MongoDB Use Cases

Release 2.2.2

MongoDB Documentation Project

December 20, 2012

Contents

1 Introduction **ii**
1.1 About MongoDB Documentation . . . iii
About MongoDB . . . iii
About the Documentation Project . . . iii

2 Operational Intelligence **iv**
2.1 Storing Log Data . . . v
Overview . . . v
Operations . . . vii
Sharding . . . xii
Managing Event Data Growth . . . xiv
2.2 Pre-Aggregated Reports . . . xv
Overview . . . xv
Schema . . . xv
Operations . . . xix
Sharding . . . xxii
2.3 Hierarchical Aggregation . . . xxiii
Overview . . . xxiii
Schema . . . xxiv
Operations . . . xxv
Sharding . . . xxix

3 Product Data Management **xxx**
3.1 Product Catalog . . . xxx
Overview . . . xxx
Operations . . . xxxiv
Scaling . . . xxxvi
3.2 Inventory Management . . . xxxvii
Overview . . . xxxvii
Operations . . . xxxviii
Sharding . . . xliii
3.3 Category Hierarchy . . . xliv
Overview . . . xliv
Operations . . . xlv
Sharding . . . xlviii

4 Content Management Systems **l**
4.1 Metadata and Asset Management l
Overview l
Operations lii
Sharding lv
4.2 Storing Comments lvi
Overview lvi
One Document per Comment lvii
Embedding All Comments lx
Hybrid Schema Design lxii
Sharding lxv

5 Python Application Development **lxv**
5.1 Write a Tumblelog Application with Django MongoDB Engine lxv
Introduction lxv
Installation lxvi
Build a Blog to Get Started lxvi
Add Comments to the Blog lxx
Add Site Administration Interface lxxiii
Convert the Blog to a Tumblelog lxxv
5.2 Write a Tumblelog Application with Flask and MongoEngine lxxvii
Introduction lxxvii
Installation lxxviii
Build a Blog to Get Started lxxix
Add Comments to the Blog lxxxiv
Add a Site Administration Interface lxxxvi
Converting the Blog to a Tumblelog xc
Additional Resources xciv

1 Introduction

The use case documents provide introductions to the patterns, design, and operation used in application development with MongoDB. Each document provides more concrete examples and implementation details to support core MongoDB use cases. These documents highlight application design, and data modeling strategies (*i.e. schema design*) for MongoDB with special attention to pragmatic considerations including indexing, performance, sharding, and scaling. Each document is distinct and can stand alone; however, each section builds on a set of common topics.

The *operational intelligence* case studies describe applications that collect machine generated data from logging systems, application output, and other systems. The *product data management* case studies address aspects of applications required for building product catalogs, and managing inventory in e-commerce systems. The *content management* case studies introduce basic patterns and techniques for building content management systems using MongoDB.

Finally, the *introductory application development tutorials with Python and MongoDB*, provides a complete and fully developed application that you can build using MongoDB and popular Python web development tool kits.

1.1 About MongoDB Documentation

About MongoDB

MongoDB is a *document*-oriented database management system designed for performance, horizontal scalability, high availability, and advanced queryability. See the following wiki pages for more information about MongoDB:

- Introduction
- Philosophy
- About

If you want to download MongoDB, see the downloads page.

If you'd like to learn how to use MongoDB with your programming language of choice, see the introduction to the `drivers`.

About the Documentation Project

This Manual

The MongoDB documentation project provides a complete manual for the MongoDB database. This resource is replacing eventually replace MongoDB's original documentation.

Editions In addition to the <http://docs.mongodb.org/manual/> site, you can also access this content in the following editions provided for your convenience:

- ePub Format
- Single HTML Page
- PDF Format

PDF files that provide access to subsets of the MongoDB Manual:

- MongoDB Reference Manual
- MongoDB Use Case Guide
- MongoDB CRUD Operation Introduction

For Emacs users Info/Texinfo users, the following experimental Texinfo manuals are available for offline use:

- MongoDB Manual Texinfo (tar.gz)
- MongoDB Reference Manual (tar.gz)
- MongoDB CURD Operation Introduction (tar.gz)

Important: The `texinfo` manuals are experimental. If you find an issue with one of these editions, please file an issue in the DOCS Jira project.

Version and Revisions This version of the manual reflects version 2.2.2 of MongoDB.

See the MongoDB Documentation Project Page for an overview of all editions and output formats of the MongoDB Manual. You can see the full revision history and track ongoing improvements and additions for all versions of the manual from its GitHub repository.

This edition reflects "`master`" branch of the documentation as of the "`312d6123917eff61bb399f1410463631 4934389d`" revision. This branch is explicitly accessible via "http://docs.mongodb.org/master" and you can always reference the commit of the current manual in the release.txt file.

The most up-to-date, current, and stable version of the manual is always available at "http://docs.mongodb.org/manual/."

Contributing to the Documentation

The entire source of the documentation is available in the docs repository along with all of the other MongoDB project repositories on GitHub. You can clone the repository by issuing the following command at your system shell:

```
git clone git://github.com/mongodb/docs.git
```

If you have a GitHub account and want to fork this repository, you may issue pull requests, and someone on the documentation team will merge in your contributions promptly. In order to accept your changes to the Manual, you have to complete the MongoDB/10gen Contributor Agreement.

This project tracks issues at MongoDB's DOCS project. If you see a problem with the documentation, please report it there.

