neomodel Documentation

Release 3.2.9

Robin Edwards

Jul 04, 2018

Contents

An Object Graph Mapper (OGM) for the neo4j graph database, built on the awesome neo4j_driver

- Familiar Django model style definitions.
- Powerful query API.
- Enforce your schema through cardinality restrictions.
- Full transaction support.
- Thread safe.
- pre/post save/delete hooks.
- Django integration via django_neomodel

CHAPTER 1

Requirements

- Python 2.7, 3.4+
- neo4j 3.0, 3.1, 3.2, 3.3

CHAPTER 2

Installation

Install from pypi (recommended):

```
$ pip install neomodel
```

To install from github:

```
$ pip install git+git://github.com/neo4j-contrib/neomodel.git@HEAD#egg=neomodel-dev
```

CHAPTER 3

Contents

3.1 Getting started

3.1.1 Connecting

Before executing any neomodel code set the connection url:

```
from neomodel import config
config.DATABASE_URL = 'bolt://neo4j:neo4j@localhost:7687'  # default
```

This needs to be called early on in your app, if you are using Django the settings.py file is ideal.

If you are using your neo4j server for the first time you will need to change the default password. This can be achieved by visiting the neo4j admin panel (default: http://localhost:7474).

You can also change the connection url at any time by calling *set_connection*:

```
from neomodel import db
db.set_connection('bolt://neo4j:neo4j@localhost:7687')
```

The new connection url will be applied to the current thread or process.

3.1.2 Definition

Below is a definition of two types of node *Person* and *Country*:

```
from neomodel import (config, StructuredNode, StringProperty, IntegerProperty,
    UniqueIdProperty, RelationshipTo, RelationshipFrom)

config.DATABASE_URL = 'bolt://neo4j:password@localhost:7687'

class Country(StructuredNode):
    code = StringProperty(unique_index=True, required=True)
```

(continues on next page)

(continued from previous page)

```
    # traverse incoming IS_FROM relation, inflate to Person objects
    inhabitant = RelationshipFrom('Person', 'IS_FROM')


class Person(StructuredNode):
    uid = UniqueIdProperty()
    name = StringProperty(unique_index=True)
    age = IntegerProperty(index=True, default=0)

    # traverse outgoing IS_FROM relations, inflate to Country objects
    country = RelationshipTo(Country, 'IS_FROM')
```

There is one type of relationship present *IS_FROM*, we are defining two different ways for traversing it one accessible via Person objects and one via Country objects

We can use the *Relationship* class as opposed to the *RelationshipTo* or *RelationshipFrom* if we don't want to specify a direction.

Neomodel automatically creates a label for each StructuredNode class in the database with the corresponding indexes and constraints.

3.1.3 Setup constraints and indexes

After creating node definitions in python, any constraints or indexes need to be added to Neo4j.

Neomodel provides a script to automate this:

```
$ neomodel_install_labels yourapp.py someapp.models --db bolt://
↪neo4j:neo4j@localhost:7687
```

It is important to execute this after altering the schema. Keep an eye on the number of classes it detects each time.

3.1.4 Remove existing constraints and indexes

For deleting all existing constraints and indexes from database, neomodel provides a script to automate this:

```
$ neomodel_remove_labels --db bolt://neo4j:neo4j@localhost:7687
```

After executing, it will print all indexes and constraints it has deleted.

3.1.5 Create, Save, Delete

Using convenient methods:

```
jim = Person(name='Jim', age=3).save()
jim.age = 4
jim.save() # validation happens here
jim.delete()
jim.refresh() # reload properties from neo
jim.id # neo4j internal id
```

3.1.6 Finding nodes

Using the '.nodes' class property:

```
# raises Person.DoesNotExist if no match
jim = Person.nodes.get(name='Jim')

# Will return None unless bob exists
someone = Person.nodes.get_or_none(name='bob')

# Will return the first Person node with the name bob. This raises Person.
↪DoesNotExist if there's no match.
someone = Person.nodes.first(name='bob')

# Will return the first Person node with the name bob or None if there's no match
someone = Person.nodes.first_or_none(name='bob')

# Return set of nodes
people = Person.nodes.filter(age__gt=3)
```

3.1.7 Relationships

Working with relationships:

```
germany = Country(code='DE').save()
jim.country.connect(germany)

if jim.country.is_connected(germany):
    print("Jim's from Germany")

for p in germany.inhabitant.all()
    print(p.name) # Jim

len(germany.inhabitant) # 1

# Find people called 'Jim' in germany
germany.inhabitant.search(name='Jim')

# Remove Jim's country relationship with Germany
jim.country.disconnect(germany)

usa = Country(code='US').save()
jim.country.connect(usa)
jim.country.connect(germany)

# Remove all of Jim's country relationships
jim.country.disconnect_all()

jim.country.connect(usa)
# Replace Jim's country relationship with a new one
jim.country.replace(germany)
```

3.2 Relationships

Directionless relationships, first argument the class second the neo4j relationship:

```
class Person(StructuredNode):
    friends = Relationship('Person', 'FRIEND')
```

When defining relationships, you may refer to classes in other modules. This avoids cyclic imports:

```
class Garage(StructuredNode):
    cars = RelationshipTo('transport.models.Car', 'CAR')
    vans = RelationshipTo('.models.Van', 'VAN')
```

3.2.1 Cardinality

It's possible to (softly) enforce cardinality restrictions on your relationships. Remember this needs to be declared on both sides of the definition:

```
class Person(StructuredNode):
    car = RelationshipTo('Car', 'OWNS', cardinality=One)

class Car(StructuredNode):
    owner = RelationshipFrom('Person', 'OWNS', cardinality=One)
```

The following cardinality classes are available:

ZeroOrOne	*One*
`ZeroOrMore` (default)	*OneOrMore*

If cardinality is broken by existing data a `CardinalityViolation` exception is raised. On attempting to break a cardinality restriction a `AttemptedCardinalityViolation` is raised.

