neo4django Documentation

Release 0.1.8

Matt Luongo

Oct 03, 2017

Contents

neo4django is an Object Graph Mapper that let's you use familiar Django model definitions and queries against the Neo4j graph database.

You can install the latest stable release from PyPi

```
> pip install neo4django
```

or get the bleeding-edge from GitHub.

```
> pip install -e git+https://github.com/scholrly/neo4django/#egg=neo4django-dev
```

CHAPTER 1

Details

Getting Started

Once you've installed neo4django, you can configure your Django project to connect to Neo4j.

Database Setup

An example settings.py:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db', 'test_database.sqlite3')
    }
}

NEO4J_DATABASES = {
    'default' : {
        'HOST':'localhost',
        'PORT':7474,
        'ENDPOINT':'/db/data'
    }
}
```

If you'd like to use other Django apps built on the regular ORM in conjunction with neo4django, you'll still need to configure `DATABASES` with a supported database. You should also install a database router in your settings.py so the databases will play nice:

```
DATABASE_ROUTERS = ['neo4django.utils.Neo4djangoIntegrationRouter']
```

Once your project is configured, you're ready to start writing-models !

Writing Models

Models look similar to typical Django models. A neo4django model definition might look like this:

```
from neo4django.db import models

class Person(models.NodeModel):
    name = models.StringProperty()
    age = models.IntegerProperty()

    friends = models.Relationship('self',rel_type='friends_with')
```

Properties

As you can see, some basic properties are provided:

```
class OnlinePerson(Person):
    email = models.EmailProperty()
    homepage = models.URLProperty()
```

Some property types can also be indexed by neo4django. This will speed up subsequent queries based on those properties:

```
class EmployedPerson(Person):
    job_title = models.StringProperty(indexed=True)
```

All instances of `EmployedPerson` will have their `job_title` properties indexed.

For a list of included property types, check out `neo4django.db.models.__init__`.

Relationships

Relationships are simple. Instead of `ForeignKey`, `ManyToManyField`, or `OneToOneField`, just use `Relationship`. In addition to the relationship target, you can specify a relationship type and direction, cardinality, and the name of the relationship on the target model:

```
class Pet(models.NodeModel):
    owner = models.Relationship(Person,
                                rel_type='owns',
                                single=True,
                                related_name='pets'
                               )
```

Note that specifying cardinality with `single` or `related_single` is optional- Neo4j doesn't enforce any relational cardinality. Instead, these options are provided as a modeling convenience.

You can also target a model that has yet to be defined with a string:

```
class Pet(models.NodeModel):
    owner = models.Relationship('Person',
                                rel_type='owns',
                                single=True,
                                related_name='pets'
                               )
```

And then in the interpreter:

```
>>> pete = Person.objects.create(name='Pete', age=30)
>>> garfield = Pet.objects.create()
>>> pete.pets.add(garfield)
>>> pete.save()
>>> list(pete.pets.all())
[<Pet: Pet object>]
```

If you care about the order of a relationship, add the `preserve_ordering=True` option. Related objects will be retrieved in the order they were saved.

Got a few models written? To learn about retrieving data, see *Querying*.

Querying

Querying should be easy for anyone familiar with Django. Model managers return a subclass of `QuerySet` that converts queries into the Cypher graph query language, which yield `NodeModel` instances on execution.

Most of the Django QuerySet API is implemented, with exceptions noted in the project issues. In particular, the library doesn't yet support relationship-spanning lookups or complex date handling. We've also added two field lookups- *member* and *member__in*- to make searching over array properties easier. For an `OnlinePerson` instance with an `emails` property, query against the field like:

```
OnlinePerson.objects.filter(emails__member="wicked_cool_email@example.com")
```

JOINs

It's important to remember that, since we're using a graph database, "JOIN-like" operations are much less expensive. Consider a more connected model:

```
class FamilyPerson(Person):
    parents = Relationship('self', rel_type='child_of')
    stepdad = Relationship('self', rel_type='step_child_of', single=True)
    siblings = Relationship('self', rel_type='sibling_of')
    # hopefully this is one-to-one...
    spouse = Relationship('self', rel_type='married_to', single=True, rel_single=True)
```

If we'd like to pre-load a subgraph around a particular `FamilyPerson`, we can use `select_related()`:

```
jack = Person.objects.all(name='Jack').select_related(depth=5)
#OR
Person.objects.get(name='Jack').select_related('spouse__mother__sister__son__stepdad')
```

...either of which will pre-load Jack's extended family so he can go about recalling names without hitting the database a million times.

Authentication

By using a custom authentication backend, you can make use of Django's authentication framework while storing users in Neo4j.

Add `neo4django.auth` in your `INSTALLED_APPS` setting, and add:

```
AUTHENTICATION_BACKENDS = ('neo4django.auth.backends.NodeModelBackend',)
```

in your settings.py.

To create a new user, use something like:

```
user = User.objects.create_user('john', 'lennon@thebeatles.com', 'johnpassword')
```

Login, reset password, and other included auth views should work as expected. In your views, `user` will contain an instance of `neo4django.auth.models.User` for authenticated users.

Referencing Users

Other models are free to reference users. Consider:

```
from django.contrib.auth import authenticate

from neo4django.db import models
from neo4django.auth import User

class Post(models.NodeModel):
    title = models.StringProperty()
    author = models.Relationship(User, rel_type='written_by', single=True,
                                 related_name='posts')

user = authenticate(username='john', password='johnpassword')
```

```
post = Post()
post.title = 'Cool Music Post'
post.author = user
post.save

assert list(user.posts.all())[0] == post
```

Customizing Users

Swappable user models are in the works, but until then users can be customized by subclassing:

```
from neo4django.db import models
from neo4django.auth import User

class TwitterUser(User):
    follows = models.Relationship('self', rel_type='follows',
                                  related_name='followed_by')

jack = TwitterUser()
jack.username = 'jack'
jack.email = 'jack@example.com'
jack.set_password("jackpassword')
jack.save()

jim = TwitterUser()
jim.username = 'jim'
jim.email = 'jim@example.com'
jim.set_password('jimpassword')
jim.follows.add(jack)
jim.save()
```

The caveats are, first, that `User` manager shortcuts, like `create_user()`, aren't available, and that `authenticate()` and other included functions to work with users will return the wrong model type. This is fairly straightforward to handle, though, using the included convenience method `from_model()`:

```
from django.contrib.auth import authenticate

user = authenticate(username='jim', password='jimpassword')
twitter_user = TwitterUser.from_model(user)
```

Permissions

Because neo4django doesn't support `django.contrib.contenttypes` or an equivalent, user permissions are not supported. Object-specific or contenttypes-style permissions would be a great place to contribute.

Writing Django Tests

There is a custom test case included which you can use to write Django tests that need access to `NodeModel` instances. If properly configured, it will wipe out the Neo4j database in between each test. To configure it, you must set up a Neo4j instance with the cleandb extension installed. If your neo4j instance were configured at port 7475, and your

cleandb install were pointing to `/cleandb/secret-key`, then you would put the following into your `settings.py`:

```
NEO4J_TEST_DATABASES = {
    'default': {
        'HOST': 'localhost',
        'PORT': 7475,
        'ENDPOINT': '/db/data',
        'OPTIONS': {
            'CLEANDB_URI': '/cleandb/secret-key',
            'username': 'lorem',
            'password': 'ipsum',
        }
    }
}
```

With that set up, you can start writing test cases that inherit from `neo4django.testcases.NodeModelTestCase` and run them as you normally would through your Django test suite.

Debugging & Optimization

A django-debug-toolbar panel has been written to make debugging Neo4j REST calls easier. It should also help debugging and optimizing neo4django.

`neo4django.testcases.NodeModelTestCase.assertNumRequests()` can also help by ensuring round trips in a piece of test code don't grow unexpectedly.

Multiple Databases & Concurrency

Multiple Databases

neo4django was written to support multiple databases- but that support is untested. In the future, we'd like to fully support multiple databases and routing similar to that already in Django. Because most of the infrastucture is complete, robust support would be a great place to contribute.

Concurrency

Because of the difficulty of transactionality over the REST API, using neo4django from multiple threads, or connecting to the same Neo4j instance from multiple servers, is not recommended without serious testing.

That said, a number of users do this in production. Hotspots like type hierarchy management are transactional, so as long as you can separate the entities being manipulated in the graph, concurrent use of neo4django is possible.

Running the Test Suite

The test suite requires that Neo4j be running, and that you have the cleandb extension installed at `localhost:<NEO4J_PORT>/cleandb`.

We test with nose. To run the suite, set `test_settings.py` as your `DJANGO_SETTINGS_MODULE` and run `nosetests`. In bash, that's simply:

```
cd <your path>/neo4django/
export DJANGO_SETTINGS_MODULE="neo4django.tests.test_settings"
nosetests
```

We've put together a nose plugin to ensure that regression tests pass. Any changesets that fail regression tests will be denied a pull. To run the tests, simply:

```
pip install nose-regression
nosetests --with-regression
```

CHAPTER 2

Contributing

We love contributions, large or small. The source is available on GitHub- fork the project and submit a pull request with your changes.

Uncomfortable / unwilling to code? If you'd like, you can give a small donation on Gittip to support the project.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search