Writing Documentation

The MongoDB Manual uses Sphinx, a sophisticated documentation engine built upon Python Docutils. The original reStructured Text files, as well as all necessary Sphinx extensions and build tools, are available in the same repository as the documentation.

You can view the documentation style guide and the build instructions in reStructured Text files in the top-level of the documentation repository. If you have any questions, please feel free to open a Jira Case.

2 Operational Intelligence

As an introduction to the use of MongoDB for operational intelligence and real time analytics use, "*Storing Log Data* (page v)" document describes several ways and approaches to modeling and storing machine generated data with MongoDB. Then, "*Pre-Aggregated Reports* (page xv)" describes methods and strategies for processing data to generate aggregated reports from raw event-data. Finally "*Hierarchical Aggregation* (page xxiii)" presents a method for using MongoDB to process and store hierarchical reports (i.e. per-minute, per-hour, and per-day) from raw event data.

2.1 Storing Log Data

Overview

This document outlines the basic patterns and principles for using MongoDB as a persistent storage engine for log data from servers and other machine data.

Problem

Servers generate a large number of events (i.e. logging,) that contain useful information about their operation including errors, warnings, and users behavior. By default, most servers, store these data in plain text log files on their local file systems.

While plain-text logs are accessible and human-readable, they are difficult to use, reference, and analyze without holistic systems for aggregating and storing these data.

Solution

The solution described below assumes that each server generates events also consumes event data and that each server can access the MongoDB instance. Furthermore, this design assumes that the query rate for this logging data is substantially lower than common for logging applications with a high-bandwidth event stream.

Note: This case assumes that you're using an standard uncapped collection for this event data, unless otherwise noted. See the section on *capped collections* (page xiv)

Schema Design

The schema for storing log data in MongoDB depends on the format of the event data that you're storing. For a simple example, consider standard request logs in the combined format from the Apache HTTP Server. A line from these logs may resemble the following:

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326 "[http://www.e
```

The simplest approach to storing the log data would be putting the exact text of the log record into a document:

```
{
  _id: ObjectId('4f442120eb03305789000000'),
 line: '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326 "[http
}
```

While this solution is does capture all data in a format that MongoDB can use, the data is not particularly useful, or it's not terribly efficient: if you need to find events that the same page, you would need to use a regular expression query, which would require a full scan of the collection. The preferred approach is to extract the relevant information from the log data into individual fields in a MongoDB *document*.

When you extract data from the log into fields, pay attention to the data types you use to render the log data into MongoDB.

As you design this schema, be mindful that the data types you use to encode the data can have a significant impact on the performance and capability of the logging system. Consider the date field: In the above example, `[10/Oct/2000:13:55:36 -0700]` is 28 bytes long. If you store this with the UTC timestamp type, you can convey the same information in only 8 bytes.

Additionally, using proper types for your data also increases query flexibility: if you store date as a timestamp you can make date range queries, whereas it's very difficult to compare two *strings* that represent dates. The same issue holds for numeric fields; storing numbers as strings requires more space and is difficult to query.

Consider the following document that captures all data from the above log entry:

```
{
     _id: ObjectId('4f442120eb03305789000000'),
     host: "127.0.0.1",
     logname: null,
     user: 'frank',
     time: ISODate("2000-10-10T20:55:36Z"),
     path: "/apache_pb.gif",
     request: "GET /apache_pb.gif HTTP/1.0",
     status: 200,
     response_size: 2326,
     referrer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
     user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
}
```

When extracting data from logs and designing a schema, also consider what information you can omit from your log tracking system. In most cases there's no need to track *all* data from an event log, and you can omit other fields. To continue the above example, here the most crucial information may be the host, time, path, user agent, and referrer, as in the following example document:

```
{
     _id: ObjectId('4f442120eb03305789000000'),
     host: "127.0.0.1",
     time:  ISODate("2000-10-10T20:55:36Z"),
     path: "/apache_pb.gif",
     referer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
     user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
}
```

You may also consider omitting explicit time fields, because the `ObjectId` embeds creation time:

```
{
     _id: ObjectId('4f442120eb03305789000000'),
     host: "127.0.0.1",
     path: "/apache_pb.gif",
     referer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
     user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
}
```

System Architecture

The primary performance concern for event logging systems are:

1. how many inserts per second can it support, which limits the event throughput, and
2. how will the system manage the growth of event data, particularly concerning a growth in insert activity.

   In most cases the best way to increase the capacity of the system is to use an architecture with some sort of *partitioning* or *sharding* that distributes writes among a cluster of systems.

Operations

Insertion speed is the primary performance concern for an event logging system. At the same time, the system must be able to support flexible queries so that you can return data from the system efficiently. This section describes procedures for both document insertion and basic analytics queries.

The examples that follow use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Inserting a Log Record

Write Concern MongoDB has a configurable *write concern*. This capability allows you to balance the importance of guaranteeing that all writes are fully recorded in the database with the speed of the insert.

For example, if you issue writes to MongoDB and do not require that the database issue any response, the write operations will return *very* fast (i.e. asynchronously,) but you cannot be certain that all writes succeeded. Conversely, if you require that MongoDB acknowledge every write operation, the database will not return as quickly but you can be certain that every item will be present in the database.

The proper write concern is often an application specific decision, and depends on the reporting requirements and uses of your analytics application.

Insert Performance The following example contains the setup for a Python console session using PyMongo, with an event from the Apache Log:

```
>>> import bson
>>> import pymongo
>>> from datetime import datetime
>>> conn = pymongo.Connection()
>>> db = conn.event_db
>>> event = {
...     _id: bson.ObjectId(),
...     host: "127.0.0.1",
...     time:  datetime(2000,10,10,20,55,36),
...     path: "/apache_pb.gif",
...     referer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
...     user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
...}
```

The following command will insert the `event` object into the `events` collection.

