neomodel

neo4j, neomodel, ogm

Connecting

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

change connection

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

환경 변수 사용

```
$ export NEO4J_USERNAME=neo4j
$ export NEO4J_PASSWORD=neo4j
$ export
NEO4J_BOLT_URL="bolt://$NEO4J_USERNAME:$NEO4J_PASSWORD@localhost:7687"
```

```
import os
from neomodel import config

config.DATABASE_URL = os.environ["NEO4J_BOLT_URL"]
```

Node Entities and Relationships

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

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

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

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')
```

property-types

Constrains and Indexes

```
$ neomodel_install_labels

usage: neomodel_install_labels [-h] [--db bolt://neo4j:neo4j@localhost:7687]
<someapp.models/app.py> [<someapp.models/app.py> ...]
```

Remove

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

Create, Update, Delete operations

```
jim = Person(name='Jim', age=3).save() # Create
jim.age = 4
jim.save() # Update, (with validation)
jim.delete()
jim.refresh() # reload properties from the database
jim.id # neo4j internal id
```

Retrieving nodes

```
# Return all nodes
all_nodes = Person.nodes.all()

# Returns Person by Person.name=='Jim' or raises neomodel.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
neomodel.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)
```

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')
result = germany.inhabitant.filter(name="Jim")
result.all()

# Find all the people called in germany except 'Jim'
germany.inhabitant.exclude(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)
```

```
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')
```

Cardinality

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

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

The following cardinality constraints are available:

- ZeroOrOne
- One
- ZeroOrMore (default)
- OneOrMore

Properties

```
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

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()
```

retrieve relationships

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

Explicit Traversal

```
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()
```

Choices

```
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'
```

Array Properties

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

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

Dates and times

```
created = DateTimeFormatProperty(format="%Y-%m-%d %H:%M:%S")

created = DateTimeProperty(default_now=True)
```

```
# Enforcing a specific timezone is done by setting the config variable
NEOMODEL_FORCE_TIMEZONE=1.
```

Complex lookups with Q objects

```
from neomodel import Q
Q(name__startswith='Py')


Lang.nodes.filter(
    Q(name__startswith='Py'),
    Q(year=2005) | Q(year=2006)
)

# This roughly translates to the following Cypher query:

MATCH (lang:Lang) WHERE name STARTS WITH 'Py'
    AND (year = 2005 OR year = 2006)
    return lang;
```

Has a relationship

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

Filtering by relationship properties

```
coffee_brand = Coffee.nodes.get(name="BestCoffeeEver")

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

Ordering by property

```
# 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)
```

Cypher queries

```
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]
```

Stand alone

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

Transactions

Basic usage

```
from neomodel import db

with db.transaction:
    Person(name='Bob').save()
@db.transaction
def update_user_name(uid, name):
    user = Person.nodes.filter(uid=uid)[0]
    user.name = name
    user.save()
db.begin()
try:
    new_user = Person(name=username, email=email).save()
    send_email(new_user)
    db.commit()
except Exception as e:
    db.rollback()
@task
@db.transaction  # comes after the task decorator
def send_email(user):
    ...
```

Explicit Transactions

```
with db.read_transaction:
    ...

with db.write_transaction:
    ...

with db.transaction:
    # This designates the transaction as WRITE even if
    # the the enclosed block of code will not modify the
    # database state.
```

```
With function decorators:

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

@db.read_transaction
def get_all_users():
    return Person.nodes.all()

@db.trasaction # Be default a WRITE transaction
...
With explicit designation:

db.begin("WRITE")
...
db.begin("READ")
...
db.begin() # By default a **WRITE** transaction
```

Hooks

Hooks

Ref

- https://neomodel.readthedocs.io/en/latest/

관련 문서

Plugin Backlinks: 아무 것도 없습니다.