3.2.2 Properties

Neomodel uses *relationship* models to define the properties stored on relations:

```
class FriendRel(StructuredRel):
    since = DateTimeProperty(
        default=lambda: datetime.now(pytz.utc)
    )
    met = StringProperty()

class Person(StructuredNode):
    name = StringProperty()
    friends = RelationshipTo('Person', 'FRIEND', model=FriendRel)

rel = jim.friends.connect(bob)
rel.since # datetime object
```

These can be passed in when calling the connect method:

```
rel = jim.friends.connect(bob,
                          {'since': yesterday, 'met': 'Paris'})

print(rel.start_node().name) # jim
print(rel.end_node().name) # bob

rel.met = "Amsterdam"
rel.save()
```

You can retrieve relationships between two nodes using the 'relationship' method. This is only available for relationships with a defined relationship model:

```
rel = jim.friends.relationship(bob)
```

3.2.3 Relationship Uniqueness

By default in neomodel there is only one relationship of one type between two nodes unless you define different properties when calling connect. neomodel utilises *CREATE UNIQUE* in cypher to achieve this.

3.2.4 Explicit Traversal

It is possible to specify a node traversal by creating a *Traversal* object. This will get all `Person` entities that are directly related to another `Person`, through all relationships:

```
definition = dict(node_class=Person, direction=OUTGOING,
                  relation_type=None, model=None)
relations_traversal = Traversal(jim, Person.__label__,
                                definition)
all_jims_relations = relations_traversal.all()
```

The `defintion` argument is a mapping with these items:

`node_class`	The class of the traversal target node.
`direction`	`match.OUTGOING` / `match.INCOMING` / `match.EITHER`
`relation_type`	Can be `None` (for any direction), `*` for all paths or an explicit name of a relation type (the edge's label).
`model`	The class of the relation model, `None` for such without one.

3.3 Property types

The following properties are available on nodes and relationships:

AliasProperty	*IntegerProperty*
ArrayProperty	*JSONProperty*
BooleanProperty	*RegexProperty*
DateProperty	*StringProperty*
DateTimeProperty	*UniqueIdProperty*
FloatProperty	

3.3.1 Defaults

Default values you may provide a default value to any property, this can also be a function or any callable:

```
from uuid import uuid4
my_id = StringProperty(unique_index=True, default=uuid4)
```

You may provide arguments as function or lambda:

```
my_datetime = DateTimeProperty(default=lambda: datetime.now(pytz.utc))
```

3.3.2 Choices

You can specify a list of valid values for a `StringProperty` using the `choices` argument. The mapping's values are supposed to be used when displaying information to users:

```
class Person(StructuredNode):
    SEXES = {'F': 'Female', 'M': 'Male', 'O': 'Other'}
    sex = StringProperty(required=True, choices=SEXES)

tim = Person(sex='M').save()
tim.sex # M
tim.get_sex_display() # 'Male'
```

The value's validity will be checked both when saved and loaded from Neo4j.

3.3.3 Array Properties

Neo4j supports arrays as a property value, these are used with the *ArrayProperty* class. You may optionally provide a list element type as the first argument to ArrayProperty with another property instance:

```
class Person(StructuredNode):
    names = ArrayProperty(StringProperty(), required=True)

bob = Person(names=['bob', 'rob', 'robert']).save()
```

In this example each element in the list is deflated to a string prior to being persisted.

3.3.4 Unique Identifiers

All nodes in neo4j have an internal id (accessible by the 'id' property in neomodel) however these should not be used by an application. neomodel provides the *UniqueIdProperty* to generate unique identifiers for your nodes (with an unique index):

```
class Person(StructuredNode):
    uid = UniqueIdProperty()

Person.nodes.get(uid='a12df...')
```

3.3.5 Dates and times

The *DateTimeProperty* accepts datetime.datetime objects of any timezone and stores them as a UTC epoch value. These epoch values are inflated to datetime.datetime objects with the UTC timezone set.

The *DateProperty* accepts datetime.date objects which are stored as a string property 'YYYY-MM-DD'.

You can use *default_now* argument to store the current time by default:

```
created = DateTimeProperty(default_now=True)
```

You can enforce timezones by setting the config var NEOMODEL_FORCE_TIMEZONE=1.

3.3.6 Other properties

- *EmailProperty* - validate emails (via a regex).
- *RegexProperty* - passing in a validator regex: *RegexProperty(expression=r'dw')*
- *NormalProperty* - use one method (normalize) to inflate and deflate.

