Updates

PyMongo can update documents in a number of different ways. We'll start this lesson by inserting a simple example document.

In [1]:
```
import pymongo
conn = pymongo.Connection()
db = conn.tutorial
db.things.insert({'_id': 123, 'foo': 'bar'})
db.things.find_one({'_id': 123})
```

Out[1]:
```
{u'_id': 123, u'foo': u'bar'}
```

Now we can use the update method to modify the document. This operation finds the document we just inserted by _id and replaces it with a new document.

In [2]:
```
db.things.update({'_id': 123}, {'hello': 'world'})
db.things.find_one({'_id': 123})
```

Out[2]:
```
{u'_id': 123, u'hello': u'world'}
```

If instead we want to modify specific fields of the document we can use MongoDB's update operators. These include `$set`, `$inc`, `$push`, `$pull` and many more.

Here's an example using `$set` to change the value of the 'hello' field in the same document.

In [3]:
```
db.things.update({'_id': 123}, {'$set': {'hello': 'PyCon'}})
db.things.find_one({'_id': 123})
```

Out[3]:
```
{u'_id': 123, u'hello': u'PyCon'}
```

In the previous examples the server has handled modifying the document for us. We could also retrieve the document using find_one(), modify it client side and save it back again using the save() method.

In [4]:
```
from datetime import datetime
doc = db.things.find_one({'_id': 123})
doc['ts'] = datetime.utcnow()
db.things.save(doc)
db.things.find_one({'_id': 123})
```

Out[4]:
```
{u'_id': 123,
 u'hello': u'PyCon',
 u'ts': datetime.datetime(2012, 3, 7, 3, 7, 56, 916000)}
```

In [5]:
```
db.things.save({'name': 'Bernie'})
for doc in db.things.find(): print doc
```

```
{u'_id': 123, u'hello': u'PyCon', u'ts': datetime.datetime(2012, 3, 7, 3, 7, 56,
916000)}
{u'_id': ObjectId('4f56d10dfba5224d42000000'), u'name': u'Bernie'}
```

In this next example we use the `$set` operator to add an interests field to a document. The update fails because no document matched the query for the update.

[illegible]

```
In [6]: db.things.update({'name': 'Rick'}, {'$set': {'interests': []}})
        db.things.find({'name': 'Rick'}).count()

Out[6]: 0
```

Using the upsert option tells MongoDB to insert a new document if a document matching the query does not exist.

```
In [7]: db.things.update({'name': 'Rick'}, {'$set': {'interests': []}}, upsert=True)
        db.things.find_one({'name': 'Rick'})

Out[7]: {u'_id': ObjectId('4f56d10e3d03a3c47fc86765'),
         u'interests': [],
         u'name': u'Rick'}
```

Now that we've upserted that new document lets make a few modifications...

```
In [8]: db.things.update({'name': 'Rick'}, {'$push': {'interests': 'programming'}})
        db.things.find_one({'name': 'Rick'})

Out[8]: {u'_id': ObjectId('4f56d10e3d03a3c47fc86765'),
         u'interests': [u'programming'],
         u'name': u'Rick'}
```

```
In [9]: db.things.update({'name': 'Rick'}, {'$pull': {'interests': 'programming'}})
        db.things.find_one({'name': 'Rick'})

Out[9]: {u'_id': ObjectId('4f56d10e3d03a3c47fc86765'),
         u'interests': [],
         u'name': u'Rick'}
```

```
In [10]: db.things.update({}, {'$set': {'city': 'Santa Clara'}})
         for doc in db.things.find(): print doc

         {u'_id': ObjectId('4f56d10dfba5224d42000000'), u'name': u'Bernie'}
         {u'interests': [], u'_id': ObjectId('4f56d10e3d03a3c47fc86765'), u'name': u'Rick'}
         {u'city': u'Santa Clara', u'_id': 123, u'hello': u'PyCon', u'ts':
         datetime.datetime(2012, 3, 7, 3, 7, 56, 916000)}
```

