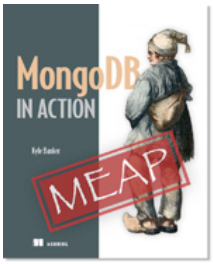
MANNING PUBLICATIONS



MongoDB in Action

By Kyle Banker

In this article, based on chapter 3 of MongoDB in Action, author Kyle Banker builds a simple application for archiving and displaying tweets and demonstrates how easy it is to consume a JSON API like Twitter's to MongoDB documents.

You may also be interested in...

Building a Mini Application

We're going to build a simple application for archiving and displaying tweets. You could imagine this as a component of a large application that allows users to keep tabs on search terms relevant to their business. Many such applications exist. This implementation will demonstrate how easy it is to consume a JSON API like Twitter's to MongoDB documents. If we were doing this with a relational database, we'd have to plan out a schema, probably consisting of multiple tables, and then perform the migrations on our database. Here, none of that is required; and yet, we'll still preserve the rich structure of the tweet documents, and we'll be able to query of them effectively.

Let's call the app TweetArchiver. TweetArchiver will consist of two components: the archiver and the viewer. The archiver will store relevant tweets by hitting the Twitter search API, and the viewer will display the results in a web browser.

Setting up

This application requires three Ruby libraries. You can install them as follows:

```
gem install mongo
gem install twitter
gem install sinatra
```

It will be useful to have a configuration file that we can share between the archiver and viewer scripts. Create a file called `config.rb`, and initialize the following constants.

```
DATABASE_NAME = "twitter-archive"
COLLECTION_NAME = "tweets"
TAGS = ["mongodb", "ruby"]
```

All we're doing here is specifying the database and collection we'll be using for our application. We're also providing an array of search terms, each of which will be sent to the Twitter API.

The next step is to write our archiver script. We start with a `TweetArchiver` class. This class will work by being instantiated with a search term. We'll then be able to call the method on the instance, update `TweetArchiver`, which will call the Twitter API, and save the results to a MongoDB collection.

Let's start with the class's constructor.

```
def initialize(tag)
  connection = Mongo::Connection.new
  db = connection[DATABASE_NAME]
  @tweets = db[COLLECTION_NAME]
  @tweets.create_index([['id', 1]], :unique => true)
  @tweets.create_index([['tags', 1], ['id', -1]])
  @tag = tag
end
```

The `initialize` method instantiates a connection, a database object, and the collection we're using to store tweets. The method also creates a couple of indexes.

Every tweet will have an `id` field (distinct from MongoDB's `_id` fields), which stores the internal Twitter id. We're creating a unique index on this field so that it will be impossible to insert the same tweet twice.

We're also creating a compound index on `tags` and `id`. Indexes can be specified in an ascending or descending order. This matters mainly when creating compound indexes; the direction of the indexes is determined by how we expect to query the data. Since we're going to want to query for a particular tag and show the results from newest to oldest, an index with tags ascending and id descending will make that query as efficient as possible. Notice that we indicate index direction with 1 for ascending and -1 for descending.

Gathering data

One of the great things about MongoDB is that we can insert data regardless of its structure. Since we don't need to know which fields we'll be given in advance, Twitter is free to modify its API return values with practically no consequences to our application. Normally, using an RDBMS, any change to Twitter's API or, more generally, to our data source, will require a database migration. With MongoDB, our code might need to change to accommodate new data schemas, but the database itself can handle any document-style schema automatically.

The Ruby Twitter library returns Ruby hashes, which we can pass directly to our MongoDB collection object. Within our `TweetArchiver`, we add the following instance method:

```
def save_tweets_for(term)
  Twitter::Search.new.containing(term).each do |tweet|
    @tweets_found ||= 0
    @tweets_found += 1
    tweet_with_tag = tweet.to_hash.merge!({"tags" => [term]})
    @tweets.save(tweet_with_tag)
  end
end
```

We do make a small modification to each tweet document before saving it. Since we want to make searching our documents easy, we add the search terms that we used to a `tags` attribute. Then we pass the modified document to the `save` method. That's it. The entire code for our archiver class is displayed in listing 1.