3.3.7 Aliasing properties

Allows aliasing to other properties can be useful to provide 'magic' behaviour, (only supported on *StructuredNodes*):

```
class Person(StructuredNode):
    full_name = StringProperty(index=True)
    name = AliasProperty(to='full_name')

Person.nodes.filter(name='Jim') # just works
```

3.3.8 Independent database property name

You can specify an independent property name with 'db_property', which is used on database level. It behaves like Django's 'db_column'. This is useful for e.g. hiding graph properties behind a python property:

```
class Person(StructuredNode):
    name_ = StringProperty(db_property='name')

    @property
    def name(self):
        return self.name_.lower() if self.name_ else None

    @name.setter
    def name(self, value):
        self.name_ = value
```

3.4 Advanced queries

Neomodel contains an API for querying sets of nodes without needing to write cypher:

```
class SupplierRel(StructuredRel):
    since = DateTimeProperty(default=datetime.now)


class Supplier(StructuredNode):
    name = StringProperty()
    delivery_cost = IntegerProperty()
    coffees = RelationshipTo('Coffee', 'SUPPLIES')


class Coffee(StructuredNode):
    name = StringProperty(unique_index=True)
    price = IntegerProperty()
    suppliers = RelationshipFrom(Supplier, 'SUPPLIES', model=SupplierRel)
```

3.4.1 Node sets and filtering

The *nodes* property on a class is the set of all nodes in the database of that type contained.

This set (or *NodeSet*) can be iterated over and filtered on. Under the hood it uses labels introduced in neo4j 2:

```
# nodes with label Coffee whose price is greater than 2
Coffee.nodes.filter(price__gt=2)

try:
    java = Coffee.nodes.get(name='Java')
except Coffee.DoesNotExist:
    print "Couldn't find coffee 'Java'"
```

The filter method borrows the same django filter format with double underscore prefixed operators:

- lt - less than
- gt - greater than
- lte - less than or equal to
- gte - greater than or equal to
- ne - not equal
- in - item in list
- isnull - *True* IS NULL, *False* IS NOT NULL
- exact - string equals
- iexact - string equals, case insensitive
- contains - contains string value
- icontains - contains string value, case insensitive
- startswith - starts with string value
- istartswith - starts with string value, case insensitive
- endswith - ends with string value
- iendswith - ends with string value, case insensitive
- regex - matches a regex expression

- iregex - matches a regex expression, case insensitive

3.4.2 Has a relationship

The *has* method checks for existence of (one or more) relationships, in this case it returns a set of *Coffee* nodes which have a supplier:

```
Coffee.nodes.has(suppliers=True)
```

This can be negated *suppliers=False*, should you wish to find *Coffee* nodes without *suppliers*.

3.4.3 Iteration, slicing and more

Iteration, slicing and counting is also supported:

```
# Iterable
for coffee in Coffee.nodes:
    print coffee.name

# Sliceable using python slice syntax
coffee = Coffee.nodes.filter(price__gt=2)[2:]
```

The slice syntax returns a NodeSet object which can in turn be chained.

Length and boolean methods dont return NodeSet objects so cant be chained further:

```
# Count with __len__
print len(Coffee.nodes.filter(price__gt=2))

if Coffee.nodes:
    print "We have coffee nodes!"
```

3.4.4 Filtering by relationship properties

Filtering on relationship properties is also possible using the *match* method. Note that again these relationships must have a definition.:

```
nescafe = Coffee.nodes.get(name="Nescafe")

for supplier in nescafe.suppliers.match(since_lt=january):
    print supplier.name
```

3.4.5 Ordering by property

To order results by a particular property, use the *order_by* method:

```
# Ascending sort
for coffee in Coffee.nodes.order_by('price'):
    print coffee, coffee.price

# Descending sort
for supplier in Supplier.nodes.order_by('-delivery_cost'):
    print supplier, supplier.delivery_cost
```

To remove ordering from a previously defined query, pass *None* to *order_by*:

```
# Sort in descending order
suppliers = Supplier.nodes.order_by('-delivery_cost')

# Don't order; yield nodes in the order neo4j returns them
suppliers = suppliers.order_by(None)
```

For random ordering simply pass '?' to the order_by method:

```
Coffee.nodes.order_by('?')
```

3.5 Cypher queries

You may handle more complex queries via cypher. Each *StructuredNode* provides an 'inflate' class method, this inflates nodes to their class. It is your responsibility to make sure nodes get inflated as the correct type:

```
class Person(StructuredNode):
    def friends(self):
        results, columns = self.cypher("MATCH (a) WHERE id(a)={self} MATCH (a)-
↪[:FRIEND]->(b) RETURN b")
        return [self.inflate(row[0]) for row in results]
```

The self query parameter is prepopulated with the current node id. It's possible to pass in your own query parameters to the cypher method.

3.5.1 Stand alone

Outside of a *StructuredNode*:

```
# for standalone queries
from neomodel import db
results, meta = db.cypher_query(query, params)
people = [Person.inflate(row[0]) for row in results]
```

3.5.2 Logging

You may log queries and timings by setting the environment variable *NEOMODEL_CYPHER_DEBUG* to *1*.