Notice in that last example only one document had the 'city' field set. By default MongoDB only modifies the first document that matches the query. If you want to modify all documents that match the query add multi=True.

```
In [11]: db.things.update({}, {'$set': {'city': 'Santa Clara'}}, multi=True)
         for doc in db.things.find(): print doc

         {u'city': u'Santa Clara', u'_id': ObjectId('4f56d10dfba5224d42000000'), u'name':
         u'Bernie'}
         {u'interests': [], u'city': u'Santa Clara', u'_id':
         ObjectId('4f56d10e3d03a3c47fc86765'), u'name': u'Rick'}
         {u'city': u'Santa Clara', u'_id': 123, u'hello': u'PyCon', u'ts':
         datetime.datetime(2012, 3, 7, 3, 7, 56, 916000)}
```

Removing data

Removing documents, collections, and databases is easy in PyMongo. This first example removes specific documents from the collection. Please note that there is no multi=True option for remove. MongoDB will remove any documents that match the query.

```
In [12]: db.things.remove({'name': 'Rick'})
         for doc in db.things.find(): print doc

         {u'city': u'Santa Clara', u'_id': ObjectId('4f56d10dfba5224d42000000'), u'name':
         u'Bernie'}
         {u'city': u'Santa Clara', u'_id': 123, u'hello': u'PyCon', u'ts':
         datetime.datetime(2012, 3, 7, 3, 7, 56, 916000)}
```

If we don't specify what documents to remove MongoDB will remove them all. This just removes the documents. The collection and its indexes still exist.

```
In [13]: db.things.remove()
         db.things.count()

Out[13]: 0
```

```
In [14]: db.collection_names()

Out[14]: [u'things', u'system.indexes']
```

We can use the database object's drop_collection method to drop the collection and its indexes.

```
In [15]: db.drop_collection('things')
         db.collection_names()

Out[15]: [u'system.indexes']
```

We can use the connection object's drop_database command to drop a database.

```
In [16]: conn.database_names()

Out[16]: [u'training', u'local', u'tutorial']
```

```
In [17]: conn.drop_database('tutorial')
         conn.database_names()

Out[17]: [u'training', u'local']
```

Fire-and-forget and safe write operations.

By default all of the 10gen drivers for MongoDB (including PyMongo) do fire-and-forget write operations. PyMongo does not check for an acknowledgment from the server that a write operation was successful. This behavior can be changed using "safe" and related options.

```
In [18]: db = conn.tutorial
         db.safe_example.insert({'_id': 123})

Out[18]: 123
```

```
In [19]: db.safe_example.insert({'_id': 123})

Out[19]: 123
```

```
Out[19]: 123
```

```
In [20]: db.safe_example.count()
```

```
Out[20]: 1
```

In the above example we tried to insert two documents with the same _id field but only one was successfully written. We can add safe=True to see what happened. The _id field has an automatic unique index causing the second insert to fail. Without safe=True write operations will fail silently.

```
In [21]: try:
             db.safe_example.insert({'_id': 123}, safe=True)
         except Exception, e:
             print str(e)
```

```
E11000 duplicate key error index: tutorial.safe_example.$_id_  dup key: { : 123 }
```

Safe can also be used to check the results of an update operation. The return value indicates if existing documents were updated (useful with upsert) and how many documents were updated (useful with multi).

```
In [22]: db.safe_example.update({'_id': 123}, {'$set': {'foo': 'bar'}}, safe=True)
```

```
Out[22]: {u'connectionId': 81082,
          u'err': None,
          u'lastOp': 5716986695182712833L,
          u'n': 1,
          u'ok': 1.0,
          u'updatedExisting': True}
```

Other options can be passed to insert, save, update, and remove that imply safe=True. These include:

w (int), wtimeout (int milliseconds) - Used with replica sets to ensure the operation is replicated to 'w' servers within 'wtimeout' milliseconds.

fsync (boolean) - Force the server to immediately sync dirty memory pages to disk after the write operation.

journal (boolean) - Tells the server not to respond until the next journal group commit.

Read-your-writes consistency

If you are doing fire-and-forget write operations it is entirely possible that you could write to the server and the result of your next query won't reflect that write operation. This could happen due to the write operation being queued on a different server thread than the query. To avoid this problem PyMongo uses socket-per-thread behavior. To share sockets between multiple thread you must call end_request() when your thread no longer needs the socket.

```
In [23]: conn.end_request()
```

Exercises

I. Using the training.scores collection set a 'grade' attribute. For example, scores greater than or equal to 90 get an 'A'.
II. You're being nice so you decide to add 10 points to every score less than 60. How would you do that?
III. Remove all scores below 80.