6>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/7>,
 <Neo4j Relationship: http://localhost:7474/db/data/relationship/8>]
```

An example using extensions over nodes:

```
>>> n1 = gdb.nodes.get(0)

>>> n1.extensions
{u'DepthTwo': <Neo4j ExtensionModule: [u'nodesOnDepthTwo',
                                       u'relationshipsOnDepthTwo',
                                       u'pathsOnDepthTwo']>,
 u'ShortestPath': <Neo4j ExtensionModule: [u'shortestPath']>}

>>> n2 = gdb.nodes.get(1)

>>> n1.relationships.create("Kwnos", n2)
<Neo4j Relationship: http://localhost:7474/db/data/relationship/36>

>>> n1.extensions.ShortestPath
<Neo4j ExtensionModule: [u'shortestPath']>

>>> n1.extensions.ShortestPath.shortestPath.parameters

[{u'description': u'The node to find the shortest path to.',
  u'name': u'target',
  u'optional': False,
  u'type': u'node'},
 {u'description': u'The relationship types to follow when searching for ...',
  u'name': u'types',
  u'optional': True,
  u'type': u'strings'},
 {u'description': u'The maximum path length to search for, ...',
  u'name': u'depth',
  u'optional': True,
  u'type': u'integer'}]
```

Transactions and Batch

Transactions in Cypher

Neo4j provides, since its version 2.0.0, a transactional endpoint for Cypher queries. That feature is wrapped in 'neo4jrestclient' in the `gdb.transaction()` method. But for backwards compatibility issues (there were *transactions* before), you need to add an extra parameter `for_query=True` in order to enable it.

Object-based

The easiest way to use a transaction is by creating a `tx` object:

```
>>> tx = gdb.transaction(for_query=True)
```

While the transaction is alive, a property `finished` is set to `False`. The property `expires` has a string with the date sent by the server.

```
>>> tx.finished
False
>>> tx.expires
"Sun, 08 Dec 2013 15:05:52 +0000"
```

Now, regular Cypher queries can be added to the transaction and executed or in server:

```
>>> tx.append("CREATE (a) RETURN a", returns=client.Node)
>>> tx.append("CREATE (b) RETURN b", params={}  )
>>> results = tx.execute()
>>> len(results) == 2
True
```

Both methods, `execute()` and *commit()*, return a `QuerySequence` with the results of the queries sent to the server. You can now perform any check on the returned objects, and if there is something wrong, rollback the transaction and restore the previous state of the database.

```
>>> tx.rollback()
>>> len(results)
0
```

Or you can commit and get the remaining results returned by server:

```
>>> tx.append("MERGE (c:Person {name:'Carol'})")
>>> tx.append("MERGE (d:Person {name:'Dave'})")
>>> results = tx.commit()
>>> len(results) == 2
True
```

After `commit()` or `rollback()`, the transaction is destroyed and no queries can be appended.

Inside a `with` statement

For your convinience and wider control of the logic of your application, transactions can be written inside a `with` statement. This way, you don't need a `tx` object and can use the regular syntax for queries. Each independent query is executed in the transaction, so you have the returned values and can operate with them:

```
>>> q = "start n=node(*) match n-[r:`{rel}`]-() return n, n.name, r, r.since"
>>> params = {"rel": "Knows"}
>>> returns = (client.Node, str, client.Relationship)
>>> with self.gdb.transaction(for_query=True) as tx:
...     self.gdb.query("MERGE (c:Person {name:'Carol'})")
...     results = self.gdb.query(q, params=params, returns=returns)
...     node = results[0][0]
...     if node["name"] == "Carol":
...         tx.rollback()
```

Batch-based Transactions

The transaction support for regular opertations, like CRUD and indexing on nodes and relationships, is based on the REST endpoint for batch operations, therefore there is some limitations because **it is not** a real transaction. When a batch of operations is sent to the server, Neo4j executes it in a transaction, but there is no option to rollback and recover a previous status of the database. In this sense, batch-emulated transactions for operations on creation, edition and deletion of elements are useful, but you won't be able to perform checks on the elements modified until the batch is sent to the server and the transaction is commited.

Deletion

Basic usage for deletion:

```
>>> n = gdb.nodes.create()
>>> n["age"] = 25
>>> n["place"] = "Houston"
>>> n.properties
{'age': 25, 'place': 'Houston'}
>>> with gdb.transaction():
...     n.delete("age")
...
>>> n.properties
{u'place': u'Houston'}
```

Creation

Apart from update or deletion of properties, there is also creation. In this case, the object just created is returned through a `TransactionOperationProxy` object, which is automatically converted in the proper object when the transaction ends. This is the second part of the commit process and a parameter in the transaction, `commit` can be added to avoid the commit:

```
>>> n1 = gdb.nodes.create()
>>> n2 = gdb.nodes.create()
>>> with gdb.transaction() as tx:
...     for i in range(1, 11):
...         n1.relationships.create("relation_%s" % i, n2)
...
>>> len(n1.relationships) != 0
True
```

Auto-update and auto-commit

When a transaction is performed, the values of the properties of the objects are updated automatically. However, this can be controled by hand adding a parameter in the transaction:

```
>>> n = gdb.nodes.create()
>>> n["age"] = 25
>>> with gdb.transaction(update=False):
   ....:         n.delete("age")
   ....:
>>> n.properties
{'age': 25}
>>> n.update()
>>> n.properties
{}
```

You can also set `commit=False` and commit manually after the `with` block is over:

```
>>> with gdb.transaction(commit=False) as tx:
   ....:         n.delete("age")
   ....:
>>> n.properties
{'age': 25}
>>> tx.commit()
>>> n.properties
{}
```

The `commit` method of the transaction object returns *True* if there's no any fail. Otherwise, it returns 'None':

```
>>> tx.commit()
True
>>> len(n1.relationships)
10
```

Globals and nesting

In order to avoid the need of setting the transaction variable, 'neo4jrestclient' uses a global variable to handle all the transactions. The name of the variable can be changed using de options:

```
>>> client.options.TX_NAME = "_tx"  # Default value
```

And this behaviour can be disabled adding the right param in the transaction: `using_globals`. Even is possible (although not very recommendable) to handle different transactions in the same time and control when they are commited. There are many ways to set the transaction of a intruction (operation):

```
>>> n = gdb.nodes.create()
>>> n["age"] = 25
>>> n["name"] = "John"
>>> n["place"] = "Houston"
>>> with gdb.transaction(commit=False, using_globals=False) as tx1, \
   ....:      gdb.transaction(commit=False, using_globals=False) as tx2:
   ....:         n.delete("age", tx=tx1)
   ....:     n["name"] = tx2("Jonathan")
   ....:     n["place", tx2] = "Toronto"
   ....:
```

```
>>> "age" in n.properties
True
```

```
>>> tx1.commit()
True
>>> "age" in n.properties
False
>>> n["name"] == "John"
True
>>> n["place"] == "Houston"
True
```

```
>>> tx2.commit()
True
>>> n["name"] == "John"
False
>>> n["place"] == "Houston"
False
```

Options

There some global options available in neo4j-rest-client that change internal behaviours.

CACHE

If `CACHE` is `True`, a `.cache` directory is created and the future requests to the same URL will be taken from cache:

```
>>> neo4jrestclient.options.CACHE = False  # Default
```

The location and name of the `.cache` can be changed by modifying the option `CACHE_STORE`.

