mongopersist

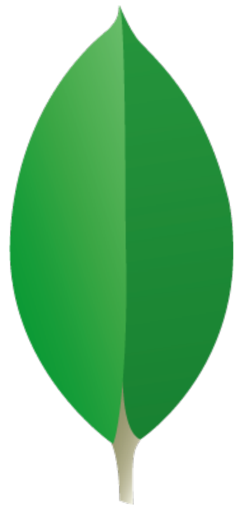
Adam GROSZER

https://github.com/zopefoundation/mongopersist
https://pypi.python.org/pypi/mongopersist
slideshare URL
https://github.com/agroszer
http://hu.linkedin.com/in/agroszer/

contents

- mongoDB
- databases
- persistence
- mongopersist features

mongopersist



mongoDB

choose your database, choose your future

	pro	con
ZODB	- very transparent - object store - only python/native	- no query lang - no 3rd party tools - no default indexing
RDBMS ORM	- ad-hoc SQL queries - indexes, tools, etc.	- **BIG** impedance mismatch - limited transparency - strict schema
mongoDB	- document store - ad-hoc queries - indexes, tools, etc.	- **small** impedance mismatch - limited/no transparency

persistence

state that outlives the process

- get the object
- modify
- (finish the transaction)

persistence

hand-crafted example, using pymongo:

```
>>> stephan = db.coll.find_one(
...     {'name': 'Stephan'})
>>> stephan['phone'] = {
...     'country': '+1',
...     'area': '555',
...     'number': '3945456'}
>>> db.coll.save(stephan)
```

persistence

ideal case with mongopersist:

```
>>> persons = dm.root
>>> stephan = persons['stephan']
>>> stephan.phone = Phone(
...     '+1', '555', '3945456')
>>> transaction.commit()
```

mongopersist features

- transparency
- ~~transactions~~ Optimistic Data Dumping
- write conflict detection
- custom de/serialization
- object caching
- query logging, incl. traceback
- (not just) ZTK corner

sample class

```
class Person(persistent.Persistent):
    def __init__(self, name, phone=None,
                 address=None, friends=None,
                 visited=(), birthday=None):
        self.name = name
        self.address = address
        self.friends = friends or {}
        self.visited = visited
        self.phone = phone
        self.birthday = birthday
        self.today = datetime.datetime.now()
...
```

transparency

setup code:

```
>>> conn = pymongo.Connection(
...     'localhost', 27017, tz_aware=False)

>>> from mongopersist import datamanager
>>> dm = datamanager.MongoDataManager(conn)
```

transparency

```
>>> dm.root

>>> stephan = Person(u'Stephan')
>>> dm.root['stephan'] = stephan

>>> stephan = dm.root['stephan']
>>> stephan.name = u'Stephan Richter'

>>> transaction.commit()
```

defaults

```
>>> dumpCollection('__main__.Person')
[{u'_id': ObjectId('...'),
  u'address': None,
  u'birthday': None,
  u'friends': {},
  u'name': u'Stephan Richter',
  u'phone': None,
  u'today': datetime.datetime(2013, 6, 18,
14, 48, 30, 970000),
  u'visited': []}]
```

customizing

```
class Address(persistent.Persistent):
    _p_mongo_collection = 'address'

    def __init__(self, city, zip):
        self.city = city
        self.zip = zip
```

sub-objects via DBRef

```
>>> stephan.address = Address('Maynard', '01754')
>>> transaction.commit()

>>> dumpCollection('address')
[{u'_id': ObjectId('...'),
  u'city': u'Maynard',
  u'zip': u'01754'}]

>>> dumpCollection('__main__.Person')
[{u'_id': ObjectId('...'),
  u'address': DBRef(u'address', ObjectId('...'),
  u'mongopersist_test'), ...}]
```

collection sharing

```
class Person(persistent.Persistent):
    _p_mongo_collection = 'person'
    name = u''
    ...

class Employee(Person):
    _p_mongo_collection = 'person'
    salary = 0
    ...
```

mongopersist will automatically notice these cases and stores the Python type as part of the document

sub object/document

```
class Car(persistent.Persistent):
    _p_mongo_sub_object = True
    def __init__(self, year, make, model):
        self.year = year
        self.make = make
        self.model = model

>>> dm.root['stephan'].car = Car('2005', 'Ford', 'Explorer')
>>> dumpCollection('__main__.Person')
[{...
  u'car': {u'_py_persistent_type': u'__main__.Car',
           u'make': u'Ford',
           u'model': u'Explorer',
           u'year': u'2005'}, ...}]
```

beware of non Persistent objects

```
class Phone(object):
    def __init__(self, country, area, number):
        ...

>>> stephan.phone = Phone('+1', '978', '394-5124')
>>> dumpCollection('__main__.Person')
[{...
  u'phone': {u'_py_type': u'__main__.Phone',
             u'area': u'978',
             u'country': u'+1',
             u'number': u'394-5124'}, ...]
>>> stephan.phone.number = '555-1234'
>>> transaction.commit()
```

