neo4j-connector Documentation

Release 1.1.0

Jelle Jan Bankert

Jan 16, 2020

Contents:

CHAPTER 1

neo4j package

1.1 Module contents

This module implements access to the Neo4j HTTP API using the requests library.

class neo4j.**Connector**(*host: str = 'http://localhost:7474'*, *credentials: Tuple[str, str] = ('neo4j', 'neo4j')*, *verbose_errors=False*)

Bases: object

Class that abstracts communication with neo4j into up-front setup and then executes one or more *Statement*. The connector doesn't maintain an open connection and thus doesn't need to be closed after use.

Parameters

- **endpoint** (*str*) – the fully qualified endpoint to send messages to
- **credentials** (*tuple[str, str]*) – the credentials that are used to authenticate the requests
- **verbose_errors** (*bool*) – if set to True the *Connector* prints *Neo4jErrors* messages and codes to the standard error output in a bit nicer format than the stack trace.

Example code:

```
>>> # default connector
>>> connector = Connector()
```

```
>>> # localhost connector, custom credentials
>>> connector = Connector(credentials=('username', 'password'))
```

```
>>> # custom connector
>>> connector = Connector('http://mydomain:7474', ('username', 'password'))
```

default_credentials = ('neo4j', 'neo4j')

default_host = 'http://localhost:7474'

default_path = '/db/data/transaction/commit'

static make_batches(*statements: List[neo4j.Statement], batch_size: int = None*) → List[T]

post(*statements: List[neo4j.Statement]*)

Method that performs an HTTP POST with the provided *Statement*s and returns the parsed data structure as specified in Neo4j's documentation. This specifically includes the metadata per row and has a separate entry for the result names and the actual values.

Parameters **statements** (*list[Statement]*) – the statements that are POST-ed to Neo4j

Returns the parsed Neo4j HTTP API response

Return type dict

Raises *Neo4jErrors*

Example code:

```
>>> cypher = "MATCH (n:node {uuid: {uuid}}) RETURN n"
>>> statements = [Statement(cypher, {'uuid': uuid}) for uuid in ['123abc',
↪'456def']
>>> statements_responses = connector.run_multiple(statements)
>>> for result in statements_responses['results']:
>>>     for datum in result['data']:
>>>         print(datum['row'][0]) #n is the first item in the row
```

run(*cypher: str, parameters: dict = None*)

Method that runs a single statement against Neo4j in a single transaction. This method builds the *Statement* object for the user.

Parameters

- **cypher** (*str*) – the Cypher statement
- **parameters** (*dict*) – [optional] parameters that are merged into the statement at the server-side. Parameters help with speeding up queries because the execution plan for identical Cypher statements is cached.

Returns a list of dictionaries, one dictionary for each row in the result. The keys in the dictionary are defined in the Cypher statement

Return type list[dict]

Raises *Neo4jErrors*

Example code:

```
>>> # retrieve all nodes' properties
>>> all_nodes = [row['n'] for row in connector.run("MATCH (n) RETURN n")]
```

```
>>> # single row result
>>> node_count = connector.run("MATCH () RETURN COUNT(*) AS node_count")[0][
↪'node_count']
```

```
>>> # get a single node's properties with a statement + parameter
>>> # in this case we're assuming: CONSTRAINT ON (node:node) ASSERT node.uuid␣
↪IS UNIQUE
>>> single_node_properties_by_uuid = connector.run("MATCH (n:node {uuid:
↪{uuid}}) RETURN n", {'uuid': '123abc'})[0]['n']
```

run_multiple (*statements: List[neo4j.Statement], batch_size: int = None*) → List[List[dict]]

Method that runs multiple *Statement*s against Neo4j in a single transaction or several batches.

Parameters

- **statements** (*list[Statement]*) – the statements to execute
- **batch_size** (*int*) – [optional] number of statements to send to Neo4j per batch. In case the batch_size is omitted (i.e. None) then all statements are sent as a single batch. This parameter can help make large jobs manageable for Neo4j (e.g not running out of memory).

Returns a list of statement results, each containing a list of dictionaries, one dictionary for each row in the result. The keys in the dictionary are defined in the Cypher statement. The statement results have the same order as the corresponding *Statement*s

Return type list[list[dict]]

