Ming Background

PyMongo is quite flexible, which is great for prototyping but can sometimes be problematic for large-scale development. Ming was developed to factor out some of the repetitive code used at SourceForge to manage our use of MongoDB and Python, particularly to develop a method of schema enforcement.

What does Ming do?

Ming provides several main facilities:

- schema validation and enforcement
- lazy and eager migration support
- an in-memory implementation of PyMongo for use in unit tests
- an object-document mapper providing higher-level constructs such as
  - a unit of work pattern
  - relations between documents

This lesson will teach you to install Ming and use it for schema validation and enforcement.

Installing Ming

Installing Ming

Due to some dependency issues, you will need to install `PasteDeploy` and `Paste` before installing Ming:

```
(tutorial-env) $ pip install PasteDeploy Paste Ming
```

Hopefully that proceeded without any problems, and now you should be able to import Ming from Python.

In [1]:
```
import ming
```

The Datastore and Session

In Ming, everything happens via the `DataStore` which is accessed via the `Session` object. You can think of a `DataStore` as a Ming-ified stand-in for the PyMongo `Database` and the `Session` as a way to link the `DataStore` to our individual `Collection`s.

But first off, lets' create a DataStore that connects to our local server:

In [2]:
```
from ming.datastore import DataStore
ds = DataStore('mongodb://localhost:27017', database='tutorial')
ds
```

Out[2]:
```
DataStore(Engine(master='mongodb://localhost:27017', slave=None, **{}), tutorial)
```

By the way, we can always get the underlying PyMongo object from a Ming object. In this case, the database is stored as `ds.db`:

In [3]:
```
ds.db
```

Out[3]:
```
Database(Connection('localhost', 27017), u'tutorial')
```

Now that we have the datastore, let's create a session:

In [4]:
```
sess = ming.Session(ds)
sess
```

Out[4]:
```
<ming.session.Session at 0x14acf90>
```

We can still access our underlying PyMongo database from the session:

In [5]:
```
sess.db
```

Out[5]:
```
Database(Connection('localhost', 27017), u'tutorial')
```

Defining a collection

So far, we have shown how you can create two levels of indirection between PyMongo and your application without any real benefit. Now, let's actually do something useful: create our first collection. Our schema will be based around a forum application, so we'll need forums, threads, and posts:

In [6]:
```
from datetime import datetime

Forum = ming.collection(
    'forum.forum', sess,
    ming.Field('_id', ming.schema.ObjectId),
    ming.Field('name', str),
    ming.Field('description', str),
    ming.Field('created', datetime, if_missing=datetime.utcnow),
    ming.Field('last_post', dict(
        when=datetime,
        user=str,
        subject=str)),
    ming.Field('num_threads', int),
    ming.Field('num_posts', int))

Forum.m.collection.drop() # to make the notebook re-runnable
Forum
```

Out[6]:
```
ming.metadata.Document<forum.forum>
```

What we've done here is create a Document class that represents all documents in the `forum.forum` collection. It turns out that our Forum is a subclass of Python's built-in `dict`, so we can do all the things we would normally do with a `dict`. In addition, Ming has added a 'manager' property `m` on the class (and instances of the class) that we can use for MongoDB-related tasks. For instance, we can access the underlying PyMongo collection:

In [7]:
```
Forum.m.collection
```

Out[7]:
```
Collection(Database(Connection('localhost', 27017), u'tutorial'), u'forum.forum')
```

Let's see the validation features of ming in action. First of all, lets' try inserting a basic forum. If we want to validate the created object, we should use the `.make` classmethod on the Forum:

In [8]:
```
f = Forum.make(dict(name='My Forum'))
f
```

```
Out[8]: {'_id': ObjectId('4f57a7e1eb03301090000001'),
         'created': datetime.datetime(2012, 3, 7, 18, 24, 33, 921632),
         'description': None,
         'last_post': {'subject': None, 'user': None, 'when': None},
         'name': 'My Forum',
         'num_posts': None,
         'num_threads': None}
```

Well, that was interesting -- we got a lot more than just a small `dict` out of that. What happened here is that Ming converted the incoming `dict` to match the schema we specified above in the definition of `Forum`.

Default Values

One interesting thing to note is that Ming has 'filled in' a few values for us. It turns out that every `SchemaItem` in Ming has an `if_missing` value (which *can* just be `ming.schema.Missing`). The `if_missing` value, if it is a callable object (a function-like object), will be called to yield the default value if none is provided, whereas any other value is simply substituted in. In this case, we see that the Ming has filled in some default values for us according to the (mostly default) `if_missing` parameters. In the case of `created`, Ming called the function `datetime.utcnow` to generate the default value at the time of object creation.

Invalid values

Let's try doing something illegal, like passing a string for an integer field:

```
In [9]: try:
            Forum.make(dict(num_posts='one'))
        except ming.schema.Invalid, inv:
            pass
        print inv
```

```
<class 'ming.metadata.Document<forum.forum>'>:
    num_posts:one is not a (<type 'int'>, <type 'long'>)
```

What happened here is that Ming tried to convert the `dict` it was given and found that it could not convert the `str` value in `num_posts` into an `int` or a `long`.