3.5.3 Utilities

The following utility functions are available:

```
clear_neo4j_database(db)  # deletes all nodes and relationships

# Change database password (you will need to call db.set_connection(...) after
change_neo4j_password(db, new_password)
```

3.6 Transactions

Transactions can be used via a context manager:

```python
from neomodel import db

with db.transaction:
    Person(name='Bob').save()
```

or as a function decorator:

```python
@db.transaction
def update_user_name(uid, name):
    user = Person.nodes.filter(uid=uid)[0]
    user.name = name
    user.save()
```

or manually:

```python
db.begin()
try:
    new_user = Person(name=username, email=email).save()
    send_email(new_user)
    db.commit()
except Exception as e:
    db.rollback()
```

Transactions are local to the thread as is the *db* object (see *threading.local*). If you're using celery or another task scheduler it's advised to wrap each task within a transaction:

```python
@task
@db.transaction  # comes after the task decorator
def send_email(user):
    ...
```

Transactions can also be designated as *write transactions*. Marking as a write transaction may not matter on a single-instance Neo4J, but is vital in a Neo4J causal cluster where writes must be sent to the leader node. For performance purposes, this should be limited to transactions that write/delete data, or any situation where talking to the leader node is essential:

```python
@db.write_transaction
def update_user_name(uid, name):
    user = Person.nodes.filter(uid=uid)[0]
    user.name = name
    user.save()
```

3.7 Hooks

You may define the following hook methods on your *StructuredNode* sub classes:

```
pre_save, post_save, pre_delete, post_delete, post_create
```

All take no arguments. An example of the post creation hook:

```
class Person(StructuredNode):

    def post_create(self):
        email_welcome_message(self)
```

Note the *post_create* hook is not called by *get_or_create* and *create_or_update* methods.

Save hooks are called regardless of wether the node is new or not. To determine if a node exists in *pre_save*, check for an *id* attribute on self.

3.7.1 Hooks on relationships

The hooks *pre_save* and *post_save* are available on *StructuredRel* models. They are executed when calling save on the object directly or when creating a new relationship via connect.

Note that in the pre_save call during a connect the start and end nodes are not available.

Django signals

Signals are now supported through the django_neomodel module.

3.8 Batch nodes operations

All batch operations can be executed with one or more nodes.

3.8.1 create()

Note that batch create is a relic of the Neo4j REST API. Now neomodel uses Bolt it exists for convenience and compatibility and a CREATE query is issued for each dict provided.

Create multiple nodes at once in a single transaction:

```
with db.transaction:
    people = Person.create(
        {'name': 'Tim', 'age': 83},
        {'name': 'Bob', 'age': 23},
        {'name': 'Jill', 'age': 34},
    )
```

3.8.2 create_or_update()

Atomically create or update nodes in a single operation:

```
people = Person.create_or_update(
    {'name': 'Tim', 'age': 83},
    {'name': 'Bob', 'age': 23},
    {'name': 'Jill', 'age': 34},
)

more_people = Person.create_or_update(
```

(continues on next page)

(continued from previous page)

```
    {'name': 'Tim', 'age': 73},
    {'name': 'Bob', 'age': 35},
    {'name': 'Jane', 'age': 24},
)
```

This is useful for ensuring data is up to date, each node is matched by its' required and/or unique properties. Any additional properties will be set on a newly created or an existing node.

It is important to provide unique identifiers where known, any fields with default values that are omitted will be generated.

3.8.3 get_or_create()

Atomically get or create nodes in a single operation:

```
people = Person.get_or_create(
    {'name': 'Tim'},
    {'name': 'Bob'},
)

people_with_jill = Person.get_or_create(
    {'name': 'Tim'},
    {'name': 'Bob'},
    {'name': 'Jill'},
)
# are same nodes
assert people[0] == people_with_jill[0]
assert people[1] == people_with_jill[1]
```

This is useful for ensuring specific nodes exist, only and all required properties must be specified to ensure uniqueness. In this example 'Tim' and 'Bob' are created on the first call, and are retrieved in the second call.

Additionally, get_or_create() allows the "relationship" parameter to be passed. When a relationship is specified, the matching is done based on that relationship and not globally:

```
class Dog(StructuredNode):
    name = StringProperty(required=True)
    owner = RelationshipTo('Person', 'owner')

class Person(StructuredNode):
    name = StringProperty(unique_index=True)
    pets = RelationshipFrom('Dog', 'owner')

bob = Person.get_or_create({"name": "Bob"})[0]
bobs_gizmo = Dog.get_or_create({"name": "Gizmo"}, relationship=bob.pets)

tim = Person.get_or_create({"name": "Tim"})[0]
tims_gizmo = Dog.get_or_create({"name": "Gizmo"}, relationship=tim.pets)

# not the same gizmo
assert bobs_gizmo[0] != tims_gizmo[0]
```

In case when the only required property is unique, the operation is redundant. However with simple required properties, the relationship becomes a part of the unique identifier.

3.9 Configuration

Covering the neomodel 'config' module and its variables.