Raises *Neo4jErrors*

Example code:

```
>>> cypher = "MATCH (n:node {uuid: {uuid}}) RETURN n"
>>> statements = [Statement(cypher, {'uuid': uuid}) for uuid in ['123abc',
↪'456def']
>>> statements_responses = connector.run_multiple(statements)
>>> for statement_responses in statements_responses:
>>>     for row in statement_responses:
>>>         print(row)
```

```
>>> # we can use batches if we're likely to overwhelm neo4j by sending␣
↪everything in a single request
>>> # note that this has no effect on the returned data structure
>>> cypher = "MATCH (n:node {uuid: {uuid}}) RETURN n"
>>> statements = [Statement(cypher, {'uuid': uuid}) for uuid in range(1_000_
↪000)
>>> statements_responses = connector.run_multiple(statements, batch_size=10_
↪000)
>>> for statement_responses in statements_responses:
>>>     for row in statement_responses:
>>>         print(row)
```

```
>>> # we can easily re-use some information from the statement in the next␣
↪example
>>> cypher = "MATCH (language {name: {name}})-->(word:word)) RETURN word"
>>> statements = [Statement(cypher, {'name': lang}) for lang in ['en', 'nl']
>>> statements_responses = connector.run_multiple(statements)
>>> for statement, responses in zip(statements, statements_responses):
>>>     for row in responses:
>>>         print("{language}: {word_lemma}".format(language=statement[
↪'parameters']['lang'], word=row['word']['lemma']))
```

class neo4j.**Neo4jError**

Bases: *neo4j.Neo4jError*

namedtuple that contains the code and message of a Neo4j error

Parameters

- **code** (*str*) – Error status code as defined in https://neo4j.com/docs/status-codes/3.5/

- **message** (*str*) – Descriptive message. For Cypher syntax errors this will contain a separate line (delimited by \n) that contains the '^' character to point to the problem

Example code:

```
>>> print(neo4j_error.code, file=sys.stderr)
>>> print(neo4j_error.message, file=sys.stderr)
```

exception neo4j.**Neo4jErrors**(*errors: List[dict]*)

Bases: Exception

Exception that is raised when Neo4j responds to a request with one or more error message. Iterate over this object to get the individual *Neo4jError* objects

Parameters **errors** (*list(dict)*) – A list of dictionaries that contain the 'code' and 'message' properties

Example code:

```
>>> try:
>>>     connector.run(...)
>>> except Neo4jErrors as neo4j_errors:
>>>     for neo4j_error in neo4j_errors:
>>>         print(neo4j_error.code, file=sys.stderr)
>>>         print(neo4j_error.message, file=sys.stderr)
```

class neo4j.**Statement**(*cypher: str*, *parameters: dict = None*)

Bases: dict

Class that helps transform a cypher query plus optional parameters into the dictionary structure that Neo4j expects. The values can easily be accessed as shown in the last code example.

Parameters

- **cypher** (*str*) – the Cypher statement
- **parameters** (*dict*) – [optional] parameters that are merged into the statement at the server-side. Parameters help with speeding up queries because the execution plan for identical Cypher statements is cached.

Example code:

```
>>> # create simple statement
>>> statement = Statement("MATCH () RETURN COUNT(*) AS node_count")
```

```
>>> # create parametrized statement
>>> statement = Statement("MATCH (n:node {uuid: {uuid}}) RETURN n", {'uuid':
↪'123abc'})
```

```
>>> # create multiple parametrized statements
>>> statements = [Statement("MATCH (n:node {uuid: {uuid}}) RETURN n", {'uuid':␣
↪uuid}) for uuid in ['123abc', '456def']]
```

```
>>> # print individual Statement values
>>> statement = Statement("MATCH (n:node {uuid: {uuid}}) RETURN n", {'uuid':
↪'123abc'})
>>> print("Cypher statement: {}".format(statement['statement']))
>>> print("Parameters dict: {}".format(str(statement['parameters']))
```

CHAPTER 2

README

This library contains everything you need to execute single-request transactions for Neo4j 3.0 and above through its HTTP API.

2.1 Background

Research into the speed of performing ETL and batch-type actions on Neo4j showed that using a large, single-request transaction POST-request through Neo4j's HTTP API outperforms other currently available libraries for this use-case (e.g. the official 'neo4j-driver' and 'py2neo'). The goal of this connector is to provide convenience methods and classes that abstract away the boilerplate communication code.

Community thread about the difference in performance between drivers: https://community.neo4j.com/t/barebones-http-requests-much-faster-than-python-neo4j-driver-and-py2neo

2.2 Example

```
import neo4j

connector = neo4j.Connector('http://localhost:7474', ('neo4j','neo4j'))
response = connector.run("""MATCH () RETURN COUNT(*) as node_count""")
first_row = response[0]
print(first_row['node_count'])
```

2.3 Installation

To install the latest stable version, use:

```
pip install neo4j-connector
```

2.4 Github

This library lives at https://github.com/textkernel/neo4j-connector. Suggestions, bug-reports and pull requests are welcome there.

2.5 Documentation

The documentation (including changelog) lives at https://neo4j-connector.readthedocs.io

CHAPTER 3

Changelog

3.1 1.1.0

- the `run_multiple` connector method now takes the `batch_size` parameter. If the method gets more statements than `batch_size` then the statements are split over multiple HTTP requests. The `run_multiple` method still returns a single list with the responses from all batches. This parameter can help make large jobs manageable for Neo4j (e.g not running out of memory).

3.2 1.0.1

- Improved documentation

3.3 1.0.0

- Initial release

CHAPTER 4

Indices and tables

- genindex
- search

Python Module Index

n

Index