The Object-Document Mapper (ODM)

In addition to the basic interface we've covered so far, Ming provides a higher-level abstraction for modeling objects in MongoDB.

The ODM provides several features beyond those provided by the basic Ming modules:

- **unit-of-work pattern** Using ODM-enabled sessions allows you to operate on in-memory objects exclusively until you are ready to "flush" all changes to the database. Although MongoDB doesn't provide transactions, the unit-of-work (UOW) pattern provides some of the benefits of transactions by delaying writes until you are fairly certain nothing is going to go wrong.
- **identity map** In base Ming, each query returns unique document objects even if those document objects refer to the same document in the database. The identity map in Ming ensures that when two queries each return results that correspond to the same document in the database, the queries will return the same Python object as well.
- **relations between objects** Although MongoDB is non-relational, it is still useful to represent relationships between documents in the database. The ODM layer in Ming provides the ability to model one-to-many relationships between documents as straightforward properties on Python objects.

Adding an ODM layer to our model

To see how the ODM works, we'll start by adding an ODM layer to our forum model. First, let's see how we update the `Forum`:

In [1]:
```
from lesson_2_0 import model as M

# Clear out the database and session first
M.sess.db.drop_collection('forum.forum')
M.sess.db.drop_collection('forum.thread')
M.sess.db.drop_collection('forum.post')
```

In [2]:
```
from ming.odm import mapper, Mapper

class Forum(object):
    pass

mapper(Forum, M.Forum)
Mapper.compile_all()
```

Here, we have just created a linkage between the "plain old Python object" (POPO) `Forum` and the document class we created earlier `M.Forum`. Exploring the Forum object reveals several new fields:

In [3]:
```
f = Forum(name='My ODM Forum')
f
```

Out[3]:
```
<Forum num_threads=None name='My ODM Forum'
  created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118510)
  num_posts=None _id=<Missing> last_post={'when': None,
  'user': None, 'subject': None} description=None>
```

ODMSession and the unit of work

Note that all the fields we defined in lesson 2.0 are present on the Forum object. By default, the `Forum` object we created above is ephemeral: it will not be persisted to the database. In order to make sure it's persisted we'll need to put it in an `ODMSession`:

In [4]:
```
# Clear out the database first
```

In [4]:
```
# clear out the database first
M.sess.db['forum.forum'].remove()

from ming.odm import ODMSession
sess = ODMSession(M.sess)
sess
```

Out[4]:
```
<session>
  <UnitOfWork>
    <new>
    <clean>
    <dirty>
    <deleted>
  <imap (0)>
```

In [5]:
```
sess.save(f)
sess
```

Out[5]:
```
<session>
  <UnitOfWork>
    <new>
      <Forum num_threads=None name='My ODM Forum'
          created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118510)
          num_posts=None _id=ObjectId('4f57a7f2eb0330109d000000')
          last_post=I{'when': None, 'user': None, 'subject': None}
          description=None>
    <clean>
    <dirty>
    <deleted>
  <imap (1)>
    Forum : 4f57a7f2eb0330109d000000 => <Forum num_threads=None name='My ODM Forum'
        created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118510)
        num_posts=None _id=ObjectId('4f57a7f2eb0330109d000000')
        last_post=I{'when': None, 'user': None, 'subject': None}
        description=None>
```

Now that we have `save`d the forum into the `ODMSession`, it is part of the unit of work. To save everything in a session's unit of work to the database, we need fo `flush` the session:

In [6]:
```
sess.flush()
M.Forum.m.find().one()
```

Out[6]:
```
{'_id': ObjectId('4f57a7f2eb0330109d000000'),
 'created': datetime.datetime(2012, 3, 7, 18, 24, 50, 118000),
 'description': None,
 'last_post': {u'subject': None, u'user': None, u'when': None},
 'name': u'My ODM Forum',
 'num_posts': None,
 'num_threads': None}
```

Now if we examine the session we'll note that it has the newly-inserted forum in the 'clean' state:

In [7]:
```
sess
```

Out[7]:
```
<session>
  <UnitOfWork>
    <new>
    <clean>
```

```
    <Forum num_threads=None name='My ODM Forum'
        created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118510)
        num_posts=None _id=ObjectId('4f57a7f2eb0330109d000000')
        last_post=I{'when': None, 'user': None, 'subject': None}
        description=None>
  <dirty>
  <deleted>
<imap (1)>
  Forum : 4f57a7f2eb0330109d000000 => <Forum num_threads=None name='My ODM Forum'
      created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118510)
      num_posts=None _id=ObjectId('4f57a7f2eb0330109d000000')
      last_post=I{'when': None, 'user': None, 'subject': None}
      description=None>
```

Modifying one of our mapped properties (such as num_posts) will mark the object as `dirty` in the session and requiring a flush:

In [8]:
```
f = sess.find(Forum).one()
f.num_posts = 5
sess
```

Out[8]:
```
<session>
  <UnitOfWork>
    <new>
    <clean>
    <dirty>
      <Forum num_threads=None name='My ODM Forum'
          created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118510)
          num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
          last_post=I{'when': None, 'user': None, 'subject': None}
          description=None>
    <deleted>
  <imap (1)>
    Forum : 4f57a7f2eb0330109d000000 => <Forum num_threads=None name='My ODM Forum'
        created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118510)
        num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
        last_post=I{'when': None, 'user': None, 'subject': None}
        description=None>
```

Flush again, and it's clean (and the database is updated).

In [9]:
```
sess.flush()
sess
```

Out[9]:
```
<session>
  <UnitOfWork>
    <new>
    <clean>
      <Forum num_threads=None name='My ODM Forum'
          created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118510)
          num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
          last_post=I{'when': None, 'user': None, 'subject': None}
          description=None>
    <dirty>
    <deleted>
  <imap (1)>
    Forum : 4f57a7f2eb0330109d000000 => <Forum num_threads=None name='My ODM Forum'
        created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118510)
        num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
```

```
                last_post=I{'when': None, 'user': None, 'subject': None}
                description=None>
```

In [10]:
```
M.Forum.m.find().one()
```

Out[10]:
```
{'_id': ObjectId('4f57a7f2eb0330109d000000'),
 'created': datetime.datetime(2012, 3, 7, 18, 24, 50, 118000),
 'description': None,
 'last_post': {u'subject': None, u'user': None, u'when': None},
 'name': u'My ODM Forum',
 'num_posts': 5,
 'num_threads': None}
```

Queries

Of course, inserting data only to retrieve it later via the 'old' interface isn't very useful, so we can use the session to load objects into memory as well:

In [11]:
```
sess = ODMSession(M.sess)
sess.find(Forum).all()
```

Out[11]:
```
[<Forum num_threads=None name=u'My ODM Forum'
  created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118000)
  num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
  last_post=I{u'when': None, u'user': None, u'subject':
  None} description=None>]
```

In [12]:
```
sess
```

Out[12]:
```
<session>
  <UnitOfWork>
    <new>
    <clean>
      <Forum num_threads=None name=u'My ODM Forum'
          created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118000)
          num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
          last_post=I{u'when': None, u'user': None, u'subject':
          None} description=None>
    <dirty>
    <deleted>
  <imap (1)>
    Forum : 4f57a7f2eb0330109d000000 => <Forum num_threads=None name=u'My ODM Forum'
        created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118000)
        num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
        last_post=I{u'when': None, u'user': None, u'subject':
        None} description=None>
```

Here, we see that the forum has been loaded into the session.

If we want a session to completely *forget* about an object, we can `expunge` it from the session:

In [13]:
```
f = sess.find(Forum).one()
sess.expunge(f)
sess
```

Out[13]:
```
<session>
```

```
<UnitOfWork>
  <new>
  <clean>
  <dirty>
  <deleted>
<imap (0)>
```

This can be useful during exception handling. For instance, suppose we want to define an `ensure_present` function that will insert a new object if its `_id` isn't present in the collection, or update the existing object if it is:

In [14]:
```
import pymongo.errors

def ensure_present(sess, cls, spec):
    '''Ensure that an object with the given spec is present in the
    database. This may entail (worst case) a find, insert, exception,
    and another find.'''

    obj = sess.find(cls, dict(_id=spec['_id'])).first()
    if obj is None:
        try:
            obj = cls(**spec)
            sess.flush(obj)
            return obj
        except pymongo.errors.DuplicateKeyError:
            obj = sess.find(cls, dict(_id=spec['_id'])).first()
    for k,v in spec.items():
        setattr(obj, k, v)
    return obj
```

Identity Map

To illustrate the identity map, let's load our forum twice:

In [15]:
```
sess = ODMSession(M.sess)
f0 = sess.find(Forum).one()
f1 = sess.find(Forum).one()
f0 is f1
```

Out[15]:
```
True
```

The nice thing about the identity map is that, no matter how we get a reference to an object (within a session), the same object in the database is the same object in Python. One interesting effect of this is that, if we load a forum one way, modify it, and then load the forum again, we get the *modified* version. This is to prevent inadvertently overwriting our in-memory changes:

In [16]:
```
sess = ODMSession(M.sess)
f0 = sess.find(Forum).one()
f0.name = 'My Modified ODM Session'
f1 = sess.find(Forum).one()
f0.name, f1.name
```

Out[16]:
```
('My Modified ODM Session', 'My Modified ODM Session')
```

If we wish to *refresh* the value from the session we can explicitly refresh it:

In [17]:
```
f2 = sess.find(Forum, refresh=True).one()
f0.name, f1.name, f2.name
```

Out[17]:
```
(u'My ODM Forum', u'My ODM Forum', u'My ODM Forum')
```

The `ODMSession` also provides a `get` method to load a single object from the identity map without even hitting the database:

In [18]:
```
sess.get(Forum, f0._id)
```

Out[18]:
```
<Forum num_threads=None name=u'My ODM Forum'
  created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118000)
  num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
  last_post=I{u'when': None, u'user': None, u'subject':
  None} description=None>
```

Thread- (or context-) local ODM sessions

Constantly having to declare a new session to do anything with the model can be somewhat tedious. To simplify this, Ming allows us to use a global `ThreadLocalODMSession` which can be *bound* to individual mappers, providing some helpers on the object itself:

In [19]:
```
from ming.odm import ThreadLocalODMSession

sess = ThreadLocalODMSession(M.sess)

sess.mapper(Forum, M.Forum)
```

Out[19]:
```
<Mapper Forum:forum.forum>
```

In [20]:
```
Forum.query.find().all()
```

Out[20]:
```
[<Forum num_threads=None name=u'My ODM Forum'
  created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118000)
  num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
  last_post=I{u'when': None, u'user': None, u'subject':
  None} description=None>]
```

By using the `ThreadLocalODMSession.mapper` instead of the plain `mapper`, we note that all ODM access for this class will happen through the given session. We also gain the feature that objects are automatically `saved` to the session upon creation:

In [21]:
```
Forum(name='My Second Forum')
```

Out[21]:
```
<Forum num_threads=None name='My Second Forum'
  created=datetime.datetime(2012, 3, 7, 18, 24, 50, 525288)
  num_posts=None _id=ObjectId('4f57a7f2eb0330109d000002')
  last_post=I{'when': None, 'user': None, 'subject': None}
  description=None>
```

In [22]:
```
sess
```

Out[22]:
```
TLProxy of <session>
  <UnitOfWork>
    <new>
      <Forum num_threads=None name='My Second Forum'
          created=datetime.datetime(2012, 3, 7, 18, 24, 50, 525288)
```

```
                num_posts=None _id=ObjectId('4f57a7f2eb0330109d000002')
                last_post=I{'when': None, 'user': None, 'subject': None}
                description=None>
          <clean>
            <Forum num_threads=None name=u'My ODM Forum'
                created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118000)
                num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
                last_post=I{u'when': None, u'user': None, u'subject':
                None} description=None>
          <dirty>
          <deleted>
        <imap (2)>
          Forum : 4f57a7f2eb0330109d000000 => <Forum num_threads=None name=u'My ODM Forum'
              created=datetime.datetime(2012, 3, 7, 18, 24, 50, 118000)
              num_posts=5 _id=ObjectId('4f57a7f2eb0330109d000000')
              last_post=I{u'when': None, u'user': None, u'subject':
              None} description=None>
          Forum : 4f57a7f2eb0330109d000002 => <Forum num_threads=None name='My Second
      Forum'
              created=datetime.datetime(2012, 3, 7, 18, 24, 50, 525288)
              num_posts=None _id=ObjectId('4f57a7f2eb0330109d000002')
              last_post=I{'when': None, 'user': None, 'subject': None}
              description=None>
```

