Using MongoDB with Django

Adding a document-oriented database to the mix

Cesar Otero February 21, 2012

Django, a Python web framework, consists of an object-relational mapper (ORM), back-end controller, and template system. MongoDB is a document-oriented database (also known as a NoSQL database) effective for scaling and high performance. In this article, learn how to call MongoDB from Python (using MongoEngine), and integrate it into a Django project in lieu of the built-in ORM. A sample web interface for creating, reading, writing, and updating data to the MongoDB back end is included.

Django is used in such a wonderfully modular style; it's simple to replace different components of a Django-based web application. Because NoSQL databases are more common these days, you might want to try running an application with a different back end rather than one of the standard relational databases such as MySQL®. In this article, you get a light taste of MongoDB, including how to call it in your Python projects using either PyMongo or MongoEngine. Then you use Django and MongoEngine to create a simple a blog that can perform Create, Read, Update, and Delete (CRUD) operations.

About NoSQL databases

According to nosql-database.org, NoSQL databases are "next generation databases mostly addressing some of the points: being non-relational, distributed, open source, and horizontally scalable." In this class of database is MongoDB, a document-oriented database.

Out-of-the-box, Django 1.3 includes support for SQLite, MySQL, PostgreSQL, and Oracle but doesn't include support for MongoDB. However, it's easy to add support for MongoDB. Unfortunately, the drawback is that you lose the automatic admin panel. Therefore, you have to weigh this against your needs.

Brief introduction to MongoDB

MongoDB acts as a JavaScript interpreter, and hence database manipulation is done through JavaScript commands. After you install it locally on your machine (see Related topics), try some of the commands shown in Listing 1.

Listing 1. Sample JavaScript commands you can try with MongoDB

```
var x = "0";
x === 0;
typeof({});
```

You don't have to be a JavaScript expert to get started with MongoDB; yet, here are a few useful concepts:

- You can create objects using object literal syntax, in other words with two braces (for example, `var myCollection = {};`).
- You can create arrays with brackets (`[]`).
- Everything in JavaScript is an object except for numbers, Boolean, null, and undefined.

If you want to learn more about JavaScript's other features such as prototypal object-oriented programming (OOP), scoping rules, and its functional programming nature, see Related topics.

MongoDB is schemaless, in stark contrast to relational databases. Instead of tables, you use collections, which consist of documents. Documents are created using object literal syntax, as shown in Listing 2.

Listing 2. Document creation examples

```
var person1 = {name:"John Doe", age:25};
var person2 = {name:"Jane Doe", age:26, dept: 115};
```

Now, execute the commands shown in Listing 3 to create a new collection.

Listing 3. Creating collections

```
db.employees.save(person1);
db.employees.save(person2);
```

Because MongoDB is schemaless, `person1` and `person2` don't have to have the same column types or even the same number of columns. Also, MongoDB is dynamic in nature, so it creates employees rather than throwing an error. You can retrieve documents through the `find()` method. To get all of the documents in employees, call `find()` without any arguments, as shown in Listing 4.

Listing 4. A simple MongoDB query

```
> db.employees.find();
// returns
[
  {   "_id" : {   "$oid" : "4e363c4dcc93747e68055fa1"   },
        "name" : "John Doe",   "age" : 25   },
  {   "_id" : {   "$oid" : "4e363c53cc93747e68055fa2"   },
        "name" : "Jane Doe",   "dept" : 115,   "age" : 26   }
]
```

Note that `_id` is the equivalent of a primary key. To run specific queries, you need to pass another object with the key/value pair indicating what you're querying for, as shown in Listing 5.

Listing 5. Query by one search parameter

```
> db.employees.find({name: "John Doe"});
// returns
[
  {   "_id" : {   "$oid" : "4e363c4dcc93747e68055fa1"   },
  "name" : "John Doe",   "age" : 25   }
]
```

To query for employees with an age greater than 25, execute the command shown in Listing 6.

Listing 6. Query for employees with an age greater than 25

```
> db.employees.find({age:{'$gt':25}});
// returns
[
  {   "_id" : {   "$oid" : "4e363c53cc93747e68055fa2"   },
  "name" : "Jane Doe",   "dept" : 115,   "age" : 26   }
]
```

The `$gt` is a special operator that means greater than. Table 1 lists some other modifiers.

Table 1. Modifiers you can use with MongoDB

Modifier	Description
$gt	Greater than
$lt	Less than
$gte	Greater than or equals
$lte	Less than or equals
$in	Check for existence in an array, similar to the `'in'` SQL operator.