Saving and querying data

So all this validation is interesting (or maybe not), but how do we actually get these documents into the database? We could always use `Forum.m.collection.insert(f)`, but that seems verbose. Fortunately Ming provides several helpers on the `m` instance manager:

```
In [10]: f = Forum.make(dict(name='My Forum'))
         f.m.save()
```

Now this allows us to use the `Forum.m.find` helper to retrieve the value. The arguments to `find()` are the same as in PyMongo, with an additional (optional) argument `validate=False` which will skip validation of the documents loaded from the database.

```
In [11]: Forum.m.find()
```

```
Out[11]: <ming.base.Cursor at 0x14bb210>
```

What we get back is a Ming cursor, which wraps the underlying PyMongo cursor and provides validation of data coming *out* of the database. You can iterate through the cursor just as you would with PyMongo cursors, and Ming also provides a few helpers to retrieve values from the cursor:

In [12]:
```
# All values
Forum.m.find().all()
```

Out[12]:
```
[{'_id': ObjectId('4f57a7e1eb03301090000003'),
  'created': datetime.datetime(2012, 3, 7, 18, 24, 33, 944000),
  'description': None,
  'last_post': {u'subject': None, u'user': None, u'when': None},
  'name': u'My Forum',
  'num_posts': None,
  'num_threads': None}]
```

In [13]:
```
# First value (equivalent to .find_one)
Forum.m.find().first()
```

Out[13]:
```
{'_id': ObjectId('4f57a7e1eb03301090000003'),
 'created': datetime.datetime(2012, 3, 7, 18, 24, 33, 944000),
 'description': None,
 'last_post': {u'subject': None, u'user': None, u'when': None},
 'name': u'My Forum',
 'num_posts': None,
 'num_threads': None}
```

In [14]:
```
# Single value (throwing an exception if != 1 result)
Forum.m.find().one()
```

Out[14]:
```
{'_id': ObjectId('4f57a7e1eb03301090000003'),
 'created': datetime.datetime(2012, 3, 7, 18, 24, 33, 944000),
 'description': None,
 'last_post': {u'subject': None, u'user': None, u'when': None},
 'name': u'My Forum',
 'num_posts': None,
 'num_threads': None}
```

We also have the ability to update documents in-place using the MongoDB modifiers:

In [15]:
```
f = Forum.m.find().one()
f.num_posts = 4
f.m.save()
f
```

Out[15]:
```
{'_id': ObjectId('4f57a7e1eb03301090000003'),
 'created': datetime.datetime(2012, 3, 7, 18, 24, 33, 944000),
 'description': None,
 'last_post': {u'subject': None, u'user': None, u'when': None},
 'name': u'My Forum',
 'num_posts': 4,
 'num_threads': None}
```

In [16]:
```
Forum.m.update_partial(dict(_id=f._id), { '$inc': {'num_posts':5}})
f = Forum.m.find().one()
f
```

Out[16]:
```
{'_id': ObjectId('4f57a7e1eb03301090000003'),
 'created': datetime.datetime(2012, 3, 7, 18, 24, 33, 944000),
 'description': None,
 'last_post': {u'subject': None, u'user': None, u'when': None},
 'name': u'My Forum',
```

```
 'num_posts': 9,
 'num_threads': None}
```

Aside: Session Binding

One of the benefits of using the session as a layer of indirection between the Collection classes and the datastore is that it allows you to define your collections *before* configuring your datastore:

In [17]:
```
sess = ming.Session.by_name('forum')

Forum = ming.collection(
    'forum.forum', sess,
    ming.Field('_id', ming.schema.ObjectId),
    ming.Field('name', str),
    ming.Field('description', str),
    ming.Field('created', datetime, if_missing=datetime.utcnow),
    ming.Field('last_post', dict(
        when=datetime,
        user=str,
        subject=str)),
    ming.Field('num_threads', int),
    ming.Field('num_posts', int))

sess
```

Out[17]:
```
<ming.session.Session at 0x14b79d0>
```

What we have done here is associated the symbolic name 'forum' with our session. We can then use it later to configure the session:

In [18]:
```
sess = ming.Session.by_name('forum')
print sess
sess.bind = ming.datastore.DataStore(
    'mongodb://localhost:27017', database='tutorial')
print sess.db
```

```
<ming.session.Session object at 0x14b79d0>
Database(Connection('localhost', 27017), u'tutorial')
```

This can be useful in situations where we want a module defining our models to be imported before we perform configuration (perhaps by loading a config file). Ming also provides a way to configure a number of named sessions via the `ming.configure` api:

In [19]:
```
ming.configure(**{
    'ming.forum.master':'mongodb://localhost:27017',
    'ming.forum.database':'tutorial'})
ming.Session.by_name('forum').db
```

Out[19]:
```
Database(Connection(u'localhost', 27017), u'tutorial')
```

Now, let's finish out our schema for the forum model:

In [20]:
```
Thread = ming.collection(
    'forum.thread', sess,
    ming.Field('_id', ming.schema.ObjectId),
```

```
    ming.Field(
        'forum_id', ming.schema.ObjectId(if_missing=None),
        index=True),
    ming.Field('subject', str),
    ming.Field('last_post', dict(
        when=datetime,
        user=str,
        subject=str)),
    ming.Field('num_posts', int))

Thread
```

Out[20]: ming.metadata.Document<forum.thread>

Here, note that we have used the `index=True` optional argument on our `forum_id` property. This index will be created when we call Thread.m.ensure_indexes(), allowing us to keep all the metadata for our collections *and* our indexes in one location.

