**APPENDIX**

**PyMongo**

This tutorial is intended as an introduction to working with **MongoDB** and **PyMongo**.

**Prerequisites**

Make sure that the **PyMongo** distribution is installed. In the Python shell, the following command has to run without raising an exception:

>>> import pymongo

This tutorial also assumes that a MongoDB instance is running on the default host and port. Assuming you have downloaded and installed MongoDB, you can start using the following command:

$ mongod

**Making a Connection with MongoClient**

The first step when working with **PyMongo** is to create a MongoClient to the running **mongod** instance. This is done as follows:

>>> from pymongo import MongoClient

>>> client = MongoClient()

The above code will connect on the default host and port. We can also specify the host and port explicitly, as follows:

>>> client = MongoClient('localhost', 27017)

**Getting a Database**

A single instance of MongoDB can support multiple independent databases. When working with PyMongo you access databases using attribute style access on MongoClient instances:

>>> db = client.test\_database

If your database name is such that using attribute style access won’t work (like test-database), you can use dictionary style access instead:

>>> db = client['test-database']

**Getting a Collection**

A collection is a group of documents stored in MongoDB, and can be thought of as roughly the equivalent of a table in a relational database. Getting a collection in PyMongo works the same as getting a database:

>>> collection = db.test\_collection

or (using dictionary style access):

>>> collection = db['test-collection']

An important note about collections (and databases) in MongoDB is that they are created lazily - none of the above commands have actually performed any operations on the MongoDB server. Collections and databases are created when the first document is inserted into them.

**Documents**

Data in MongoDB is represented (and stored) using JSON-style documents. In PyMongo we use dictionaries to represent documents. As an example, the following dictionary might be used to represent a blog post:

>>> import datetime

>>> post = {"author": "Mike",

... "text": "My first blog post!",

... "tags": ["mongodb", "python", "pymongo"],

... "date": datetime.datetime.utcnow()}

Note that documents can contain native Python types (like datetime.datetime instances) which will be automatically converted to and from the appropriate BSON types.

**Inserting a Document**

To insert a document into a collection we can use the insert() method:

>>> posts = db.posts

>>> post\_id = posts.insert(post)

>>> post\_id

ObjectId('...')

When a document is inserted a special key, "\_id", is automatically added if the document doesn’t already contain an "\_id" key. The value of "\_id" must be unique across the collection. insert() returns the value of "\_id" for the inserted document. For more information, see the documentation on \_id.

After inserting the first document, the *posts* collection has actually been created on the server. We can verify this by listing all of the collections in our database:

>>> db.collection\_names()

[u'system.indexes', u'posts']

Note

The *system.indexes* collection is a special internal collection that was created automatically.

**Getting a Single Document With find\_one()**

The most basic type of query that can be performed in MongoDB is find\_one(). This method returns a single document matching a query (or None if there are no matches). It is useful when you know there is only one matching document, or are only interested in the first match. Here we use find\_one() to get the first document from the posts collection:

>>> posts.find\_one()

{u'date': datetime.datetime(...), u'text': u'My first blog post!', u'\_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'mongodb', u'python', u'pymongo']}

The result is a dictionary matching the one that we inserted previously.

Note

The returned document contains an "\_id", which was automatically added on insert.

find\_one() also supports querying on specific elements that the resulting document must match. To limit our results to a document with author “Mike” we do:

>>> posts.find\_one({"author": "Mike"})

{u'date': datetime.datetime(...), u'text': u'My first blog post!', u'\_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'mongodb', u'python', u'pymongo']}

If we try with a different author, like “Eliot”, we’ll get no result:

>>> posts.find\_one({"author": "Eliot"})

>>>

**Querying By ObjectId**

We can also find a post by its \_id, which in our example is an ObjectId:

>>> post\_id

ObjectId(...)

>>> posts.find\_one({"\_id": post\_id})

{u'date': datetime.datetime(...), u'text': u'My first blog post!', u'\_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'mongodb', u'python', u'pymongo']}

Note that an ObjectId is not the same as its string representation:

>>> post\_id\_as\_str = str(post\_id)

>>> posts.find\_one({"\_id": post\_id\_as\_str}) # No result

A common task in web applications is to get an ObjectId from the request URL and find the matching document. It’s necessary in this case to **convert the ObjectId from a string** before passing it to find\_one:

from bson.objectid import ObjectId

# The web framework gets post\_id from the URL and passes it as a string

def get(post\_id):

# Convert from string to ObjectId:

document = client.db.collection.find\_one({'\_id': ObjectId(post\_id)})

**A Note On Unicode Strings**

You probably noticed that the regular Python strings we stored earlier look different when retrieved from the server (e.g. u’Mike’ instead of ‘Mike’). A short explanation is in order.