3.9.1 Database

Set your connection details:

```
config.DATABASE_URL = 'bolt://neo4j:neo4j@localhost:7687`
```

Disable encrypted connection (usefult for development:

```
config.ENCRYPTED_CONNECTION = False
```

Adjust connection pool size:

```
config.MAX_POOL_SIZE = 50  # default
```

3.9.2 Enable automatic index and constraint creation

After a StructuredNode definition neomodel can install the corresponding constraints and indexes at compile time. However this method is only recommended for testing:

```
from neomodel import config
# before loading your node definitions
config.AUTO_INSTALL_LABELS = True
```

Neomodel provides a script *neomodel_install_labels* for the task, however if you want to handle this manually see below.

Install indexes and constraints for a single class:

```
from neomodel import install_labels
install_labels(YourClass)
```

Or for your entire 'schema'

```
import yourapp  # make sure your app is loaded
from neomodel import install_all_labels

install_all_labels()
# Output:
# Setting up labels and constraints...
# Found yourapp.models.User
# + Creating unique constraint for name on label User for class yourapp.models.User
# ...
```

3.9.3 Require timezones on DateTimeProperty

Ensure all DateTimes are provided with a timezone before being serialised to UTC epoch:

```
config.FORCE_TIMEZONE = True  # default False
```

3.10 Extending neomodel

3.10.1 Inheritance

You may want to create a 'base node' classes which extends the functionality which neomodel provides (such as *neomodel.contrib.SemiStructuredNode*).

Or just have common methods and properties you want to share. This can be achieved using the *__abstract_node__* property on any base classes you wish to inherit from:

```
class User(StructuredNode):
    __abstract_node__ = True
    name = StringProperty(unique_index=True)

class Shopper(User):
    balance = IntegerProperty(index=True)

    def credit_account(self, amount):
        self.balance = self.balance + int(amount)
        self.save()
```

3.10.2 Mixins

You can use mixins to share the functionality between nodes classes:

```
class UserMixin(object):
    name = StringProperty(unique_index=True)
    password = StringProperty()

class CreditMixin(object):
    balance = IntegerProperty(index=True)

    def credit_account(self, amount):
        self.balance = self.balance + int(amount)
        self.save()

class Shopper(StructuredNode, UserMixin, CreditMixin):
    pass

jim = Shopper(name='jimmy', balance=300).save()
jim.credit_account(50)
```

Make sure your mixins *dont* inherit from *StructuredNode* and your concrete class does.

3.10.3 Overriding the StructuredNode constructor

When defining classes that have a custom *__init__(self, ...)* method, you must always call *super()* for the neomodel magic to work:

```
class Item(StructuredNode):
    name = StringProperty(unique_index=True)
    uid = StringProperty(unique_index=True)

    def __init__(self, product, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        self.product = product
        kwargs["uid"] = 'g.' + str(self.product.pk)
        kwargs["name"] = self.product.product_name

        super(Item, self).__init__(self, *args, **kwargs)
```

It's important to note that *StructuredNode*'s constructor will override properties set (which are defined on the class). So you must pass the values in via *kwargs* (as above). You may set them after calling the constructor but it does skip validation.

3.11 Modules documentation

3.11.1 Core

class `neomodel.core.`**`NodeBase`**(***kwargs*)

`DoesNotExist`
alias of `NodeBaseDoesNotExist`

class `neomodel.core.`**`StructuredNode`**(**args*, ***kwargs*)
Base class for all node definitions to inherit from.

If you want to create your own abstract classes set: __abstract_node__ = True

`DoesNotExist`
alias of `StructuredNodeDoesNotExist`

classmethod create(**props*, ***kwargs*)
Call to CREATE with parameters map. A new instance will be created and saved.

Parameters

- **`props`** (*`tuple`*) – dict of properties to create the nodes.
- **`lazy`** – False by default, specify True to get nodes with id only without the parameters.

Type bool

Return type list

classmethod create_or_update(**props*, ***kwargs*)
Call to MERGE with parameters map. A new instance will be created and saved if does not already exists, this is an atomic operation. If an instance already exists all optional properties specified will be updated.

Note that the post_create hook isn't called after create_or_update

Parameters

- **`props`** (*`tuple`*) – List of dict arguments to get or create the entities with.
- **`relationship`** – Optional, relationship to get/create on when new entity is created.
- **`lazy`** – False by default, specify True to get nodes with id only without the parameters.

Return type list

`cypher`(*query*, *params=None*)
Execute a cypher query with the param 'self' pre-populated with the nodes neo4j id.

Parameters

- **`query`** – cypher query string
- **`params`** – query parameters

Type string

Type dict

Returns list containing query results

Return type list

`delete``()`
Delete a node and it's relationships

Returns True

`classmethod get_or_create``(`**props*, ***kwargs*`)`
Call to MERGE with parameters map. A new instance will be created and saved if does not already exists, this is an atomic operation. Parameters must contain all required properties, any non required properties with defaults will be generated.

Note that the post_create hook isn't called after get_or_create

Parameters

- **`props`** (`tuple`) – dict of properties to get or create the entities with.
- **`relationship`** – Optional, relationship to get/create on when new entity is created.
- **`lazy`** – False by default, specify True to get nodes with id only without the parameters.

Return type list

`classmethod inflate``(`*node*`)`
Inflate a raw neo4j_driver node to a neomodel node :param node: :return: node object

`classmethod inherited_labels``()`
Return list of labels from nodes class hierarchy.