```
>>> db.events.insert(event, w=0)
```

By setting `w=0`, you do not require that MongoDB acknowledges receipt of the insert. Although very fast, this is risky because the application cannot detect network and server failures. See *write-concern* for more information.

If you want to ensure that MongoDB acknowledges inserts, you can pass `w=1` argument as follows:

```
>>> db.events.insert(event, w=1)
```

MongoDB also supports a more stringent level of write concern, if you have a lower tolerance for data loss:

You can ensure that MongoDB not only *acknowledge* receipt of the message but also commit the write operation to the on-disk journal before returning successfully to the application, use can use the following `insert()` operation:

```
>>> db.events.insert(event, j=True)
```

Note: `j=True` implies `w=1`.

Finally, if you have *extremely low* tolerance for event data loss, you can require that MongoDB replicate the data to multiple *secondary replica set* members before returning:

```
>>> db.events.insert(event, w=majority)
```

This will force your application to acknowledge that the data has replicated to a majority of configured members of the *replica set*. You can combine options as well:

```
>>> db.events.insert(event, j=True, w=majority)
```

In this case, your application will wait for a successful journal commit on the *primary and* a replication acknowledgment from a majority of configured secondaries. This is the safest option presented in this section, but it is the slowest. There is always a trade-off between safety and speed.

Note: If possible, consider using bulk inserts to insert event data.

All write concern options apply to bulk inserts, but you can pass multiple events to the `insert()` method at once. Batch inserts allow MongoDB to distribute the performance penalty incurred by more stringent write concern across a group of inserts.

See Also:

"*Write Concern for Replica Sets*" and `getLastError`.

Finding All Events for a Particular Page

The value in maintaining a collection of event data derives from being able to query that data to answer specific questions. You may have a number of simple queries that you may use to analyze these data.

As an example, you may want to return all of the events associated with specific value of a field. Extending the Apache access log example from above, a common case would be to query for all events with a specific value in the `path` field: This section contains a pattern for returning data and optimizing this operation.

Query Use a query that resembles the following to return all documents with the `http://docs.mongodb.org/manual/apache_pb.gif` value in the `path` field:

```
>>> q_events = db.events.find({'path': '/apache_pb.gif'})
```

Note: If you choose to *shard* the collection that stores this data, the *shard key* you choose can impact the performance of this query. See the *sharding* (page xii) section of the sharding document.

Index Support Adding an index on the `path` field would significantly enhance the performance of this operation.

```
>>> db.events.ensure_index('path')
```

Because the values of the `path` likely have a random distribution, in order to operate efficiently, the entire index should be resident in RAM. In this case, the number of distinct paths is typically small in relation to the number of documents, which will limit the space that the index requires.

If your system has a limited amount of RAM, or your data set has a wider distribution in values, you may need to re investigate your indexing support. In most cases, however, this index is entirely sufficient.

See Also:

The `db.collection.ensureIndex()` JavaScript method and the `db.events.ensure_index()` method in PyMongo.

Finding All the Events for a Particular Date

The next example describes the process for returning all the events for a particular date.

Query To retrieve this data, use the following query:

```
>>> q_events = db.events.find('time':
...     { '$gte':datetime(2000,10,10),'$lt':datetime(2000,10,11)})
```

Index Support In this case, an index on the `time` field would optimize performance:

```
>>> db.events.ensure_index('time')
```

Because your application is inserting events in order, the parts of the index that capture recent events will always be in active RAM. As a result, if you query primarily on recent data, MongoDB will be able to maintain a large index, quickly fulfill queries, and avoid using much system memory.

See Also:

The `db.events.ensureIndex()` JavaScript method and the `db.events.ensure_index()` method in PyMongo.

Finding All Events for a Particular Host/Date

The following example describes a more complex query for returning all events in the collection for a particular host on a particular date. This kinds analysis may be useful for investigating suspicious behavior by a specific user.

Query Use a query that resembles the following:

```
>>> q_events = db.events.find({
...     'host': '127.0.0.1',
...     'time': {'$gte':datetime(2000,10,10),'$lt':datetime(2000,10,11)}
... })
```

This query selects *documents* from the `events` collection where the `host` field is `127.0.0.1` (i.e. local host), and the value of the `time` field represents a date that is on or after (i.e. `$gte`) `2000-10-10` but before (i.e. `$lt`) `2000-10-11`.

Index Support The indexes you use may have significant implications for the performance of these kinds of queries. For instance, you *can* create a compound index on the `time` and `host` field, using the following command:

```
>>> db.events.ensure_index([('time', 1), ('host', 1)])
```

To analyze the performance for the above query using this index, issue the `q_events.explain()` method in a Python console. This will return something that resembles:

```
{ ...
  u'cursor': u'BtreeCursor time_1_host_1',
  u'indexBounds': {u'host': [[u'127.0.0.1', u'127.0.0.1']],
  u'time': [
      [ datetime.datetime(2000, 10, 10, 0, 0),
        datetime.datetime(2000, 10, 11, 0, 0)]]
  },
  ...
  u'millis': 4,
  u'n': 11,
  u'nscanned': 1296,
  u'nscannedObjects': 11,
  ... }
```

This query had to scan 1296 items from the index to return 11 objects in 4 milliseconds. Conversely, you can test a different compound index with the `host` field first, followed by the `time` field. Create this index using the following operation:

```
>>> db.events.ensure_index([('host', 1), ('time', 1)])
```

Use the `q_events.explain()` operation to test the performance:

```
{ ...
  u'cursor': u'BtreeCursor host_1_time_1',
  u'indexBounds': {u'host': [[u'127.0.0.1', u'127.0.0.1']],
  u'time': [[datetime.datetime(2000, 10, 10, 0, 0),
      datetime.datetime(2000, 10, 11, 0, 0)]]},
  ...
  u'millis': 0,
  u'n': 11,
  ...
  u'nscanned': 11,
  u'nscannedObjects': 11,
  ...
}
```

Here, the query had to scan 11 items from the index before returning 11 objects in less than a millisecond. By placing the more selective element of your query *first* in a compound index you may be able to build more useful queries.

Note: Although the index order has an impact query performance, remember that index scans are *much* faster than collection scans, and depending on your other queries, it may make more sense to use the `{ time: 1, host: 1 }` index depending on usage profile.

See Also:

The `db.events.ensureIndex()` JavaScript method and the `db.events.ensure_index()` method in PyMongo.

Counting Requests by Day and Page

The following example describes the process for using the collection of Apache access events to determine the number of request per resource (i.e. page) per day in the last month.

Aggregation New in version 2.1. The *aggregation framework* provides the capacity for queries that select, process, and aggregate results from large numbers of documents. The `aggregate()` (and `aggregate` *command*) offers greater flexibility, capacity with less complexity than the existing `mapReduce` and `group` aggregation commands.

Consider the following aggregation *pipeline*: [1]

```
>>> result = db.command('aggregate', 'events', pipeline=[
...         {  '$match': {
...               'time': {
...                   '$gte': datetime(2000,10,1),
...                   '$lt':  datetime(2000,11,1) } } },
...         {  '$project': {
...                 'path': 1,
...                 'date': {
...                     'y': { '$year': '$time' },
...                     'm': { '$month': '$time' },
...                     'd': { '$dayOfMonth': '$time' } } } },
...         { '$group': {
...                 '_id': {
...                     'p':'$path',
...                     'y': '$date.y',
...                     'm': '$date.m',
...                     'd': '$date.d' },
...                 'hits': { '$sum': 1 } } },
...         ])
```

This command aggregates documents from the `events` collection with a pipeline that:

1. Uses the `$match` to limit the documents that the aggregation framework must process. `$match` is similar to a `find()` query.

   This operation selects all documents where the value of the `time` field represents a date that is on or after (i.e. `$gte`) `2000-10-10` but before (i.e. `$lt`) `2000-10-11`.

2. Uses the `$project` to limit the data that continues through the pipeline. This operator:

   - Selects the `path` field.
   - Creates a `y` field to hold the year, computed from the `time` field in the original documents.
   - Creates a `m` field to hold the month, computed from the `time` field in the original documents
   - Creates a `d` field to hold the day, computed from the `time` field in the original documents.

3. Uses the `$group` to create new computed documents. This step will create a single new document for each unique path/date combination. The documents take the following form:

   - the `_id` field holds a sub-document with the contents `path` field from the original documents in the `p` field, with the `date` fields from the `$project` as the remaining fields.
   - the `hits` field use the `$sum` statement to increment a counter for every document in the group. In the aggregation output, this field holds the total number of documents at the beginning of the aggregation pipeline with this unique date and path.

Note: In sharded environments, the performance of aggregation operations depends on the *shard key*. Ideally, all the items in a particular `$group` operation will reside on the same server.

While this distribution of documents would occur if you chose the `time` field as the shard key, a field like `path` also has this property and is a typical choice for sharding. Also see the "*sharding considerations* (page xii)." of this document for additional recommendations for using sharding.

See Also:

[1] To translate statements from the `aggregation framework` to SQL, you can consider the `$match` equivalent to `WHERE`, `$project` to `SELECT`, and `$group` to `GROUP BY`.

“`http://docs.mongodb.org/manual/applications/aggregation`“

Index Support To optimize the aggregation operation, ensure that the initial `$match` query has an index. Use the following command to create an index on the `time` field in the `events` collection:

```
>>> db.events.ensure_index('time')
```

Note: If you have already created a compound index on the `time` and `host` (i.e. `{ time: 1, host, 1 }`,) MongoDB will use this index for range queries on just the `time` field. Do not create an additional index, in these situations.

Sharding

Eventually your system’s events will exceed the capacity of a single event logging database instance. In these situations you will want to use a *sharded cluster*, which takes advantage of MongoDB’s *sharding* functionality. This section introduces the unique sharding concerns for this event logging case.

See Also:

“`http://docs.mongodb.org/manual/faq/sharding`” and the “Sharding wiki page.

Limitations

In a sharded environment the limitations on the maximum insertion rate are:

- the number of shards in the cluster.
- the *shard key* you chose.

Because MongoDB distributed data in using “ranges” (i.e. *chunks*) of *keys*, the choice of shard key can control how MongoDB distributes data and the resulting systems’ capacity for writes and queries.

Ideally, your shard key should allow insertions balance evenly among the shards [2] and for most queries to only *need* to access a single shard. [3] Continue reading for an analysis of a collection of shard key choices.

Shard by Time

While using the timestamp, or the `ObjectId` in the `_id` field, [4] would distribute your data evenly among shards, these keys lead to two problems:

1. All inserts always flow to the same shard, which means that your *sharded cluster* will have the same write throughput as a standalone instance.
2. Most reads will tend to cluster on the same shard, as analytics queries.

[2] For this reason, avoid shard keys based on the timestamp or the insertion time (i.e. the `ObjectId`) because all writes will end up on a single node.

[3] For this reason, avoid randomized shard keys (e.g. hash based shard keys) because any query will have to access all shards in the cluster.

[4] The `ObjectId` derives from the creation time, and is effectively a timestamp in this case.

Shard by a Semi-Random Key

To distribute data more evenly among the shards, you may consider using a more "random" piece of data, such as a hash of the `_id` field (i.e. the `ObjectId` as a *shard key*.

While this introduces some additional complexity into your application, to generate the key, it will distribute writes among the shards. In these deployments having 5 shards will provide 5 times the write capacity as a single instance.

Using this shard key, or any hashed value as a key presents the following downsides:

- the shard key, and the index on the key will consume additional space in the database.
- queries, unless they include the shard key itself, [5] must run in parallel on all shards, which may lead to degraded performance.

This might be an acceptable trade-off in some situations. The workload of event logging systems tends to be heavily skewed toward writing, read performance may not be as critical as more robust write performance.

Shard by an Evenly-Distributed Key in the Data Set

If a field in your documents has values that are evenly distributed among the documents, you may consider using this key as a *shard key*.

Continuing the example from above, you may consider using the `path` field. Which may have a couple of advantages:

1. writes will tend to balance evenly among shards.
2. reads will tend to be selective and local to a single shard if the query selects on the `path` field.

There are a few potential problems with these kinds of shard keys:

1. If a large number of documents will have the same shard key, you run the risk of having a portion of your data collection MongoDB cannot distribute throughout the cluster.
2. If there are a small number of possible values, there may be a limit to how much MongoDB will be able to distribute the data among the shard.

Note: Test using your existing data to ensure that the distribution is truly even, and that there is a sufficient quantity of distinct values for the shard key.

Shard by Combine a Natural and Synthetic Key

MongoDB supports compound *shard keys* that combine the best aspects of *sharding by a evenly distributed key in the set* (page xiii) and *sharding by a random key* (page xiii). In these situations, the shard key would resemble `{ path: 1 , ssk: 1 }` where, `path` is an often used "natural key, or value from your data and `ssk` is a hash of the `_id` field. [6]

Using this type of shard key, data is largely distributed by the natural key, or `path`, which makes most queries that access the `path` field local to a single shard or group of shards. At the same time, if there is not sufficient distribution for specific values of `path`, the `ssk` makes it possible for MongoDB to create *chunks* and data across the cluster.

In most situations, these kinds of keys provide the ideal balance between distributing writes across the cluster and ensuring that most queries will only need to access a select number of shards.

[5] Typically, it is difficult to use these kinds of shard keys in queries.

[6] You must still calculate the value of this synthetic key in your application when you insert documents into your collection.

Test with Your Own Data

Selecting shard keys is difficult because: there are no definitive "best-practices," the decision has a large impact on performance, and it is difficult or impossible to change the shard key after making the selection.

The *sharding options* (page xii) provides a good starting point for thinking about *shard key* selection. Nevertheless, the best way to select a shard key is to analyze the actual insertions and queries from your own application.

Managing Event Data Growth

Without some strategy for managing the size of your database, most event logging systems can grow infinitely. This is particularly important in the context of MongoDB may not relinquish data to the file system in the way you might expect. Consider the following strategies for managing data growth:

Capped Collections

Depending on your data retention requirements as well as your reporting and analytics needs, you may consider using a *capped collection* to store your events. Capped collections have a fixed size, and drop old data when inserting new data after reaching cap.

Note: In the current version, it is not possible to shard capped collections.

Multiple Collections, Single Database

Strategy: Periodically rename your event collection so that your data collection rotates in much the same way that you might rotate log files. When needed, you can drop the oldest collection from the database.

This approach has several advantages over the single collection approach:

1. Collection renames are fast and atomic.
2. MongoDB does not bring any document into memory to drop a collection.
3. MongoDB can effectively reuse space freed by removing entire collections without leading to data fragmentation.

Nevertheless, this operation may increase some complexity for queries, if any of your analyses depend on events that may reside in the current and previous collection. For most real time data collection systems, this approach is the most ideal.

Multiple Databases

Strategy: Rotate databases rather than collections, as in the "*Multiple Collections, Single Database* (page xiv) example.

While this *significantly* increases application complexity for insertions and queries, when you drop old databases, MongoDB will return disk space to the file system. This approach makes the most sense in scenarios where your event insertion rates and/or your data retention rates were extremely variable.

For example, if you are performing a large backfill of event data and want to make sure that the entire set of event data for 90 days is available during the backfill, during normal operations you only need 30 days of event data, you might consider using multiple databases.

2.2 Pre-Aggregated Reports

Overview

This document outlines the basic patterns and principles for using MongoDB as an engine for collecting and processing events in real time for use in generating up to the minute or second reports.

Problem

Servers and other systems can generate a large number of documents, and it can be difficult to access and analyze such large collections of data originating from multiple servers.

This document makes the following assumptions about real-time analytics:

- There is no need to retain transactional event data in MongoDB, and how your application handles transactions is outside of the scope of this document.
- You require up-to-the minute data, or up-to-the-second if possible.
- The queries for ranges of data (by time) must be as fast as possible.

See Also:

"*Storing Log Data* (page v)."

Solution

The solution described below assumes a simple scenario using data from web server access logs. With this data, you will want to return the number of hits to a collection of web sites at various levels of granularity based on time (i.e. by minute, hour, day, week, and month) as well as by the path of a resource.

To achieve the required performance to support these tasks, *upserts* and `increment` operations will allow you to calculate statistics, produce simple range-based queries, and generate filters to support time-series charts of aggregated data.

Schema

Schemas for real-time analytics systems must support simple and fast query and update operations. In particular, attempt to avoid the following situations which can degrade performance:

- *documents* growing significantly after creation.

  Document growth forces MongoDB to move the document on disk, which can be time and resource consuming relative to other operations;
- queries requiring MongoDB to scan documents in the collection without using indexes; and
- deeply nested documents that make accessing particular fields slow.

Intuitively, you may consider keeping "hit counts" in individual documents with one document for every unit of time (i.e. minute, hour, day, etc.) However, queries must return multiple documents for all non-trivial time-rage queries, which can slow overall query performance.

Preferably, to maximize query performance, use more complex documents, and keep several aggregate values in each document. The remainder of this section outlines several schema designs that you may consider for this real-time analytics system. While there is no single pattern for every problem, each pattern is more well suited to specific classes of problems.

One Document Per Page Per Day

Consider the following example schema for a solution that stores all statistics for a single day and page in a single *document*:

```
{
    _id: "20101010/site-1/apache_pb.gif",
    metadata: {
        date: ISODate("2000-10-10T00:00:00Z"),
        site: "site-1",
        page: "/apache_pb.gif" },
    daily: 5468426,
    hourly: {
        "0": 227850,
        "1": 210231,
        ...
        "23": 20457 },
    minute: {
        "0": 3612,
        "1": 3241,
        ...
        "1439": 2819 }
}
```

This approach has a couple of advantages:

- For every request on the website, you only need to update one document.
- Reports for time periods within the day, for a single page require fetching a single document.

There are, however, significant issues with this approach. The most significant issue is that, as you *upsert* data into the `hourly` and `monthly` fields, the document grows. Although MongoDB will pad the space allocated to documents, it must still will need to reallocate these documents multiple times throughout the day, which impacts performance.

Pre-allocate Documents

Simple Pre-Allocation To mitigate the impact of repeated document migrations throughout the day, you can tweak the "*one document per page per day* (page xvi)" approach by adding a process that "pre-allocates" documents with fields that hold `0` values throughout the previous day. Thus, at midnight, new documents will exist.

Note: To avoid situations where your application must pre-allocate large numbers of documents at midnight, it's best to create documents throughout the previous day by *upserting* randomly when you update a value in the current day's data.

This requires some tuning, to balance two requirements:

1. your application should have pre-allocated all or nearly all of documents by the end of the day.
2. your application should infrequently pre-allocate a document that already exists to save time and resources on extraneous upserts.

As a starting point, consider the average number of hits a day (`h`), and then upsert a blank document upon update with a probability of `1/h`.

Pre-allocating increases performance by initializing all documents with `0` values in all fields. After create, documents will never grow. This means that:

1. there will be no need to migrate documents within the data store, which is a problem in the "*one document per page per day* (page xvi)" approach.
2. MongoDB will not add padding to the records, which leads to a more compact data representation and better memory use of your memory.

Add Intra-Document Hierarchy

Note: MongoDB stores *BSON documents* as a sequence of fields and values, *not* as a hash table. As a result, writing to the field `stats.mn.0` is considerably faster than writing to `stats.mn.1439`.

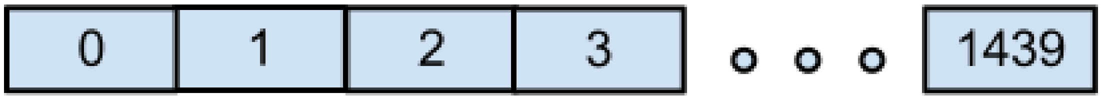


Figure 1: In order to update the value in minute #1349, MongoDB must skip over all 1349 entries before it.

To optimize update and insert operations you can introduce intra-document hierarchy. In particular, you can split the `minute` field up into 24 hourly fields:

```
{
    _id: "20101010/site-1/apache_pb.gif",
    metadata: {
        date: ISODate("2000-10-10T00:00:00Z"),
        site: "site-1",
        page: "/apache_pb.gif" },
    daily: 5468426,
    hourly: {
        "0": 227850,
        "1": 210231,
        ...
        "23": 20457 },
    minute: {
        "0": {
            "0": 3612,
            "1": 3241,
            ...
            "59": 2130 },
        "1": {
        "60": ... ,
        },
        ...
        "23": {
            ...
            "1439": 2819 }
    }
}
```

This allows MongoDB to "skip forward" throughout the day when updating the minute data, which makes the update performance more uniform and faster later in the day.

Separate Documents by Granularity Level

Pre-allocating documents (page xvi) is a reasonable design for storing intra-day data, but the model breaks down when displaying data over longer multi-day periods like months or quarters. In these cases, consider storing daily statistics in a single document as above, and then aggregate monthly data into a separate document.

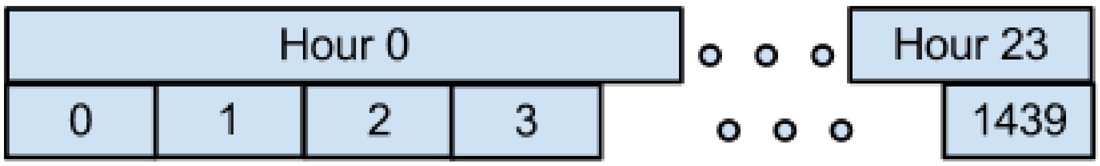


Figure 2: To update the value in minute #1349, MongoDB first skips the first 23 hours and then skips 59 minutes for only 82 skips as opposed to 1439 skips in the previous schema.

This introduce a second set of *upsert* operations to the data collection and aggregation portion of your application but the gains reduction in disk seeks on the queries, should be worth the costs. Consider the following example schema:

1. Daily Statistics

```
{
    _id: "20101010/site-1/apache_pb.gif",
    metadata: {
        date: ISODate("2000-10-10T00:00:00Z"),
        site: "site-1",
        page: "/apache_pb.gif" },
    hourly: {
        "0": 227850,
        "1": 210231,
        ...
        "23": 20457 },
    minute: {
        "0": {
            "0": 3612,
            "1": 3241,
            ...
            "59": 2130 },
        "1": {
            "0": ...,
        },
        ...
        "23": {
            "59": 2819 }
    }
}
```

2. Monthly Statistics

```
{
    _id: "201010/site-1/apache_pb.gif",
    metadata: {
        date: ISODate("2000-10-00T00:00:00Z"),
        site: "site-1",
        page: "/apache_pb.gif" },
    daily: {
        "1": 5445326,
        "2": 5214121,
        ... }
}
```

Operations

This section outlines a number of common operations for building and interacting with real-time-analytics reporting system. The major challenge is in balancing performance and write (i.e. *upsert*) performance. All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Log an Event

Logging an event such as a page request (i.e. "hit") is the main "write" activity for your system. To maximize performance, you'll be doing in-place updates with the *upsert* operation. Consider the following example:

```
from datetime import datetime, time

def log_hit(db, dt_utc, site, page):

    # Update daily stats doc
    id_daily = dt_utc.strftime('%Y%m%d/') + site + page
    hour = dt_utc.hour
    minute = dt_utc.minute

    # Get a datetime that only includes date info
    d = datetime.combine(dt_utc.date(), time.min)
    query = {
        '_id': id_daily,
        'metadata': { 'date': d, 'site': site, 'page': page } }
    update = { '$inc': {
            'hourly.%d' % (hour,): 1,
            'minute.%d.%d' % (hour,minute): 1 } }
    db.stats.daily.update(query, update, upsert=True)

    # Update monthly stats document
    id_monthly = dt_utc.strftime('%Y%m/') + site + page
    day_of_month = dt_utc.day
    query = {
        '_id': id_monthly,
        'metadata': {
            'date': d.replace(day=1),
            'site': site,
            'page': page } }
    update = { '$inc': {
            'daily.%d' % day_of_month: 1} }
    db.stats.monthly.update(query, update, upsert=True)
```

The upsert operation (i.e. `upsert=True`) performs an update if the document exists, and an insert if the document does not exist.

Note: This application requires upserts, because the *pre-allocation* (page xx) method only pre-allocates new documents with a high probability, not with complete certainty.

Without preallocation, you end up with a dynamically growing document, slowing upserts as MongoDB moves documents to accommodate growth.

Pre-allocate

To prevent document growth, you can preallocate new documents before the system needs them. As you create new documents, set all values to `0` for so that documents will not grow to accommodate updates. Consider the following `preallocate()` function:

```
def preallocate(db, dt_utc, site, page):

    # Get id values
    id_daily = dt_utc.strftime('%Y%m%d/') + site + page
    id_monthly = dt_utc.strftime('%Y%m/') + site + page

    # Get daily metadata
    daily_metadata = {
        'date': datetime.combine(dt_utc.date(), time.min),
        'site': site,
        'page': page }
    # Get monthly metadata
    monthly_metadata = {
        'date': daily_m['d'].replace(day=1),
        'site': site,
        'page': page }

    # Initial zeros for statistics
    hourly = dict((str(i), 0) for i in range(24))
    minute = dict(
        (str(i), dict((str(j), 0) for j in range(60)))
        for i in range(24))
    daily = dict((str(i), 0) for i in range(1, 32))

    # Perform upserts, setting metadata
    db.stats.daily.update(
        {
            '_id': id_daily,
            'hourly': hourly,
            'minute': minute},
        { '$set': { 'metadata': daily_metadata }},
        upsert=True)
    db.stats.monthly.update(
        {
            '_id': id_monthly,
            'daily': daily },
        { '$set': { 'm': monthly_metadata }},
        upsert=True)
```

The function pre-allocated both the monthly *and* daily documents at the same time. The performance benefits from separating these operations are negligible, so it's reasonable to keep both operations in the same function.

Ideally, your application should pre-allocate documents *before* needing to write data to maintain consistent update performance. Additionally, its important to avoid causing a spike in activity and latency by creating documents all at once.

In the following example, document updates (i.e. "`log_hit()`") will also pre-allocate a document probabilistically. However, by "tuning probability," you can limit redundant `preallocate()` calls.