```
>>> neo4jrestclient.options.CACHE_STORE = "/path/to/cache"
```

The neo4j-rest-client's cache is implemented using CacheControl for requests, so you shouldn't have any problem using your own custom cache (e.g `LocMemCache` from Django):

```
>>> neo4jrestclient.options.CACHE_STORE = LocMemCache()
```

DEBUG

If `DEBUG` is `True`, `httplib.HTTPConnection.debuglevel` is set to `1`, and requests enables its logger:

```
>>> neo4jrestclient.options.DEBUG = False   # Default
```

SMART_DATES

There is experimental support for `date`, `time`, and `datetime` objects since Neo4j does not support natively (yet) those data types. What neo4j-rest-client does is to use a specific format to store them as strings, and convert them from Python objects to string (and viceversa) when needed.

To enable this feature you can set `SMART_DATES` to `True`:

```
>>> neo4jrestclient.options.SMART_DATES = False  # Default
```

The format in which `date`, `time`, and `datetime` objects are stored can be changed by modifying the next values:

```
>>> neo4jrestclient.options.DATE_FORMAT = "%Y-%m-%d"
>>> neo4jrestclient.options.TIME_FORMAT = "%H:%M:%S.%f"
>>> neo4jrestclient.options.DATETIME_FORMAT = "%Y-%m-%dT%H:%M:%S.%f"
```

SMART_ERRORS

And `SMART_ERRORS`, set to `False` by default. In case of `True`, the standard HTTP errors will be replaced by more pythonic errors (i.e. `KeyError` instead of `NotFoundError` in some cases):

```
>>> neo4jrestclient.options.SMART_ERRORS = False  # Default
```

TX_NAME

It is extremely weird to have the need to change this option, but in case that you need a different name for the memory variable that will store in progress transactions (aka batch operations), you can do that with `TX_NAME`:

```
>>> neo4jrestclient.options.TX_NAME = "_tx"  # Default
```

URI_REWRITES

When using transactional Cypher endpoint behind SSL, Neo4j fails to returns the right URI, (mistakenly produces *http* instead of *https*, and *localhost:0* instead of the actual URI). By using this option *neo4jrestclient* can rewrite those URIs to the right one. It is disabled by default (set to *None*)

For example, to replace *"http://localhost:0/"* with *"https://db.host.com:8000/"*, you will need a dictionary like:

```
>>> neo4jrestclient.options.URI_REWRITES = {
    {"http://localhost:0/": "https://db.host.com:8000/",
}
```

If the order of the replace operation is important, a *SortedDict* can be used.

VERIFY_SSL

This option is used to set to `True` the verification of SSL certificates. By default is set to `False`