You can, of course, update a record by using the `update()` method. You can update the entire record, as shown in Listing 7.

Listing 7. Update an entire record

```
> db.employees.update({
    name:"John Doe",  // Document to update
    {name:"John Doe", age:27} // updated document
  });
```

Alternatively, you can update just a single value using the `$set` operator, as shown in Listing 8.

Listing 8. Update a single value in a record

```
> db.employees.update({name:"John Doe",
     { '$set': {age:27} }
  });
```

To empty a collection, call the `remove()` method without any arguments. For instance, if you want to remove John Doe from the employees collection, you could do what's shown in Listing 9.

Listing 9. Remove John Doe from the employees collection

```
> db.employees.remove({"name":"John Doe"});
> db.employees.find();
// returns
[
  {   "_id" : {   "$oid" : "4e363c53cc93747e68055fa2"   },   "name" : "Jane Doe",
      "dept" : 115,   "age" : 26   }
]
```

That's just enough to get you started. Of course you can continue to explore on the official website, which has a neat web-based interactive mongodb command prompt complete with tutorial as well as the official documents. See Related topics.

Integrating Django with MongoDB

You have a few options for accessing MongoDB from Python or Django. The first is using the Python module, PyMongo. Listing 10 is a sample PyMongo session, assuming you've installed MongoDB and already have an instance running on a port.

Listing 10. Sample PyMongo session

```
from pymongo import Connection

databaseName = "sample_database"
connection = Connection()

db = connection[databaseName]
employees = db['employees']

person1 = { "name" : "John Doe",
            "age" : 25, "dept": 101, "languages":["English","German","Japanese"] }

person2 = { "name" : "Jane Doe",
            "age" : 27, "languages":["English","Spanish","French"] }

print "clearing"
employees.remove()

print "saving"
employees.save(person1)
employees.save(person2)

print "searching"
for e in employees.find():
    print e["name"] + " " + unicode(e["languages"])
```

PyMongo allows you to run more than one database concurrently. To define a connection, simply pass in a database name to a connection instance. Python dictionaries, in this case, substitute the JavaScript object literals for creating new document definitions, and Python lists substitute JavaScript arrays. The `find` method returns a database cursor object that you can iterate over.

The similarity in syntax makes it easy to switch between the MongoDB command line and running commands with PyMongo. For example, Listing 11 shows how to run a query with PyMongo.

Listing 11. Run a query with PyMongo

```
for e in employees.find({"name":"John Doe"}):
    print e
```

Your other option for calling MongoDB from Python is MongoEngine, which should feel familiar if you've used Django's built-in ORM. MongoEngine is a document-to-object mapper, which is similar in concept to an ORM. Listing 12 shows an example session with MongoEngine.

Listing 12. Example MongoEngine session

```
from mongoengine import *

connect('employeeDB')

class Employee(Document):
    name = StringField(max_length=50)
    age = IntField(required=False)

john = Employee(name="John Doe", age=25)
john.save()

jane = Employee(name="Jane Doe", age=27)
jane.save()

for e in Employee.objects.all():
    print e["id"], e["name"], e["age"]
```

The `Employee` object inherits from `mongoengine.Document`. In this example, you use two field types: `StringField` and `IntField`. Similar to Django's ORM, to query for all the documents in the collection you call `Employee.objects.all()`. Notice that to access the unique object ID, you use `"id"` rather than `"_id"`.

A sample blog

Now you'll create a simple blog called Blongo. You'll use Python 2.7, Django 1.3, MongoDB 1.8.2, MongoEngine 0.4, and Hypertext Markup Language (HTML) 5. If you want to recreate my exact settings, I used Ubuntu Linux with FireFox. Blongo displays any blog entries entered upon page load and allows updating and deleting of any entries—in other words, all the standard CRUD operations. The Django views have three methods: `index`, `update`, and `delete`.

The cascading style sheets (CSS) definitions go in a separate static file. I won't go into details here, but feel free to explore the source code included in Download.

Assuming everything is installed and running well, create a new Django project and the necessary components as shown in Listing 13.