The `ThreadLocalODMSession` is particularly useful in multithreaded web servers, where you want a psuedo-global `ODMSession` for storing the objects loaded during a request. In this case, at the end of each request, you should make sure the session is closed so objects don't "leak" from one request to another. The `ThreadLocalODMSession` provides a helper for doing just that:

In [23]:
```
sess.close()
```

Multiple sessions

There may be instances where you need multiple `ODMSession`s in a single request. In this case, you should use the classmethods provided with `ThreadLocalODMSession` to ensure that *all* sessions for the current thread are flushed and closed:

In [24]:
```
ThreadLocalODMSession.flush_all()
ThreadLocalODMSession.close_all()
```

WSGI Middleware

If you are using a WSGI-compatible Python web framework, you may be interested in the `ming.odm.middleware`, which will automatically flush and close sessions on a per-request basis (unless there is an exception raised, when the session will just be closed). The psuedocode for the `ming.odm.middleware.MingMiddleware` is as follows:

```
def make_middleware(app,
    flush_on_errors=(webob.exc.HTTPRedirection,)
    def flushing_middleware(environ, start_response):
        try:
            result = app(environ, start_response)
            ThreadLocalODMSession.flush_all()
            return result
        except flush_on_errors, err:
            ThreadLocalODMSession.flush_all()
            return result
        except:
```

```
        raise
    finally:
        ThreadLocalODMSession.close_all()
```

Using this middleware gives a sort of psuedo-transaction within your request, since all changes are pending in RAM until you reach the flush state. Note that this is not a *true* transaction, as there could be errors while flushing the session (`DuplicateKeyError`, for instance).

Relations

One of the nicest features of the ODM is the ability to add psuedo-relational features to a Ming model. Revisiting our full forum model, let's go ahead and create all three levels of POPO:

In [25]:
```
# Clear out the database and session first
M.sess.db.drop_collection('forum.forum')
M.sess.db.drop_collection('forum.thread')
M.sess.db.drop_collection('forum.post')

sess.close_all()

from ming.odm import ThreadLocalODMSession, Mapper
from ming.odm import ForeignIdProperty, RelationProperty

from lesson_2_0 import model as M

sess = ThreadLocalODMSession(M.sess)

class Forum(object):
    pass

class Thread(object):
    pass

class Post(object):
    pass
```

Now, we'll create the mappers, but we'll add an additional argument to the mapper: `property`:

In [26]:
```
Mapper.clear_all()

sess.mapper(Forum, M.Forum, properties=dict(
        threads=RelationProperty('Thread')))
sess.mapper(Thread, M.Thread, properties=dict(
        forum_id=ForeignIdProperty('Forum'),
        forum=RelationProperty('Forum'),
        posts=RelationProperty('Post')))
sess.mapper(Post, M.Post, properties=dict(
        forum_id=ForeignIdProperty('Forum'),
        thread_id=ForeignIdProperty('Thread'),
        forum=RelationProperty('Forum'),
        thread=RelationProperty('Thread')))

Mapper.compile_all()
```

What we've done here is tell Ming how the objects *relate* to one another. For instance, the `forum_id` property in threads and posts represents something like a `FOREIGN KEY` constraint in a relational database (though MongoDB does not enforce this). Once Ming knows how objects relate to one another, it can create some convenient properties. Let's see what happens when we populate some posts:

In [27]:
```
# Create a forum with some threads and posts
f = Forum(name='My ODM Forum')
t0 = Thread(forum_id=f._id, subject='First Thread')
Post(forum_id=f._id, thread_id=t0._id, subject='Frist psot!',
     slug='abcd', fullslug='abcd')
Post(forum_id=f._id, thread_id=t0._id, subject='I like PyCon',
     slug='bcde', fullslug='bcde')
t1 = Thread(forum_id=f._id, subject='Second Thread')
Post(forum_id=f._id, thread_id=t1._id, subject='Post in another thread',
     slug='cdef', fullslug='cdef')

sess.flush_all()
sess.close_all()
```

Now let's load the forum from the DB again...