Returns list

`labels``()`
Returns list of labels tied to the node from neo4j.

Returns list of labels

Return type list

`refresh``()`
Reload the node from neo4j

`save``()`
Save the node to neo4j or raise an exception

Returns the node instance

`neomodel.core.`**`drop_constraints`**`(`*quiet=True*, *stdout=None*`)`
Discover and drop all constraints.

Type bool

Returns None

`neomodel.core.`**`drop_indexes`**`(`*quiet=True*, *stdout=None*`)`
Discover and drop all indexes.

Type bool

Returns None

`neomodel.core.`**`install_all_labels`**(*stdout=None*)

Discover all subclasses of StructuredNode in your application and execute install_labels on each. Note: code most be loaded (imported) in order for a class to be discovered.

Parameters **`stdout`** – output stream

Returns None

`neomodel.core.`**`install_labels`**(*cls*, *quiet=True*, *stdout=None*)

Setup labels with indexes and constraints for a given class

Parameters

- **`cls`** – StructuredNode class
- **`quiet`** – (default true) enable standard output
- **`stdout`** – stdout stream

Type class

Type bool

Returns None

`neomodel.core.`**`remove_all_labels`**(*stdout=None*)

Calls functions for dropping constraints and indexes.

Parameters **`stdout`** – output stream

Returns None

3.11.2 Properties

`class` `neomodel.properties.`**`AliasProperty`**(*to=None*)

Bases: `property`, *`neomodel.properties.Property`*

Alias another existing property

`class` `neomodel.properties.`**`ArrayProperty`**(*base_property=None*, ***kwargs*)

Bases: *`neomodel.properties.Property`*

Stores a list of items

`default_value`()

Generate a default value

Returns the value

`class` `neomodel.properties.`**`BooleanProperty`**(*unique_index=False*, *index=False*, *required=False*, *default=None*, *db_property=None*, *label=None*, *help_text=None*, ***kwargs*)

Bases: *`neomodel.properties.Property`*

Stores a boolean value

`default_value`()

Generate a default value

Returns the value

class neomodel.properties.**DateProperty**(*unique_index=False*, *index=False*, *required=False*, *default=None*, *db_property=None*, *label=None*, *help_text=None*, ***kwargs*)

Bases: *neomodel.properties.Property*

Stores a date

class neomodel.properties.**DateTimeProperty**(*default_now=False*, ***kwargs*)

Bases: *neomodel.properties.Property*

A property representing a `datetime.datetime` object as unix epoch.

Parameters **default_now** (`bool`) – If `True`, the creation time (UTC) will be used as default. Defaults to `False`.

class neomodel.properties.**EmailProperty**(*expression=None*, ***kwargs*)

Bases: *neomodel.properties.RegexProperty*

Store email addresses

class neomodel.properties.**FloatProperty**(*unique_index=False*, *index=False*, *required=False*, *default=None*, *db_property=None*, *label=None*, *help_text=None*, ***kwargs*)

Bases: *neomodel.properties.Property*

Store a floating point value

default_value()

Generate a default value

Returns the value

class neomodel.properties.**IntegerProperty**(*unique_index=False*, *index=False*, *required=False*, *default=None*, *db_property=None*, *label=None*, *help_text=None*, ***kwargs*)

Bases: *neomodel.properties.Property*

Stores an Integer value

default_value()

Generate a default value

Returns the value

class neomodel.properties.**JSONProperty**(**args*, ***kwargs*)

Bases: *neomodel.properties.Property*

Store a data structure as a JSON string.

The structure will be inflated when a node is retrieved.

class neomodel.properties.**NormalProperty**(**args*, ***kwargs*)

Bases: *neomodel.properties.NormalizedProperty*

class neomodel.properties.**NormalizedProperty**(*unique_index=False*, *index=False*, *required=False*, *default=None*, *db_property=None*, *label=None*, *help_text=None*, ***kwargs*)

Bases: *neomodel.properties.Property*

Base class for normalized properties. These use the same normalization method to in- or deflating.

default_value()

Generate a default value

Returns the value

class `neomodel.properties.`**`Property`**(*unique_index=False, index=False, required=False, default=None, db_property=None, label=None, help_text=None, **kwargs*)

Bases: `object`

Base class for object properties.

Parameters

- **`unique_index`** (`bool`) – Creates a unique index for this property. Defaults to `False`.
- **`index`** (`bool`) – Creates an index for this property. Defaults to `False`.
- **`required`** (`bool`) – Marks the property as required. Defaults to `False`.
- **`default`** – A default value or callable that returns one to set when a node is initialized without specifying this property.
- **`db_property`** (`str`) – A name that this property maps to in the database. Defaults to the model's property name.
- **`label`** (`str`) – Optional, used by `django_neomodel`.
- **`help_text`** (`str`) – Optional, used by `django_neomodel`.

`default_value`()

Generate a default value

Returns the value

class `neomodel.properties.`**`PropertyManager`**(***kwargs*)

Bases: `object`

Common methods for handling properties on node and relationship objects.

class `neomodel.properties.`**`RegexProperty`**(*expression=None, **kwargs*)

Bases: *`neomodel.properties.NormalizedProperty`*

Validates a property against a regular expression.