MongoDB stores data in BSON format. BSON strings are UTF-8 encoded so PyMongo must ensure that any strings it stores contain only valid UTF-8 data. Regular strings (<type ‘str’>) are validated and stored unaltered. Unicode strings (<type ‘unicode’>) are encoded UTF-8 first. The reason our example string is represented in the Python shell as u’Mike’ instead of ‘Mike’ is that PyMongo decodes each BSON string to a Python unicode string, not a regular str.

**Bulk Inserts**

In order to make querying a little more interesting, let’s insert a few more documents. In addition to inserting a single document, we can also perform *bulk insert* operations, by passing an iterable as the first argument to insert(). This will insert each document in the iterable, sending only a single command to the server:

>>> new\_posts = [{"author": "Mike",

... "text": "Another post!",

... "tags": ["bulk", "insert"],

... "date": datetime.datetime(2009, 11, 12, 11, 14)},

... {"author": "Eliot",

... "title": "MongoDB is fun",

... "text": "and pretty easy too!",

... "date": datetime.datetime(2009, 11, 10, 10, 45)}]

>>> posts.insert(new\_posts)

[ObjectId('...'), ObjectId('...')]

There are a couple of interesting things to note about this example:

* The call to insert() now returns two ObjectId instances, one for each inserted document.
* new\_posts has a different “shape” than the other posts - there is no "tags" field and we’ve added a new field, "title". This is what we mean when we say that MongoDB is *schema-free*.

**Querying for More Than One Document**

To get more than a single document as the result of a query we use the find() method. find() returns a Cursor instance, which allows us to iterate over all matching documents. For example, we can iterate over every document in the posts collection:

>>> for post in posts.find():

... post

...

{u'date': datetime.datetime(...), u'text': u'My first blog post!', u'\_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'mongodb', u'python', u'pymongo']}

{u'date': datetime.datetime(2009, 11, 12, 11, 14), u'text': u'Another post!', u'\_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'bulk', u'insert']}

{u'date': datetime.datetime(2009, 11, 10, 10, 45), u'text': u'and pretty easy too!', u'\_id': ObjectId('...'), u'author': u'Eliot', u'title': u'MongoDB is fun'}

Just like we did with find\_one(), we can pass a document to find() to limit the returned results. Here, we get only those documents whose author is “Mike”:

>>> for post in posts.find({"author": "Mike"}):

... post

...

{u'date': datetime.datetime(...), u'text': u'My first blog post!', u'\_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'mongodb', u'python', u'pymongo']}

{u'date': datetime.datetime(2009, 11, 12, 11, 14), u'text': u'Another post!', u'\_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'bulk', u'insert']}

**Counting**

If we just want to know how many documents match a query we can perform a count() operation instead of a full query. We can get a count of all of the documents in a collection:

>>> posts.count()

3

or just of those documents that match a specific query:

>>> posts.find({"author": "Mike"}).count()

2

**Range Queries**

MongoDB supports many different types of advanced queries. As an example, lets perform a query where we limit results to posts older than a certain date, but also sort the results by author:

>>> d = datetime.datetime(2009, 11, 12, 12)

>>> for post in posts.find({"date": {"$lt": d}}).sort("author"):

... print post

...

{u'date': datetime.datetime(2009, 11, 10, 10, 45), u'text': u'and pretty easy too!', u'\_id': ObjectId('...'), u'author': u'Eliot', u'title': u'MongoDB is fun'}

{u'date': datetime.datetime(2009, 11, 12, 11, 14), u'text': u'Another post!', u'\_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'bulk', u'insert']}

Here we use the special "$lt" operator to do a range query, and also call sort() to sort the results by author.

**Indexing**

To make the above query fast we can add a compound index on "date" and "author". To start, lets use the explain() method to get some information about how the query is being performed without the index:

>>> posts.find({"date": {"$lt": d}}).sort("author").explain()["cursor"]

u'BasicCursor'

>>> posts.find({"date": {"$lt": d}}).sort("author").explain()["nscanned"]

3

We can see that the query is using the *BasicCursor* and scanning over all 3 documents in the collection. Now let’s add a compound index and look at the same information:

>>> from pymongo import ASCENDING, DESCENDING

>>> posts.create\_index([("date", DESCENDING), ("author", ASCENDING)])

u'date\_-1\_author\_1'

>>> posts.find({"date": {"$lt": d}}).sort("author").explain()["cursor"]

u'BtreeCursor date\_-1\_author\_1'

>>> posts.find({"date": {"$lt": d}}).sort("author").explain()["nscanned"]

2

Now the query is using a *BtreeCursor* (the index) and only scanning over the 2 matching documents.