In [28]:
```
f = Forum.query.get()
f
```

Out[28]:
```
<Forum num_threads=None name=u'My ODM Forum'
  created=datetime.datetime(2012, 3, 7, 18, 24, 50, 683000)
  num_posts=None _id=ObjectId('4f57a7f2eb0330109d000003')
  last_post=I{u'when': None, u'user': None, u'subject':
  None} description=None>
```

In [29]:
```
f.threads
```

Out[29]:
```
I[<Thread forum_id=ObjectId('4f57a7f2eb0330109d000003')
  num_posts=None _id=ObjectId('4f57a7f2eb0330109d000006')
  last_post=I{u'when': None, u'user': None, u'subject':
  None} subject=u'First Thread'>, <Thread
forum_id=ObjectId('4f57a7f2eb0330109d000003')
  num_posts=None _id=ObjectId('4f57a7f2eb0330109d00000c')
  last_post=I{u'when': None, u'user': None, u'subject':
  None} subject=u'Second Thread'>]
```

In [30]:
```
f.threads[0].posts
```

Out[30]:
```
I[<Post fullslug=u'bcde' parent_id=None
  timestamp=datetime.datetime(2012, 3, 7, 18, 24, 50,
  684000) forum_id=ObjectId('4f57a7f2eb0330109d000003')
  thread_id=ObjectId('4f57a7f2eb0330109d000006')
  _id=ObjectId('4f57a7f2eb0330109d00000b') slug=u'bcde'
  subject=u'I like PyCon'>, <Post fullslug=u'abcd' parent_id=None
  timestamp=datetime.datetime(2012, 3, 7, 18, 24, 50,
  684000) forum_id=ObjectId('4f57a7f2eb0330109d000003')
  thread_id=ObjectId('4f57a7f2eb0330109d000006')
  _id=ObjectId('4f57a7f2eb0330109d00000a') slug=u'abcd'
  subject=u'Frist psot!'>]
```

In [31]:
```
f.threads[1].posts
```

Out[31]:
```
I[<Post fullslug=u'cdef' parent_id=None
  timestamp=datetime.datetime(2012, 3, 7, 18, 24, 50,
  685000) forum_id=ObjectId('4f57a7f2eb0330109d000003')
  thread_id=ObjectId('4f57a7f2eb0330109d00000c')
  _id=ObjectId('4f57a7f2eb0330109d00000d') slug=u'cdef'
  subject=u'Post in another thread'>]
```

In [32]:
```
f.threads[0].forum
```

Out[32]:
```
<Forum num_threads=None name=u'My ODM Forum'
  created=datetime.datetime(2012, 3, 7, 18, 24, 50, 683000)
  num_posts=None _id=ObjectId('4f57a7f2eb0330109d000003')
  last_post=I{u'when': None, u'user': None, u'subject':
  None} description=None>
```

In [33]:
```
f.threads[0].forum is f
```

Out[33]:
```
True
```

Extensions

Although you can get a lot done with Ming, there are always places where you might want to extend it. To support this, Ming provides extensions points in both the mapper and the `ORMSession`. You might use these to:

- Ensure that an update is sent to SOLR whenever an object is modified to support full-text indexing
- Automatically maintain a `last_updated` field that gets timestamped each time an object is saved

To implement a `SessionExtension`, just create a subclass of `ming.odm.SessionExtension` and pass an your class to the session constructor:

