Implementing git-orm

https://github.com/natano/python-git-orm/

Martin Natano

June 06, 2013

whoami

Martin Natano

- Web Developer @RadarServices
- previously: Medical University of Vienna, Mjam

2 / 34

Hey everyone!

My name is Martin Natano. I'm currently working at RadarServices and previously at the Medical University of Vienna and Mjam.

I'd like to present git-orm today.

note 1 of slide 2

Description

- □ django-esque model interface for storing objects in a git repository
- □ ISC licensed
- □ written in python

3 / 34

Greeting

git-orm is a django-esque model interface for storing object in a git repository. It's written in the python programming language and freely available under the ISC license.

note 1 of slide 3

Models

Listing 1: Python

```python
from git_orm import models

class User(models.Model):
    email = models.TextField(primary_key=True)
    name = models.TextField(null=True)

class Article(models.Model):
    author = models.ForeignKey(User)
    summary = models.TextField()
    content = models.TextField()
```

5 / 34

What does it look like?

A model definition looks like this.

Here we define a user and an article model. The user model has an explicit primary key.

Every model that doesn't define a primary key (like our Article model) will have one added automatically, namly a random UUID.

note 1 of slide 5

Creating an object

Listing 2: Python

```
hansel = User()
hansel.email = 'hansel@example.com'
hansel.name = 'Hans'
hansel.save()

# or

gretel = User.create(email='grete@example.com',
                     name='Grete')
```


Creating an object

Creating an object is easy. Just create an instance of the model class, assign some attributes and save it.

Alternatively we could do it in one call with the "create" method.

Every change creates a commit in the git repository. The path of an object in the repository consists of the model name and the value of the primary key field.

Timestamps

Listing 3: Python

```
>>> gretel.created_at
datetime.datetime(2013, 6, 6, 12, 27, 35, 276071)
>>> gretel.updated_at
datetime.datetime(2013, 6, 6, 10, 08, 33, 3)
```

7 / 34

Every objects has two timestamps: "created_at" and "updated_at". They are added automatically.

note 1 of slide 7

Expressive Syntax

Listing 4: Python

```
User.objects.get(email='user@example.org')
User.objects.exists()
Article.objects.all()
Article.objects.count()
Article.objects.filter(author_email='user@example.org')
Article.objects.exclude(summary__contains='vienna.rb')
```

9 / 34

Querying

Querying is easy too. There are the methods "all", "filter", "exclude", "order_by", "exists", "get" and "count".

note 1 of slide 9

Chaining

Listing 5: Python

```
(Article.objects
    .exclude(author_email='user@example.org')
    .filter(summary__icontains='vienna.rb')
    .count())
User.objects.order_by('email')[:10]
```

10 / 34

Chaining

Most of them are chainable. What you see on the screen are perfectly valid statements.

Btw, slicing is also possible. (and chainable too)

note 1 of slide 10

Lazy Evaluation

Listing 6: Python

```
articles = Article.objects.all()
articles = articles.filter(published_at__lt=now)
articles.count()
```

11 / 34

Lazyness

All chainable methods are lazyly evaluated. So in our example the the first line produces a queryset that containes the conditions for the query we finally want to produce. The second line adds a condition to the query and the last line executes it.

note 1 of slide 11

Q Objects (Advanced Querying)

Listing 7: Python

```
from git_orm.models import Q

User.objects.filter(
    Q(email__endswith='@example.org')
    |Q(name__icontains='grete')
)
```

13 / 34

Q Objects

Q objects are the centerpiece of queriesets. They represent a condition of a query. Direct use of them allows for complex queries that would not be possible with just using the "filter" and "exclude" methods.

note 1 of slide 13

Q Objects (connect 'em)

Listing 8: Python

```python
from git_orm.models import Q

~Q(name='Hexe') & (
    Q(email='grete@example.com') | Q(name='Hansel'))
```

14 / 34

Q Objects

Possible operations are "and", "or", "not" and grouping with parentheses. Operator precedence is that of the basic operations in python. Just use parentheses when in doubt.