```
>>> neo4jrestclient.options.VERIFY_SSL = False  # Default
```

Changes

2.1.1 (2015-11-20)

- Add dockerized travis

- Fix #119. Allow labels create nodes directly
- Fix #122. Problem clearing labels of a node. Fix also a cache issue. Update travis to run python 3.4 and Neo4j verions
- Add more serialization support for extensions

2.1.0 (2014-11-09)

- Fix pip install error
- Remove lucene-querybuilder as dependency
- Add stats for query execution
- Add support for resultDataContents param in the transactional Cypher endpoint
- Fix #116. Although the old reference is still kept, the object does not exist in server and will fail
- Add Neo4j 2.1.5 to travis
- Fixes issues #109 and #114, related to a memory leak in query transactions
- Fix #113. Add a way to cast query results from collection functions in Cypher
- Change .iteritems to items for Python 3 compatibility
- Change to enterprise for testing
- Fix URI_REWRITES option. Remove testing for 1.7.2, and add 2.1.4.
- Fix the download script for Neo4j, neo4j-clean-remote-db-addon no longer used.
- Add uri rewrites as a work around neo4j issue #2985
- Some pruning bugs
- Labels url is stored in node_labels settings key
- Fixed bug in prune function: it didn't return the self object
- Fixed bug for prunning with JS code: added the case in traverse method for prunning based on an arbitrary Javascript code.

2.0.4 (2014-06-20)

- Typos
- Bugfixes
- Drop support for 1.6 branch

2.0.3 (2014-05-16)

- Update travis to test Neo4j versions 1.9.7 and 2.0.3
- Fix #104. Keep backwards compatibility for 'nullable' prior 2.0 It will be deprecated for Neo4j>=2.0.0
- Update Q class for nullable=True
- Fix un/pickling extenions
- Refactorize get auth information from the connection URL

- Update queries.rst (typo)
- Fix the lazy loading of extensions

2.0.2 (2014-04-04)

- Add Pickle support for GraphDatabase objects
- Add small control to change display property in IPython
- Add a new parameter to auto_execute transactions in one single request
- Fix auto transaction in Cypher queries for Neo4j versions prior 2.0
- The non transactional Cypher will be removed eventually, so we create now a transaction per query automatically
- Experimental support for IPython Notebook rendering
- Fix #101. Fix a problem when accessing node properties inside transaction for queries

2.0.1 (2014-03-23)

- Fix coveralls for Travis
- Fix #100. Fixes rollback problem when outside a with statement
- Update Neo4j versions for testing
- Remove inrange test for version 1.7.2 of Neo4j
- Add specific test for inrange lookups
- Fixes #98. Bug due to an incorrect treatment of numbers in eq, equals, neq, notequals lookups
- Add downloads
- Split exceptions from request.py file to a exceptions.py file
- Update requirements.txt
- Fix #96, fix dependency versions
- Fix #95. Support for creating spatial indexes

2.0.0 (2013-12-30)

- Add support for Neo4j 2.0
- Add Python3 support
- Remove Python 2.6 support
- Add support for Cypher transactional endpoint
- Add documentation for Cypher transactions
- Add support for Labels
- Add documentation for Labels
- Add support to pass Neo4j URL as the host, and neo4j-rest-client will request for the '/db/data' part in an extra request
- Add option for enabling verification of SSL certificates

- Fix #94. Disable lazy loading from Cypher queries but keep if for filters
- Update documentation
- Add the option to 'create' labels and add nodes to them
- Add filtering support for Labels
- Add tests for Labels
- Better structure to organize tests
- Add UnitTest.skipIf instead of my own decorator @versions
- Add development requirements and PyPy to Travis
- Add flake8
- Add support for tox
- Skip some test that depend on newer versions of other dependencies
- Update README with Coveralls.io image
- Add coverage
- Add extra requires for tests
- Enable syntax highlighting, fix spelling errors
- Fix #92. Allow nodes to be deleted from index without key or value
- Fix an error on traversals time_out when decimal values are passed
- Update Neo4j versions for Travis
- PEP8 review
- Add .all method to get all the elements. Underneath, it invokes .filter with no arguments
- Merge pull request #85 from carlsonp/patch-1

1.9.0 (2013-05-27)

- Add Neo4j 1.9 and 2.0.0-M02 to tests and Travis.
- Fix Python 2.6 compatibility. Last Python 2.6 issue fixed.
- Fix test_filter_nodes_complex_lookups test for empty databases
- Fix get_or_create and create_or_fail tests and add SMART_ERRORS for those functions
- Add support for Neo4j versions when testing in Travis
- Add support for get_or_create and create_or_fail index operations
- Adding integration tests with Travis-CI