If sub-classing set:

expression = r'[^@]+@[^@]+.[^@]+'

class `neomodel.properties.`**`StringProperty`**(*choices=None, **kwargs*)

Bases: *`neomodel.properties.NormalizedProperty`*

Stores a unicode string

Parameters **`choices`** (Any type that can be used to initiate a `dict`.) – A mapping of valid strings to label strings that are used to display information in an application. If the default value `None` is used, any string is valid.

`default_value`()

Generate a default value

Returns the value

class `neomodel.properties.`**`UniqueIdProperty`**(***kwargs*)

Bases: *`neomodel.properties.Property`*

A unique identifier, a randomly generated uid (uuid4) with a unique index

3.11.3 Relationships

class `neomodel.relationship.`**`StructuredRel`**`(`**args*, ***kwargs*`)`

Bases: `neomodel.relationship.RelationshipBase`

Base class for relationship objects

`end_node``()`

Get end node

Returns StructuredNode

`classmethod inflate``(`*rel*`)`

Inflate a neo4j_driver relationship object to a neomodel object :param rel: :return: StructuredRel

`save``()`

Save the relationship

Returns self

`start_node``()`

Get start node

Returns StructuredNode

`neomodel.relationship.`**`StructuredRelBase`**

alias of `neomodel.relationship.RelationshipBase`

class `neomodel.relationship_manager.`**`RelationshipManager`**`(`*source*, *key*, *definition*`)`

Bases: `object`

Base class for all relationships managed through neomodel.

I.e the 'friends' object in *user.friends.all()*

`all``()`

Return all related nodes.

Returns list

`all_relationships``(`**args*, ***kwargs*`)`

Retrieve all relationship objects between self and node.

Parameters **`node`** –

Returns [StructuredRel]

`connect``(`**args*, ***kwargs*`)`

Connect a node

Parameters

- **`node`** –
- **`properties`** – for the new relationship

Type dict

Returns

`disconnect``(`**args*, ***kwargs*`)`

Disconnect a node

Parameters **`node`** –

Returns

disconnect_all(**args*, ***kwargs*)
Disconnect all nodes

Returns

exclude(***kwargs*)
Exclude nodes that match the provided properties.

Parameters **kwargs** – same syntax as *NodeSet.filter()*

Returns NodeSet

filter(***kwargs*)
Retrieve related nodes matching the provided properties.

Parameters **kwargs** – same syntax as *NodeSet.filter()*

Returns NodeSet

get(***kwargs*)
Retrieve a related node with the matching node properties.

Parameters **kwargs** – same syntax as *NodeSet.filter()*

Returns node

get_or_none(***kwargs*)
Retrieve a related node with the matching node properties or return None.

Parameters **kwargs** – same syntax as *NodeSet.filter()*

Returns node

is_connected(*node*)
Check if a node is connected with this relationship type :param node: :return: bool

match(***kwargs*)
Return set of nodes who's relationship properties match supplied args

Parameters **kwargs** – same syntax as *NodeSet.filter()*

Returns NodeSet

order_by(**props*)
Order related nodes by specified properties

Parameters **props** –

Returns NodeSet

reconnect(**args*, ***kwargs*)
Disconnect old_node and connect new_node copying over any properties on the original relationship.

Useful for preventing cardinality violations

Parameters

- **old_node** –
- **new_node** –

Returns None

relationship(**args*, ***kwargs*)
Retrieve the relationship object for this first relationship between self and node.

Parameters **node** –

Returns StructuredRel

replace(**args*, ***kwargs*)
Disconnect all existing nodes and connect the supplied node

Parameters

- **node** –
- **properties** – for the new relationship

Type dict

Returns

search(***kwargs*)
Retrieve related nodes matching the provided properties.

Parameters **kwargs** – same syntax as *NodeSet.filter()*

Returns NodeSet

single()
Get a single related node or none.

Returns StructuredNode

class neomodel.relationship_manager.**ZeroOrMore**(*source*, *key*, *definition*)
Bases: *neomodel.relationship_manager.RelationshipManager*

A relationship of zero or more nodes (the default)

class neomodel.cardinality.**One**(*source*, *key*, *definition*)
Bases: *neomodel.relationship_manager.RelationshipManager*

A relationship to a single node

all()
Return single node in an array

Returns [node]

connect(*node*, *properties=None*)
Connect a node

Parameters

- **node** –
- **properties** – relationship properties

Returns True / rel instance

disconnect(*node*)
Disconnect a node

Parameters **node** –

Returns

disconnect_all()
Disconnect all nodes

Returns

single()
Return the associated node.

Returns node

class neomodel.cardinality.**OneOrMore**(*source*, *key*, *definition*)
Bases: *neomodel.relationship_manager.RelationshipManager*

A relationship to zero or more nodes.

all()
Returns all related nodes.

Returns [node1, node2…]

disconnect(*node*)
Disconnect node :param node: :return:

single()
Fetch one of the related nodes

Returns Node

class neomodel.cardinality.**ZeroOrOne**(*source*, *key*, *definition*)
Bases: *neomodel.relationship_manager.RelationshipManager*

A relationship to zero or one node.

all()
Return all related nodes.

Returns list

connect(*node*, *properties=None*)
Connect to a node.