Listing 1 A class for fetching tweets and archiving them in MongoDB

```
require 'rubygems'
require 'mongo'
require 'twitter'
require 'config'

class TweetArchiver
  # Create a new instance of TweetArchiver
  def initialize(tag)
    connection = Mongo::Connection.new
    db = connection[DATABASE_NAME]
    @tweets = db[COLLECTION_NAME]

    @tweets.create_index([['id', 1]], :unique => true)
    @tweets.create_index([['tags', 1], ['id', -1]])
    @tag = tag
  end

  def update
    save_tweets_for(@tag)
  end

  private

  def save_tweets_for(term)
    Twitter::Search.new(term).each do |tweet|
      tweet_with_tag = tweet.to_hash.merge!({"tags" => [term]})
      @tweets.save(tweet_with_tag)
    end
  end
end
```

All that remains is to write a small script to run the `TweetArchiver` code against each of our search terms. Create a file called `update.rb` and add the following:

```
require 'config'
require 'archiver'

TAGS.each do |tag|
  archive = TweetArchiver.new(tag)
  archive.update
end
```

Next, run the update script.

```
ruby update.rb
```

You'll see some status messages indicating that tweets have been found and saved, but you can verify this by opening up the MongoDB shell and querying the collection.

```
MongoDB shell version: 1.6
url: test
connecting to: test
type "help" for help
> use twitter-archive
switched to db twitter-archive
> db.tweets.count()
30
```

To keep the archive current, the update script could be run once every few minutes via a cron job. But that's just an administrative detail. The exciting part is that we've accomplished the basic task of storing tweets from Twitter searches with some relatively simple code.[1] Now comes the task of displaying the results.

Viewing the archive

We're going to be using Ruby's Sinatra web framework to build a simple app to display the results. Create a file called `viewer.rb` and place it in the same directory used in the previous section. Next, create a subdirectory called `views` and place a file there called `tweets.erb`. The project's file structure should look like this:

```
- config.rb
- archiver.rb
- update.rb
- viewer.rb
```

[1] Of course, all of this could have been accomplished in many fewer lines of code and without a class declaration, but we're trying to write respectable code here.

```
- /views
  - tweets.erb
```

Now edit `viewer.rb` with the code in listing 2.

Listing 2 A simple Sinatra application for displaying and searching the Tweet Archive

```
require 'rubygems'
require 'mongo'
require 'sinatra'

require 'config'

configure do
  db = Mongo::Connection.new[DATABASE_NAME]
  TWEETS = db[COLLECTION_NAME]
end

get '/' do
  if params['tag']
    selector = {:tags => params['tag']}
  else
    selector = {}
  end

  @tweets = TWEETS.find(selector).sort(["id", -1])

  erb :tweets
end
```

The first lines require the necessary libraries along with our config file. Next, we have a configuration block that creates a connection to MongoDB and stores a reference to our `tweets` collection in the constant `TWEETS`.

The real meat of the application is in the lines beginning with `get '/' do`. The code in this block handles requests to our application's root URL. First, we build our query selector. If a tags URL parameter has been provided, then we create a query selector that restricts the result set to the given tags. Otherwise, we create a blank selector, which will return all documents in the collection. We then issue the query. By now, you should know that what gets assigned to the `@tweets` variable isn't a result set, but a cursor. We'll iterate over that cursor in our view. The last line renders the view file `tweets.erb`, whose full code listing can be seen in listing 3.

Listing 3 HTML with embedded Ruby for rendering the Tweets

```
-->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html lang='en' xml:lang='en' xmlns='http://www.w3.org/1999/xhtml'>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

  <style>
    body {
      background-color: #DBD4C2;
      width: 1000px;
      margin: 50px auto;
    }

    h2 {
      margin-top: 2em;
    }
  </style>
</head>

<body>

<h1>Tweet Archive</h1>

<% TAGS.each do |tag| %>
  <a href="/?tag=<%= tag %>"><%= tag %></a>
<% end %>

<% @tweets.each do |tweet| %>
  <h2><%= tweet['text'] %></h2>
  <p>
    <a href="http://twitter.com/<%= tweet['from_user'] %>">
      <%= tweet['from_user'] %>
    </a>
    on <%= tweet['created_at'] %>
  </p>

  <img src="<%= tweet['profile_image_url'] %>" width="48" />
<% end %>

</body>
</html>
<!-
```