- Updated requirements.txt with Shrubbery proposals
- Add experimental support for smart dates

1.8.0 (2012-12-09)

- Updated lucene-querybuilder requirement.
- Add support for using Indexes as start points when filtering
- Add support for using filters in indices.
- Fixes an error when using cert and key files.
- Adding order by and filtering for relationships.
- First implementation of complex filtering and slicing for nodes based on Cypher.
- Improving stability of tests.
- Fixes #74. Added the new .query() method and casting for returns. Also a very initial .filter method with an special Q object for composing complex filters.
- Fixes #64, added a small unicode check.
- Feature cache store and cache extension requests. Every time extension is used a get request is made before post this only needs to happen once per extension.
- Allow user to configure own cache engine, (e.g djangos cache).
- Read test db url from environ.
- Fixes #71. Pass correct url to get. Get with missing '/' was causing an additional 302.
- Support keep-alive / pipelining: httplib now instantiated on module load not per quest this also fixes caching, when the CACHE option was set a no-cache header was added that by passed the cache system.
- Fixes #68. Gremlin query trips on "simple" list, but not an error no neo4j-rest-client side.
- Fixes #69. Incorrect node references when splitting transactions.
- Adding support for retrieving index elements in a transaction.
- Fixes #66. Ditch exception catch on root fetch at GraphDatabase.__init__(). As per #65, current behaviour when auth fails is that a 401 StatusException is raised, and caught by this try/except block and a misleading NotFoundError is raised in its place - lets just let the StatusException through. Unsure about what other Exceptions may be raised but cannot reproduce.
- Fixed issue #69. Transaction split.
- Adding support for retrieving index elements in a transaction.

1.7.0 (2012-05-17)

- Fixing an error when reating relationships with nodes created previously in a transactions.
- Fixing typo (self._aith vs self_auth).
- Fixing #60. Adding support when no port is specified.
- Fixing an error with unicode property names and indexing.

1.6.2 (2012-03-26)

- Fixing an error indexing with numeric values.
- Fixing an error indexing with boolean values.

- Adding initial unicode suppport for indices. Adding better debug messages to 400 response codes.

1.6.1 (2012-02-27)

- Fixes #29. Adding support for authentication.

1.6.0 (2012-02-27)

- Adding documentation site.
- Finishing the experimental support for indexing and transactions.
- Adding preliminar indexing support in trasnsactions.
- Adding a new way to traverse the graph based on python-embedded.
- Removing __credits__ in favor of AUTHORS file. Updating version number.
- Fixes #33. Deprecating the requirement of a reference node.
- Added methods to bring it in line with the embedded driver.
- Added .single to Iterable and .items() to Node to bring it into alignment with the embedded driver.
- Adding non-functional realtionshos creation inside transactions.
- New returnable type "RAW", added in constants. Very useful for Gremlin and Cypher queries.
- Extensions can now return raw results. Fixes #52.
- Added a test for issue #52, returns=RAW.
- Adding relationships support to transactions.
- Fixes #49. Usage in extensions.
- Improving transaction support. Related #49.
- Fixing some PEP08 warnings.
- Fixes #43. Unable to reproduce the error.
- Fixes #49. Improving the batch efficiency in get requests.
- Fixes #47. Improving Paths management in traversals.
- Adding 'content-location' as possible header in responses instead of just 'location'.
- Fixing an error wwhen the value of a set property operation is None.
- Merge branch 'master' of github.com:versae/neo4j-rest-client into devel.
- Fix for paginated traversals under Neo4j 1.5.
- Added check for 'content-location' header in PaginatedTraversal, ensuring traversals don't stop early with Neo4j 1.5.

1.5.0 (2011-10-31)

- Removing the smart_quote function from indexing. It's not needed anymore with the new way to add elements to indices.
- Fixes #37.

- Using JSON object to set index key and value.

1.4.5 (2011-09-15)

- Adding more testing to returns parameter in the extensions.
- Fixes 32. It needs some more testing, maybe.
- Updated to using lucene-querybuilder 0.1.5 (bugfixes and better wildcard support).
- Fixed the test issue found in #34, and updated the REST client to using lucene-querybuilder 0.1.5.
- Fixes #34. Fixing dependency of lucene-querybuilder version
- Fixes #30. Fixing an issue deleting all index entries for a node.