Parameters

- **node** –
- **properties** – relationship properties

Type StructuredNode

Type dict

Returns True / rel instance

single()
Return the associated node.

Returns node

3.11.4 Match

class neomodel.match.**BaseSet**
Base class for all node sets.

Contains common python magic methods, __len__, __contains__ etc

all()
Return all nodes belonging to the set :return: list of nodes :rtype: list

query_cls
alias of `QueryBuilder`

class neomodel.match.**NodeSet**(*source*)
A class representing as set of nodes matching common query parameters

exclude(***kwargs*)

Exclude nodes from the NodeSet via filters.

Parameters **kwargs** – filter parameters see syntax for the filter method

Returns self

filter(***kwargs*)

Apply filters to the existing nodes in the set.

Parameters **kwargs** – filter parameters

Filters mimic Django's syntax with the double '__' to separate field and operators.

e.g *.filter(salary__gt=20000)* results in *salary > 20000*.

The following operators are available:

- 'lt': less than
- 'gt': greater than
- 'lte': less than or equal to
- 'gte': greater than or equal to
- 'ne': not equal to
- 'in': matches one of list (or tuple)
- 'isnull': is null
- 'regex': matches supplied regex (neo4j regex format)
- 'exact': exactly match string (just '=')
- 'iexact': case insensitive match string
- 'contains': contains string
- 'icontains': case insensitive contains
- 'startswith': string starts with
- 'istartswith': case insensitive string starts with
- 'endswith': string ends with
- 'iendswith': case insensitive string ends with

Returns self

first(***kwargs*)

Retrieve the first node from the set matching supplied parameters

Parameters **kwargs** – same syntax as *filter()*

Returns node

first_or_none(***kwargs*)

Retrieve the first node from the set matching supplied parameters or return none

Parameters **kwargs** – same syntax as *filter()*

Returns node or none

get(***kwargs*)

Retrieve one node from the set matching supplied parameters

Parameters **kwargs** – same syntax as *filter()*

Returns node

get_or_none (***kwargs*)
Retrieve a node from the set matching supplied parameters or return none

Parameters **kwargs** – same syntax as *filter()*

Returns node or none

has (***kwargs*)

order_by (**props*)
Order by properties. Prepend with minus to do descending. Pass None to remove ordering.

class neomodel.match.**Traversal** (*source*, *name*, *definition*)
Models a traversal from a node to another.

Parameters

- **source** (A *StructuredNode* subclass, an instance of such, a *NodeSet* instance or a *Traversal* instance.) – Starting of the traversal.
- **name** (str) – A name for the traversal.
- **definition** – A relationship definition that most certainly deserves a documentation here.

match (***kwargs*)
Traverse relationships with properties matching the given parameters.

e.g: *.match(price__lt=10)*

Parameters **kwargs** – see *NodeSet.filter()* for syntax

Returns self

3.11.5 Exceptions

exception neomodel.exceptions.**AttemptedCardinalityViolation**
Bases: *neomodel.exceptions.NeomodelException*

Attempted to alter the database state against the cardinality definitions.

Example: a relationship of type *One* trying to connect a second node.

exception neomodel.exceptions.**CardinalityViolation** (*rel_manager*, *actual*)
Bases: *neomodel.exceptions.NeomodelException*

The state of database doesn't match the nodes cardinality definition.

For example a relationship type *OneOrMore* returns no nodes.

exception neomodel.exceptions.**ConstraintValidationFailed** (*msg*)
Bases: exceptions.ValueError, *neomodel.exceptions.NeomodelException*

exception neomodel.exceptions.**DeflateConflict** (*cls*, *key*, *value*, *nid*)
Bases: *neomodel.exceptions.InflateConflict*

exception neomodel.exceptions.**DeflateError** (*key*, *cls*, *msg*, *obj*)
Bases: exceptions.ValueError, *neomodel.exceptions.NeomodelException*

exception neomodel.exceptions.**DoesNotExist** (*msg*)
Bases: *neomodel.exceptions.NeomodelException*

exception `neomodel.exceptions.`**`InflateConflict`**(*cls*, *key*, *value*, *nid*)
Bases: *`neomodel.exceptions.NeomodelException`*

exception `neomodel.exceptions.`**`InflateError`**(*key*, *cls*, *msg*, *obj=None*)
Bases: `exceptions.ValueError`, *`neomodel.exceptions.NeomodelException`*

exception `neomodel.exceptions.`**`MultipleNodesReturned`**(*msg*)
Bases: `exceptions.ValueError`, *`neomodel.exceptions.NeomodelException`*

exception `neomodel.exceptions.`**`NeomodelException`**
Bases: `exceptions.Exception`

A base class that identifies all exceptions raised by `neomodel`.

exception `neomodel.exceptions.`**`NotConnected`**(*action*, *node1*, *node2*)
Bases: *`neomodel.exceptions.NeomodelException`*

exception `neomodel.exceptions.`**`RequiredProperty`**(*key*, *cls*)
Bases: *`neomodel.exceptions.NeomodelException`*

exception `neomodel.exceptions.`**`UniqueProperty`**(*msg*)
Bases: *`neomodel.exceptions.ConstraintValidationFailed`*

CHAPTER 4

Indices and tables

- genindex
- modindex

Python Module Index

Index