In [21]:
```
Post = ming.collection(
    'forum.post', sess,
    ming.Field('_id', ming.schema.ObjectId),
    ming.Field('subject', str),
    ming.Field('forum_id', ming.schema.ObjectId(if_missing=None)),
    ming.Field('thread_id', ming.schema.ObjectId(if_missing=None)),
    ming.Field('parent_id', ming.schema.ObjectId(if_missing=None)),
    ming.Field('timestamp', datetime, if_missing=datetime.utcnow),
    ming.Field('slug', str),
    ming.Field('fullslug', str, unique=True),
    ming.Index([('forum_id', 1), ('thread_id', 1)]),
    ming.Index('slug', unique=True))

Post
```

Out[21]: ming.metadata.Document<forum.post>

Here, we have used the alternate (verbose) syntax for specifying indexes, allowing us to use compound, unique, and/or sparse indexes. We can also specify unique=True in the `Field` declaration to create a simple (single-property, ascending) unique index.

Polymorphism

Since MongoDB doesn't enforce a schema on your collections, sometimes it's useful to keep related documents that don't have exactly the same schema in a collection. For instance, you might have a product catalog where you store some information (price, name, sku) for all products, but other information might be product category-specific (genre, size, etc.) For this case, Ming provides *polymorphic* collections. Here, we'll define a simple two-level product hierarchy to illustrate:

In [22]:
```
# Create hierarchy
Product = ming.collection(
    'product', sess,
    ming.Field('_id', str), # sku
    ming.Field('category', str, if_missing='product'),
    ming.Field('name', str),
    ming.Field('price', int), # in cents
    polymorphic_on='category',
    polymorphic_identity='base')

Shirt = ming.collection(
```

```
    Product,
    ming.Field('category', str, if_missing='shirt'),
    ming.Field('size', str),
    polymorphic_identity='shirt')

Film = ming.collection(
    Product,
    ming.Field('category', str, if_missing='film'),
    ming.Field('genre', str),
    polymorphic_identity='film')
```

There are a couple of things to note here. First is the use of `polymorphic_identity` in the `Product` definition. This is a parameter that tells Ming which field to inspect in objects coming from the database to see what type they are. The `polymorphic_identity` parameter tells Ming that any document in the `product` collection where the field `category` equals `'base'` should be loaded as a `Product`.

Likewise, in the `Shirt` definition, the `polymorphic_identity='shirt'` tells Ming that any document in `product` that has `category='shirt'` should be loaded as a `Shirt`. Also note that the signature of `ming.collection` is a bit different for `Shirt` and `Film`, substituting `Product` for the collection name and session, and omitting a number of fields. This is how to use Ming to declare that `Shirt` and `Film` are to be subclasses of `Product`, inheriting all fields not explicitly overridden.

Finally, note that `Shirt` and `Film` override the `category` field. This is not strictly necessary, but it's a nice feature that allows us to use Ming's built-in schema validation to fill in the `category` value for us. An example will illustrate this better:

In [23]:
```
# Make sure we have no products defined in the DB
sess.db.drop_collection('product')

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

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

Out[23]:
```
[{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'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` collection using Ming:

```
In [24]: products = Product.m.find().all()
         isinstance(products[0], Product)
```

```
Out[24]: True
```

```
In [25]: isinstance(products[0], Shirt)
```

```
Out[25]: True
```

```
In [26]: isinstance(products[0], Film)
```

```
Out[26]: False
```

```
In [27]: [ (p.name, type(p)) for p in products ]
```

```
Out[27]: [(u'Sailor Shirt', ming.metadata.Document<product:shirt>),
          (u'Pirate Shirt', ming.metadata.Document<product:shirt>),
          (u'The Matrix', ming.metadata.Document<product:film>),
          (u'The Matrix Reloaded', ming.metadata.Document<product:film>)]
```

Note how Ming was able to correctly deduce the type of each product, and use the appropriate validator on each one as well.

Exercises

I. Create a Python module containing the forum model.
II. Import the model module into an IPython session and list the methods available on `Forum.m` and `Forum().m`.
III. Create a couple of forums in your database
IV. Using the `forum.m.delete()` method, remove an instance of a forum.
V. Using the `Forum.m.remove()` method, remove an instance of a forum.
VI. Try to insert two posts with different slugs. What happens?
VII. Using the PyMongo interface (Post.m.collection, etc.), create an invalid document in the database. Use Ming to (attempt to) `find()` it. Now pass `validate=False` to allow the document to be loaded.
VIII. Create a product hierarchy. Using base PyMongo, create an invalid document (a shirt with an integer genre). See what happens when you try to query for it.