- Fixing an issue with parameters in extensions.
- Ensure that self.result is always present on the object, even if it's None.
- Fixing naming glitch in exception message
- Ensure that self.result is always present on the object, even if it's None
- Fixing an error retrieving relationships in paths.
- Fixing an error in extensions, Path and Position.

1.4.4 (2011-08-17)

- Merge pull request #28 from mhluongo/master
- Made the DeprecationWarnings a bit more specific.
- Nodes can now be used in set and as dict keys, differentiated by id.
- Added a test for node hashing on id.
- Removed the 'Undirected' reference from tests to avoid a DepreactionWarning.
- Moved the relationship creation DeprecationWarning so creating a relationship the preferred way won't raise it.
- Got rid of the DeprecationWarning on import- moved in to whenever using Undirected.*.
- Fixed traversal return filters.
- Enabled return filters, including those with custom javascript bodies. Eventually a more elegant (Python instead of string based) solution for return filter bodies is in order.
- Fixed a mispelling in the test_traversal_return_filter case.
- Added a test for builtin and custom traversal return filters.
- Small bug fix for traversal
- Fixed bug in traverse method for POSITION and PATH return types.

1.4.3 (2011-07-28)

- Added some deprecation warnings.
- Added support for pickling ans some tests.
- Fixed an error deleting nodes and relationships on transactions.

- Finishied and refactored the full unicode support.

1.4.2 (2011-07-18)

- Updated the documentation and version.
- Added support for indices deletion.
- Improved Unicode support in properties keys and values and relationships types. Adding some tests.

1.4.1 (2011-07-12)

- Fixed an error retrieving relationships by id.
- Added control to handle exceptions raised by Request objects.
- Updated changes, manifest and readme files.

1.4.0 (2011-07-11)

- Updated version number for the new release.
- Updated documentation.
- Updated develpment requirements.
- Added support for paginated traversals.
- Passed pyflakes and PEP8 on tests.
- Added weight to Path class.
- Index values now quoted_plus.
- Changed quote to quote_plus for index values.
- Added two tests for unicode and url chars in index values.
- Added initial documentacion for transactions.
- Added the transaction support and several tests.
- Fixed the implementation of __contains__ in Iterable class for evaluation of 'in' and 'not in' expressions.
- Added documentation for Iterable objects.
- Added more transactions features.
- Added requirements file for virtual environments in development.
- Improved number of queries slicing the returned objects in a Iterable wrapper class.
- Added Q syntax for more complicated queries.
- Added support for the Q query syntax for indexes using the DSL at http://github.com/scholrly/lucene-querybuilder
- Fixed an error in the test_query_index case (forgot to include an 'or'. between queries).
- Added lucene-querybuilder to the test requirements in setup.py.
- Added a test case for Q-based queries.

1.3.4 (2011-06-22)

- Fixed the setup.py and httplib2 import error during installing.
- Reordered the options variables in an options.py file. Allows index.query() to be called with or without a key
- Fixed issue #15 regarding dependency to httplib2
- Patched index.query() so it can take a query without a key (to support, say, mutli-field Lucene queries). Ultimately, query so probably be refactored to Index (instead of IndexKey) because IndexKey doesn't actually help with full-text queries.
- Fixed for issue #19 (missed that urllib.quote).
- Altered the test_query_index case to reflect how I think indexing should work.
- Using assertTrue instead of failUnless in tests.py, failUnless is deprecated in 2.7 and up, so I figured we might as well switch.
- Added SMART_ERRORS (aka "Django mode"). If you set SMART_ERROR to True it will make the client throw KeyError instead of NotFoundError when a key is missing.

1.3.3 (2011-06-14)

- Fixed an introspection when the results list of a traverse is empty.
- Merge pull request #17 from mhluongo/master
- Resolved the STOP_AT_END_OF_GRAPH traversal test case. Calling .traverse(stop=STOP_AT_END_OF_GRAPH) will now traverse the graph without a max depth (and without 500 errors).
- Added a failing test case for traverse(stop=STOP_AT_END_OF_GRAPH).

1.3.2 (2011-05-30)

- Added a test for deleting relationships.
- Fixing an Index compatibility issue with Python 2.6.1.
- Fixing an error in extensions support with named params.

1.3.1 (2011-04-16)

- Fixing setup.py.

1.3.0 (2011-04-15)

- First Python Index Package release with full support for Neo4j 1.3.