Listing 13. Commands for setting up the Django blog project

```
$ django-admin.py startproject blongo
$ cd blongo
$ django-admin.py startapp blogapp
$ mkdir templates
$ cd blogapp
$ mkdir static
```

New to Django 1.3 is an included contributed application for improved handling of static files. By adding a static directory to any application directory (such as blogapp in this case) and making sure that django.contrib.staticfiles is included in the installed applications, Django is able to find static files such as .css and .js files without needing any additional tweaks. Listing 14 shows the lines of the settings files that have been altered (from the default settings.py file) to get the blog application running.

Listing 14. Lines of the settings files that have been altered (from the default settings.py file)

```
# Django settings for blog project.
import os
APP_DIR = os.path.dirname( globals()['__file__'] )

DBNAME = 'blog'

TEMPLATE_DIRS = (
    os.path.join( APP_DIR, 'templates' )
)

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.blogapp',
)
```

You have three templates in this project as well: index.html, update.html, and delete.html. Listing 15 shows the code for all three template files.

Listing 15. Code for the index.html, update.html, and delete.html template files

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <link href="{{STATIC_URL}}blog.css" rel="stylesheet" type="text/css">
  </head>
  <body>
    <h1>Blongo</h1>
    <form method="post" action="http://127.0.0.1:8000/">
      {% csrf_token %}
      <ul>
        <li>
          <input type="text" name="title" placeholder="Post Title" required>
        </li>
        <li>
          <textarea name="content" placeholder="Enter Content" rows=5 cols=50 required>
          </textarea>
        </li>
        <li>
          <input type="submit" value="Add Post">
        </li>
      </ul>
    </form>
<!-- Cycle through entries -->
    {% for post in Posts %}
      <h2> {{ post.title }} </h2>
      <p>{{ post.last_update }}</p>
```

```
      <p>{{ post.content }}</p>
      <form method="get" action="http://127.0.0.1:8000/update">
        <input type="hidden" name="id" value="{{ post.id }}">
        <input type="hidden" name="title" value="{{ post.title }}">
        <input type="hidden" name="last_update" value="{{ post.last_update }}">
        <input type="hidden" name="content" value="{{ post.content }}">
        <input type="submit" name="" value="update">
      </form>
      <form method="get" action="http://127.0.0.1:8000/delete">
        <input type="hidden" name="id" value="{{post.id}}">
        <input type="submit" value="delete">
      </form>
    {% endfor %}
  </body>
</html>

<!-- update.html -->
<!DOCTYPE html>
<html>
  <head>
    <link href="{{STATIC_URL}}blog.css" rel="stylesheet" type="text/css">
  </head>
  <body>
    <h1>Blongo - Update Entry</h1>
    <form method="post" action="http://127.0.0.1:8000/update/">
      {% csrf_token %}
      <ul>
        <li><input type="hidden" name="id" value="{{post.id}}"></li>
        <li>
          <input type="text" name="title" placeholder="Post Title"
             value="{{post.title}}" required>
          <input type="text" name="last_update"
             value="{{post.last_update}}" required>
        </li>
        <li>
          <textarea name="content" placeholder="Enter Content"
            rows=5 cols=50 required>
            {{post.content}}
          </textarea>
        </li>
        <li>
          <input type="submit" value="Save Changes">
        </li>
      </ul>
    </form>
  </body>
</html>
<!-- delete.html -->
<!DOCTYPE html>
<html>
  <head>
    <link href="{{STATIC_URL}}blog.css" rel="stylesheet" type="text/css">
  </head>
  <body>
    <h1>Blongo - Delete Entry</h1>
    <form method="post" action="http://127.0.0.1:8000/delete/">
      {% csrf_token %}
      <input type="hidden" name="id" value="{{id}}">
      <p>Are you sure you want to delete this post?</p>
      <input type="submit" value="Delete">
    </form>
  </body>
</html>
```

Next, change your URL mappings to the code shown in Listing 16, which points to the views for index, update, and delete. You want the sample blog to create new blog entries (at the index),

update existing blog posts, and delete them when desired. Each action is accomplished by posting to a specific URL.

Listing 16. URL mappings for index, update, and delete

```
from django.conf.urls.defaults import patterns, include, url

urlpatterns = patterns('',
    url(r'^$', 'blog.blogapp.views.index'),
    url(r'^update/', 'blog.blogapp.views.update'),
    url(r'^delete/', 'blog.blogapp.views.delete'),
)
```

Notice that you *don't* need to run the `syncdb` Django command. To integrate MongoDB into your application, you need MongoEngine. In the models.py file of the blogapp directory, add the code shown in Listing 17.

Listing 17. Including MongoEngine in the data layer