git-orm includes some basic query optimizations for minimizing query execution time.

note 1 of slide 14

Automatic Transaction Management

Listing 9: Python

```python
from git_orm import transaction

with transaction.wrap():
    ...
```

16 / 34

Transactions (as a context manager)

Transactions can help with grouping a collection of changes into one commit in the repository, so all of that changes are commited at once or none of them is.

note 1 of slide 16

Automatic Transaction Management

Listing 10: Python

```
@transaction.wrap()
def persist():
    ...
```

17 / 34

Transactions (as a context manager)

When using transaction.wrap as a decorator you can wrap a whole function into a transaction

note 1 of slide 17

Manual Transaction Management

Listing 11: Python

```python
from git_orm import transaction

transaction.begin()
...
transaction.commit()

transaction.begin()
...
transaction.rollback()
```

18 / 34

Transactions (manually)

If the automatic approach is not flexible enough you can fall back to the manual transaction handling primitives. When not using transactions at all, every update will be performed in a separate git commit.

note 1 of slide 18

Syntactic Sugar

19 / 34

For a nice API I implemented some syntactic sugar. As this is a ruby meetup I will show the ruby equivalent for every pattern.

note 1 of slide 19

Context Managers

20 / 34

Context Managers

Listing 12: Python

```
with transaction.wrap() as trans:
    ...
```

Listing 13: Ruby

```
Transaction.wrap do |trans|
    ...
end
```

21 / 34

git-orm uses context managers for transactions. Python has a dedicated syntax for context managers. In ruby we have the more generic syntax of passing a block to a function.

note 1 of slide 21

Context Managers

Listing 14: Python

```
class wrap:
    ...
    def __enter__(self):
        begin()
        return _transaction

    def __exit__(self, type, value, traceback):
        if not type and _transaction.has_changes:
            commit(self.message)
        else:
            rollback()
    ...
```

22 / 34

Before the code in the with block is executed, the __enter__ method is called. After the block finished (or raised an exception) the __exit__ method is called.

note 1 of slide 22

Decorators

Decorators

Listing 15: Python

```python
@transaction.wrap()
def persist:
    ...
```

Listing 16: Ruby

```ruby
def persist
    # ??
end
```

24 / 34

transaction.wrap is also available as a decorator. A decorator wraps a function inside of another one and updates the variable containing the inner function with the result. Is there a closer equivalent for decorators in Ruby? Please tell me!

note 1 of slide 24

Decorators

Listing 17: Python

```
class wrap:
    ...
    def __call__(self, fn):
        @wraps(fn)
        def _inner(*args, **kwargs):
            with self:
                return fn(*args, **kwargs)
        return _inner
    ...
```

25 / 34

A decorator is a callable that takes a callable as an argument and returns another one.

note 1 of slide 25

Operator Overloading

26 / 34

Operator Overloading

Listing 18: Python

```
Q(...) & Q(...) | Q(...)
```

Listing 19: Ruby

```
Q(...) & Q(...) | Q(...)
```

27 / 34

I implemented operator overloading for the Q objects and Querysets for constructing queries.

note 1 of slide 27

Metaclasses

28 / 34

Metaclasses

Listing 20: Python

```
class Foo(object):
    __metaclass__ = Bar
```

Listing 21: Ruby

```
class Foo
    extend Bar
end
```

29 / 34

In python classes are objects as every other object. Every class is the instance of a metaclass. Most classes use "type" as metaclass.

The two code samples are not equivalent because I would have used another approach implementing Models in Ruby then in python.

note 1 of slide 29

Bugs & Shortcomings

30 / 34

Bugs & Shortcomings

- ☐ python 3 only
- ☐ delete not implemented yet
- ☐ concurrent transactions are not handled correctly

31 / 34

Up next

Up next & Ideas

- ☐ python 2 & 3 support with six
- ☐ merging of concurrent transactions
- ☐ more query optimizations

Thx!

```
$ pip install git-orm
```