Ming Migrations

Many migrations can be handled mostly transparently. For instance, if you add a field to your model with an `if_missing` parameter, your application will (mostly) just work:

In [1]:
```
from datetime import datetime
import ming
from lesson_2_0 import model as M20

# Reset our database to have one forum inside it
M20.Forum.m.remove()
M20.Forum.make(dict(name='My Forum')).m.insert()

sess = M20.sess

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),
    # new field with default value
    ming.Field('some_new_field', str, if_missing='not set(yet)'))

Forum.m.find().one()
```

Out[1]:
```
{'_id': ObjectId('4f5687e8eb03307762000001'),
 'created': datetime.datetime(2012, 3, 6, 21, 55, 52, 325000),
 'description': None,
 'last_post': {u'subject': None, u'user': None, u'when': None},
 'name': u'My Forum',
 'num_posts': None,
 'num_threads': None,
 'some_new_field': 'not set(yet)'}
```

Note how `some_new_field` was initialized even though it didn't exist in the database.

There are some cases in which this approach won't work, however:

- if your schema change is *deleting* instead of *creating* a field (though you can always set the field definition to be a `ming.schema.Deprecated` type, which will remove it during validation)
- if your schema change is *restructuring* a field (suppose we wanted to combine name, description, and created into a subdocument `metadata`)
- if you wish to query on the properties you are changing, you need to write your queries to take *both* forms into account, as they may both appear in the database

Ming migrations seeks to make such schema changes easier. This lesson will describe Ming's support for lazy migrations, which support progressively migrating your documents, as well as Ming Flyway, a way to eagerly migrate your documents and maintain versioning in your database.

Lazy migration support

Ming's lazy migration support is linked to its validation support. Whenever a document fails validation, Ming looks looks for a special `version_of` property defined with the class. This flags the collection as a migrateable collection. If a collection is migrateable, Ming will attempt to validate the previous schema against the document.

If the previous schema successfully validates, Ming then looks for a migration function on the (new) schema and calls it on the document. Then Ming attempts to validate the output of the migration function. An example will illustrate this best. Suppose we wish to redefine our `Forum` collection to unify all its metadata under a `metadata` attribute. Our (new) `Forum` declaration would be as follows:

In [2]:
```
def migrate_forum(doc):
    metadata = dict(
        name=doc.pop('name'),
        description=doc.pop('description'),
        created=doc.pop('created'))
    return dict(doc, metadata=metadata)

Forum = ming.collection(
    'forum.forum', sess,
    ming.Field('_id', ming.schema.ObjectId),
    ming.Field('metadata', dict(
            name=str,
            description=str,
            created=ming.schema.DateTime(if_missing=datetime.utcnow)),
        required=True),
    ming.Field('last_post', dict(
        when=datetime,
        user=str,
        subject=str)),
    ming.Field('num_threads', int),
    ming.Field('num_posts', int),
    version_of=M20.Forum,
    migrate=migrate_forum)
```

Now, if we load our (previously-defined) `Forum` from the database, we "magically" get our new schema back:

In [3]:
```
Forum.m.find().one()
```

Out[3]:
```
{'_id': ObjectId('4f5687e8eb03307762000001'),
 'last_post': {u'subject': None, u'user': None, u'when': None},
 'metadata': {'created': datetime.datetime(2012, 3, 6, 21, 55, 52, 325000),
  'description': None,
  'name': u'My Forum'},
 'num_posts': None,
 'num_threads': None}
```

The data in the database, however, is still unmodified until we save it back:

In [4]:
```
Forum.m.collection.find_one()
```

Out[4]:
```
{u'_id': ObjectId('4f5687e8eb03307762000001'),
 u'created': datetime.datetime(2012, 3, 6, 21, 55, 52, 325000),
 u'description': None,
 u'last_post': {u'subject': None, u'user': None, u'when': None},
```

```
 u'name': u'My Forum',
 u'num_posts': None,
 u'num_threads': None}
```

In [5]:
```
Forum.m.find().one().m.save()
Forum.m.collection.find_one()
```

Out[5]:
```
{u'_id': ObjectId('4f5687e8eb03307762000001'),
 u'last_post': {u'subject': None, u'user': None, u'when': None},
 u'metadata': {u'created': datetime.datetime(2012, 3, 6, 21, 55, 52, 325000),
  u'description': None,
  u'name': u'My Forum'},
 u'num_posts': None,
 u'num_threads': None}
```

Ming also supports a special schema validator, `schema.Value`, which fails if the value of a field is different from what is expected. If you are doing lots of lazy migrations, it might be a good idea to use this to store a schema version. For instance, we might define a `schema_version` property that gets incremented with each version. This is handy when we want to make a minor change that would otherwise not fail validation, but it's a good practice regardless:

In [6]:
```
# Old Forum definition
OldForum = 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),
    ming.Field('schema_version', ming.schema.Value(1, if_missing=1)))

# Clear the database and put an 'old' forum document in
OldForum.m.remove()
OldForum.make(dict(name='My Forum')).m.insert()

# Redefine our migrate function to take the new version into
#  account
def migrate_forum(doc):
    metadata = dict(
        name=doc.pop('name'),
        description=doc.pop('description'),
        created=doc.pop('created'))
    return dict(doc, metadata=metadata, schema_version=2)

Forum = ming.collection(
    'forum.forum', sess,
    ming.Field('_id', ming.schema.ObjectId),
    ming.Field('metadata', dict(
            name=str,
            description=str,
            created=ming.schema.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),
    ming.Field('schema_version', ming.schema.Value(2, if_missing=2)),
    version_of=OldForum,
    migrate=migrate_forum)

Forum.m.find().one()
```

Out[6]:
```
{'_id': ObjectId('4f5687e8eb03307762000009'),
 'last_post': {u'subject': None, u'user': None, u'when': None},
 'metadata': {'created': datetime.datetime(2012, 3, 6, 21, 55, 52, 752000),
  'description': None,
  'name': u'My Forum'},
 'num_posts': None,
 'num_threads': None,
 'schema_version': 2}
```

In using lazy migrations, our model file can often become quite cluttered, so it's a good idea to split out old model versions into their own Python module and import them as needed.