```
from mongoengine import *
from blog.settings import DBNAME

connect(DBNAME)

class Post(Document):
    title = StringField(max_length=120, required=True)
    content = StringField(max_length=500, required=True)
    last_update = DateTimeField(required=True)
```

The database name is taken from the settings file to separate the concerns. Each blog post contains three required fields: `title`, `content`, and `last_update`. If you compare and contrast this listing with what you would normally do in Django, the difference isn't enormous. Instead of having a class that inherits from django.db.models.Model, this listing uses the `mongoengine.Document` class instead. I don't have space here to enter into the difference between the data types, but feel free to check out the MongoEngine documents (see Related topics).

Table 2 lists the MongoEngine field types and shows the equivalent Django ORM field type, if one exists.

Table 2. MongoEngine field types and Django ORM equivalents

MongoEngine field type	Django ORM equivalent
StringField	CharField
URLField	URLField
EmailField	EmailField
IntField	IntegerField
FloatField	FloatField
DecimalField	DecimalField
BooleanField	BooleanField
DateTimeField	DateTimeField

EmbeddedDocumentField	None
DictField	None
ListField	None
SortedListField	None
BinaryField	None
ObjectIdField	None
FileField	FileField

Finally, you can set up your views. Here you have three view methods: `index`, `update`, and `delete`. To execute the intended action, a post request must be made to the specific URL. For example, to update a document a post must be made to localhost:8000/update. Executing an http 'GET' request will not save, update, and so on. New blog posts are inserted from the index view. Listing 18 shows the implementations for the index, update, and delete views.

Listing 18. The Django views

```
from django.shortcuts import render_to_response
from django.template import RequestContext
from models import Post
import datetime

def index(request):
    if request.method == 'POST':
       # save new post
       title = request.POST['title']
       content = request.POST['content']

       post = Post(title=title)
       post.last_update = datetime.datetime.now()
       post.content = content
       post.save()

    # Get all posts from DB
    posts = Post.objects
    return render_to_response('index.html', {'Posts': posts},
                              context_instance=RequestContext(request))


def update(request):
    id = eval("request." + request.method + "['id']")
    post = Post.objects(id=id)[0]

    if request.method == 'POST':
        # update field values and save to mongo
        post.title = request.POST['title']
        post.last_update = datetime.datetime.now()
        post.content = request.POST['content']
        post.save()
        template = 'index.html'
        params = {'Posts': Post.objects}

    elif request.method == 'GET':
        template = 'update.html'
        params = {'post':post}

    return render_to_response(template, params, context_instance=RequestContext(request))


def delete(request):
```

```
    id = eval("request." + request.method + "['id']")

    if request.method == 'POST':
        post = Post.objects(id=id)[0]
        post.delete()
        template = 'index.html'
        params = {'Posts': Post.objects}
    elif request.method == 'GET':
        template = 'delete.html'
        params = { 'id': id }

    return render_to_response(template, params, context_instance=RequestContext(request))
```

You may have noticed the `eval` statements used to retrieve the document IDs. This is used to avoid having to write the `if` statement shown in Listing 19.

Listing 19. Alternate way of retrieving the document ID

```
    if request.method == 'POST':
        id = request.POST['id']
    elif request.method == 'GET':
        id = request.GET['id']
```

You could also write it that way. That's all it takes to get a simple blog up and running. There are obviously many components missing for a final product such as users, a login, tags, and so on.

Conclusion

As you can see, there really isn't much to calling MongoDB from Django. In this article, I introduced MongoDB briefly, and explained how to access it and manipulate its collections and documents from Python through the PyMongo wrapper and the MongoEngine object-to-document mapper. Finally, I offered a quick demonstration of how to create a basic CRUD form using Django. Although this is just a first step, hopefully you now understand how apply this setup in your own projects.

Downloadable resources

Description	Name	Size
Sample Django application with MongoEngine	blongo.zip	12KB

Related topics

- Develop and deploy your next app on the IBM Bluemix cloud platform.
- Learn more about JavaScript from Mozilla's tutorial, "A re-introduction to JavaScript," and Douglas Crockford's book, *JavaScript: The Good Parts* (O'Reilly Media/Yahoo Press, May 2008).
- Learn more about and download MongoDB.
- Download and explore Django.
- Visit the Python website for downloads and documentation.
- Check out MongoEngine.
- Dig in to PyMongo.
- Start developing with product trials, free downloads, and IBM Bluemix services.