In [34]:
```
from ming.odm import SessionExtension
class MySessionExtension(SessionExtension):

    def before_insert(self, obj, st): pass
    def after_insert(self, obj, st): pass
    def before_update(self, obj, st): pass
    def after_update(self, obj, st): pass
    def before_delete(self, obj, st): pass
    def after_delete(self, obj, st): pass
    def before_remove(self, cls, *args, **kwargs): pass
    def after_remove(self, cls, *args, **kwargs): pass
    def before_flush(self, obj=None): pass
    def after_flush(self, obj=None): pass

    def cursor_created(self, cursor, action, *args, **kw): pass
    def before_cursor_next(self, cursor): pass
    def after_cursor_next(self, cursor): pass

tl_sess = ThreadLocalODMSession(
    M.sess, extensions=[ MySessionExtension ])
sess = ODMSession(
    M.sess, extensions=[ MySessionExtension ])
```

Similarly, you can extend the mapper with a `MapperExtension`:

In [35]:
```
from ming.odm import MapperExtension

class MyMapperExtension(MapperExtension):
    def before_insert(mapper, instance, state): pass
    def after_insert(mapper, instance, state): pass
    def before_update(mapper, instance, state): pass
    def after_update(mapper, instance, state): pass
    def before_delete(mapper, instance, state): pass
    def after_delete(mapper, instance, state): pass
    def before_remove(self, *args, **kwargs): pass
    def after_remove(self, *args, **kwargs): pass

tl_sess.mapper(Forum, M.Forum, extensions=[MyMapperExtension])
```

Out[35]:
```
<Mapper Forum:forum.forum>
```

Polymorphism

Like the schema validation layer, the ODM supports polymorphic loading of classes:

In [36]:
```
class Product(object):
    pass

class Shirt(Product):
    pass

class Film(Product):
    pass

sess.mapper(Product, M.Product)
sess.mapper(Shirt, M.Shirt)
sess.mapper(Film, M.Film)

Mapper.compile_all()
```

Nothing special here; the polymorphism is all inferred from the schema level. Now create some objects:

In [37]:
```
# Make sure we have no products defined in the DB
sess.impl.db.drop_collection('product')

# Insert some items
Shirt(_id='S001', name='Sailor Shirt', price=1500, size='S')
Shirt(_id='S002', name='Pirate Shirt', price=2200, size='L')
Film(_id='F001', name='The Matrix', price=1999, genre='SciFi')
Film(_id='F002', name='The Matrix Reloaded', price=199, genre='SciFi')

sess.flush()
sess.clear()

list(sess.impl.db.product.find())
```

Out[37]:
```
[{u'_id': u'S001',
  u'category': u'shirt',
  u'name': u'Sailor Shirt',
```

```
 u'price': 1500,
 u'size': u'S'},
{u'_id': u'S002',
 u'category': u'shirt',
 u'name': u'Pirate Shirt',
 u'price': 2200,
 u'size': u'L'},
{u'_id': u'F001',
 u'category': u'film',
 u'genre': u'SciFi',
 u'name': u'The Matrix',
 u'price': 1999},
{u'_id': u'F002',
 u'category': u'film',
 u'genre': u'SciFi',
 u'name': u'The Matrix Reloaded',
 u'price': 199}]
```

Here, we can see that the products were inserted into the database with the correct categories even though we didn't explicitly put those categories in the documents we created.

Now, let's see what happens when we query the `Product` class using Ming:

In [38]:
```
products = Product.query.find().all()
products
```

Out[38]:
```
[<Shirt category=u'shirt' price=1500 _id=u'S001'
  name=u'Sailor Shirt' size=u'S'>,
 <Shirt category=u'shirt' price=2200 _id=u'S002'
  name=u'Pirate Shirt' size=u'L'>,
 <Film category=u'film' genre=u'SciFi' price=1999 _id=u'F001'
  name=u'The Matrix'>,
 <Film category=u'film' genre=u'SciFi' price=199 _id=u'F002'
  name=u'The Matrix Reloaded'>]
```

... and once again, Ming instantiates the correct classes.

Exercises

I. Create a forum with a few threads. See what happens when you try to append to the `threads` property.
II. Create a product hierarchy with a couple of items. What happens when you query a `Shirt` for everything?
III. Create a session extension that prints out a message on various operations
IV. Create a mapper extension that saves the last modified date on insert or update