Most of the code is just HTML, with some ERB mixed in.[2] The important parts come near the end, with the two iterators. The first of these cycles through our list of tags to display links for restricting the result set to a given tag. The second iterator, beginning with the `@tweets.each` code, cycles through each tweet to display the tweet's text, creation date, and user profile image. The results of this can be seen by running the application:

[2] ERB stands for "embedded Ruby." The Sinatra app runs our tweets.erb file through an ERB processor and evaluates any code between <% and %> as Ruby in the context of our application.

```
ruby viewer.rb
```

If the application starts without error, you'll see the standard Sinatra startup message:

```
arete:tweets kyle$ ruby viewer.rb
== Sinatra/1.0.0 has taken the stage on 4567 for development
with backup from Mongrel
```

You can then point your web browser to http://localhost:4567. The page should look something like the screenshot in figure 1. Try clicking on the links at the top of the screen to narrow the results to a particular tag.

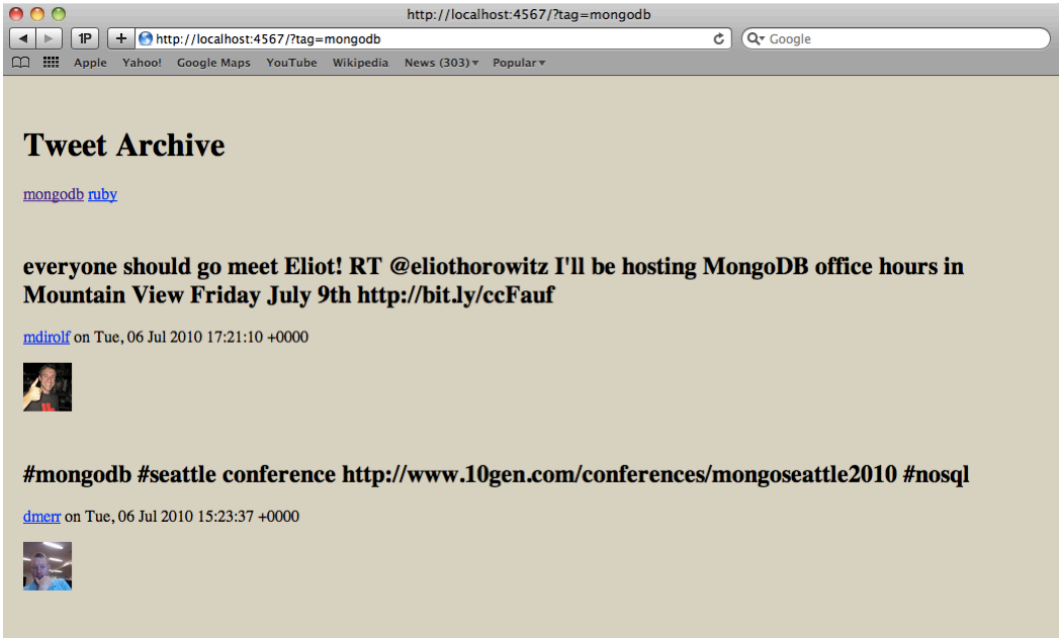


Figure 1 The Tweet Archiver output rendered in a web browser

And that's our application.

Summary

Our application, admittedly, is quite simple. But, it demonstrates the ease of using MongoDB. We didn't have to define our schema in advance; we took advantage of secondary indexes to make our queries fast and prevent duplicate inserts; and we had a relatively simple integration with our programming language.

Here are some other Manning titles you might be interested in:

SQL Server MVP Deep Dives

Paul Nielsen, Kalen Delaney, Greg Low, Adam Machanic, Paul S. Randal, and Kimberly L. Tripp

SQL Server DMVs in Action

Ian W. Stirk

SQL Server 2008 Administration in Action

Rod Colledge

Last updated: July 13, 2011