Changes not saved, because not subclassing Persistent

add/delete property

```
>>> stephan.foobar = 42
>>> transaction.commit()
```

No declaration needed!

```
>>> dumpCollection('__main__.Person')
[{...
  u'foobar': 42,
  u'name': u'Stephan',
  ...}]

>>> del stephan.foobar
>>> transaction.commit()
```

custom property

```
>>> stephan.friends[u'roger'] = Person(u'Roger')
>>> stephan.visited.append('Italy')
>>> transaction.commit()
>>> dumpCollection('__main__.Person')
[{...
  u'friends': {u'roger': DBRef(u'__main__.Person',
               ObjectId('...'), u'mongopersist_test')},
  u'visited': [u'Italy']},
 {... u'name': u'Roger', ...]
>>> stephan.friends[u'roger'].name
u'Roger'
```

Circular references:
- Persistent: OK
- non-Persistent: no-go!

Optimistic Data Dumping

The process of dumping data during a transaction under the assumption the transaction will succeed.

object modifications
...
object modifications
...
automatic/implicit flush
query

Optimistic Data Dumping

```
>>> stephan.foobar = 42
...code...
>>> roy.foobar = 88
...code...
>>> dm.get_collection_from_object(
...     roy).count({'foobar': 88})
1
```

ALL query methods are wrapped to call flush first

write conflict detection

`_py_serial`

- Conflict detection
- Conflict resolution

Handlers:

- `NoCheckConflictHandler`
  - Ignore conflicts, last flush wins, default
- `SimpleSerialConflictHandler`
  - Detects conflicts, always raise `ConflictError`
- `ResolvingSerialConflictHandler`
  - Detects conflicts, calls `_p_resolveConflict()`

custom serializers

```
stephan.birthday = datetime.date(1980, 1, 25)

u'birthday': {u'_py_factory': u'datetime.date',
              u'_py_factory_args': [Binary('\x07\xbc\x01\x19', 0)]},

class DateSerializer(serialize.ObjectSerializer):
    def can_read(self, state):
        return isinstance(state, dict) and \
               state.get('_py_type') == 'datetime.date'
    def read(self, state):
        return datetime.date.fromordinal(state['ordinal'])
    def can_write(self, obj):
        return isinstance(obj, datetime.date)
    def write(self, obj):
        return {'_py_type': 'datetime.date', 'ordinal': obj.toordinal()}

>>> serialize.SERIALIZERS.append(DateSerializer())

u'birthday': {u'_py_type': u'datetime.date', u'ordinal': 722839},
```

querying mongoDB

```
datamanager.get_collection(dbname, collname)
datamanager.get_collection_from_object(obj)
```

- `find`, `find_one`, `count`, etc

extra methods, which return objects:

- `find_objects()`
- `find_one_object()`

ALL query methods are wrapped to call flush first

```
datamanager.load(dbref)
```

object caching

DB access and object instantiation is quite slow

- class Lookup Cache: dbref → python class lookup
- object Cache: dbref → object (within transaction)
- document Cache: dbref → document (avoid DB trip)

query logging, incl. traceback

`LoggingDecorator`: Logs the calls to

`insert`, `update`, `remove`, `save`, `find`, `find_one`, `find_and_modify`, `count`

including args and kwargs.

With optional traceback.

containers and collections

```
class People(MongoCollectionMapping):
    __mongo_collection__  = '__main__.Person'
    __mongo_mapping_key__ = 'name'
```

- Mapping/dict API for a Mongo collection.
- Specify the collection to use for the mapping
- Specify the attribute that represents the dictionary key.

(not just) ZTK corner

`Contained`, `Container`, `__name__`, `__parent__`

`zope.container.MongoContained`

`zope.container.MongoContainer`
- mapping/dict interface
- and more: `add`, `(raw_)find`, `(raw_)find_one`

`zope.container.IdNamesMongoContainer`
- uses the item's ObjectID as key

(not just) ZTK corner

connection pooling

app setup:

```
mdmp = pool.MongoDataManagerProvider(
    host='localhost', port=27017,
    logLevel=20, tz_aware=True, w=1, j=True)
zope.component.provideUtility(mdmp)
```

request setup:

```
mdmp = getUtility(interfaces.IMongoDataManagerProvider)
dm = mdmp.get()
```

(not just) ZTK corner

`IMongoAttributeAnnotatable`
- storing metadata

DublinCore
- standard for metadata

Q&A