Eager migration support

While lazy migrations are a nice feature, sometimes they are insufficient. In particular, if you need to query on a new field, your lazy migrations will not magically make those queries work until your application has touched *every object* in the collection. For these cases, Ming provides eager migrations via the `flyway` module.

`flyway` works with the entry points feature from setuptools/distribute. To specify a flyway migration, you must add a named entry point to your module's `setup.py` with a reference to your migration definition. Here is a sample (mostly empty setup.py) file that we'll use to create a migration:

```
from setuptools import setup

setup(
    name='FlywayTutorial',
    packages=['flyway_tutorial'],
    entry_points='''
    [flyway.migrations]
    a = flyway_tutorial.migrations_a
    b = flyway_tutorial.migrations_b''',
    paster_plugins=['Ming']
    )
```

Next, you'll need to create a 'flyway_tutorial' package in the same directory as your new setup.py file with two modules, 'migrations_a' and 'migrations_b'. They can be empty files for now.

To activate the entry points, you need to run (in the same directory as setup.py):

```
(tutorial) $ pip install -e ./
```

Once this is done, you should be able to see some help on the `paster flyway` command:

```
(tutorial) $ paster flyway -h
...
```

To actually begin using migrations, we need to create some. Let's go ahead and reset our database to contain an 'old' Forum:

In [7]:
```
from lesson_2_0 import model as M20
```

```
M20.Forum.m.remove()
M20.Forum.make(dict(name='My Forum')).m.save()
```

Now let's define a 2-way migration that will move the metadata into its own property. Put the following code in `flyway_tutorial/migrations_a.py`:

```
from flyway import Migration

class Version0(Migration):
    version = 0

    def up(self):
        collection = self.session.db['forum.forum']
        for doc in collection.find():
            doc['metadata'] = dict(
                name=doc.pop('name'),
                description=doc.pop('description'),
                created=doc.pop('created'))
            collection.save(doc)

    def down(self):
        collection = self.session.db['forum.forum']
        for doc in collection.find():
            metadata = doc.pop('metadata')
            doc.update(
                name=metadata['name'],
                description=metadata['description'],
                created=metadata['created'])
            collection.save(doc)
```

Here, we provide a *reversible migration*. This is particularly handy when migrations go wrong. Now, let's run our migration. In the same directory as your `setup.py` file, run the following command:

```
(tutorial) $ paster flyway --database tutorial
```

Now, let's see what's in the collection:

```
In [8]: M20.sess.db['forum.forum'].find_one()

Out[8]: {u'_id': ObjectId('4f5687e8eb0330776200000b'),
         u'created': datetime.datetime(2012, 3, 6, 21, 55, 52, 768000),
         u'description': None,
         u'last_post': {u'subject': None, u'user': None, u'when': None},
         u'name': u'My Forum',
         u'num_posts': None,
         u'num_threads': None}
```

... and there's our new document schema.

Now let's try a down-migration (the 'unversioned version', the one before 0, is version -1 in flyway):

```
(tutorial) $ paster flyway --database tutorial a=-1
```

And checking our collection...

```
In [9]: M20.sess.db['forum.forum'].find_one()

Out[9]: {u'_id': ObjectId('4f5687e8eb0330776200000b'),
```

```
 u'created': datetime.datetime(2012, 3, 6, 21, 55, 52, 768000),
 u'description': None,
 u'last_post': {u'subject': None, u'user': None, u'when': None},
 u'name': u'My Forum',
 u'num_posts': None,
 u'num_threads': None}
```

our old schema is back.

Flyway stores the information about what versions of our various schemas exist in the database in the `_flyway_migration_info` collection. You can either `find()` it directly:

```
In [10]: list(M20.sess.db._flyway_migration_info.find())

Out[10]: [{u'_id': ObjectId('4f55437deb03305d87000001'),
           u'versions': {u'a': -1, u'b': -1}}]
```

Or you can use the `flyway --status` command:

```
(tutorial) $ paster flyway --database tutorial --status
```

Interdependent migrations

Flyway also supports a form of inter-schema dependencies via the `up_requires` and `down_requires` methods of migrations. This can be useful in a system with a base schema and several dependent 'plugin' schemas. To see this in action, create a migration in the `flyway_tutorial/migrations_b.py` with the following content:

```
from flyway import Migration

class Version0(Migration):
    version=0
    def up(self):
        pass
    def down(self):
        pass
    def up_requires(self):
        yield ('a', self.version)
        for req in Migration.up_requires(self):
            yield req
    def down_requires(self):
        yield ('a', self.version)
        for req in Migration.down_requires(self):
            yield req
```

Now, let's see what happens when we try to migrate (only) the 'b' schema:

```
(tutorial) $ paster flyway --database tutorial b=0
```

Flyway correctly upgraded the 'a' schema before upgrading the 'b' schema.

Exercises

I. Create a new version of the `Forum` model which factors the statistics (`num_posts` and `num_threads`) into their own subdocument. Verify that the new schema migrates forward the old schema lazily.
II. Create a migration (version 1) for the new `Forum`. Verify that flyway correctly migrates the data in the database to the new version.

III. Investigate what happens when you manually specify (b=0) on the flyway command line.