```
from random import random
from datetime import datetime, timedelta, time

# Example probability based on 500k hits per day per page
```

```
prob_preallocate = 1.0 / 500000

def log_hit(db, dt_utc, site, page):
    if random.random() < prob_preallocate:
        preallocate(db, dt_utc + timedelta(days=1), site_page)
    # Update daily stats doc
    ...
```

Using this method, there will be a high probability that each document will already exist before your application needs to issue update operations. You'll also be able to prevent a regular spike in activity for pre-allocation, and be able to eliminate document growth.

Retrieving Data for a Real-Time Chart

This example describes fetching the data from the above MongoDB system, for use in generating a chart that displays the number of hits to a particular resource over the last hour.

Querying Use the following query in a `find_one` operation at the Python/PyMongo console to retrieve the number of hits to a specific resource (i.e. `http://docs.mongodb.org/manual/index.html`) with minute-level granularity:

```
>>>``db.stats.daily.find_one(
...     {'metadata': {'date':dt, 'site':'site-1', 'page':'/index.html'}},
...     { 'minute': 1 })
```

Use the following query to retrieve the number of hits to a resource over the last day, with hour-level granularity:

```
>>> db.stats.daily.find_one(
...     {'metadata': {'date':dt, 'site':'site-1', 'page':'/foo.gif'}},
...     { 'hy': 1 })
```

If you want a few days of hourly data, you can use a query in the following form:

```
>>> db.stats.daily.find(
...     {
...         'metadata.date': { '$gte': dt1, '$lte': dt2 },
...         'metadata.site': 'site-1',
...         'metadata.page': '/index.html'},
...     { 'metadata.date': 1, 'hourly': 1 } },
...     sort=[('metadata.date', 1)])
```

Indexing To support these query operation, create a compound index on the following daily statistics fields: `metadata.site`, `metadata.page`, and `metadata.date` (in that order.) Use the following operation at the Python/PyMongo console.

```
>>> db.stats.daily.ensure_index([
...     ('metadata.site', 1),
...     ('metadata.page', 1),
...     ('metadata.date', 1)])
```

This index makes it possible to efficiently run the query for multiple days of hourly data. At the same time, any compound index on page and date, will allow you to query efficiently for a single day's statistics.

Get Data for a Historical Chart

Querying To retrieve daily data for a single month, use the following query:

```
>>> db.stats.monthly.find_one(
...     {'metadata':
...         {'date':dt,
...         'site': 'site-1',
...         'page':'/index.html'}},
...     { 'daily': 1 })
```

To retrieve several months of daily data, use a variation on the above query:

```
>>> db.stats.monthly.find(
...     {
...         'metadata.date': { '$gte': dt1, '$lte': dt2 },
...         'metadata.site': 'site-1',
...         'metadata.page': '/index.html'},
...     { 'metadata.date': 1, 'daily': 1 } },
...     sort=[('metadata.date', 1)])
```

Indexing Create the following index to support these queries for monthly data on the `metadata.site`, `metadata.page`, and `metadata.date` fields:

```
>>> db.stats.monthly.ensure_index([
...     ('metadata.site', 1),
...     ('metadata.page', 1),
...     ('metadata.date', 1)])
```

This field order will efficiently support range queries for a single page over several months.

Sharding

The only potential limits on the performance of this system are the number of *shards* in your *system*, and the *shard key* that you use.

> An ideal shard key will distribute *upserts* between the shards while routing all queries to a single shard, or a small number of shards.

While your choice of shard key may depend on the precise workload of your deployment, consider using `{ metadata.site: 1, metadata.page: 1 }` as a *shard key*. The combination of site and page (or event) will lead to a well balanced cluster for most deployments.

Enable sharding for the daily statistics collection with the following `shardCollection` command in the Python/PyMongo console:

```
>>> db.command('shardCollection', 'stats.daily', {
...     key : { 'metadata.site': 1, 'metadata.page' : 1 } })
```

Upon success, you will see the following response:

```
{ "collectionsharded" : "stats.daily", "ok" : 1 }
```

Enable sharding for the monthly statistics collection with the following `shardCollection` command in the Python/PyMongo console:

```
>>> db.command('shardCollection', 'stats.monthly', {
...     key : { 'metadata.site': 1, 'metadata.page' : 1 } })
```

Upon success, you will see the following response:

```
{ "collectionsharded" : "stats.monthly", "ok" : 1 }
```

One downside of the `{ metadata.site: 1, metadata.page: 1 }` *shard key* is: if one page dominates all your traffic, all updates to that page will go to a single shard. This is basically unavoidable, since all update for a single page are going to a single *document*.

You may wish to include the date in addition to the site, and page fields so that MongoDB can split histories so that you can serve different historical ranges with different shards. Use the following `shardCollection` command to shard the daily statistics collection in the Python/PyMongo console:

```
>>> db.command('shardCollection', 'stats.daily', {
...     'key':{'metadata.site':1,'metadata.page':1,'metadata.date':1}})
{ "collectionsharded" : "stats.daily", "ok" : 1 }
```

Enable sharding for the monthly statistics collection with the following `shardCollection` command in the Python/PyMongo console:

```
>>> db.command('shardCollection', 'stats.monthly', {
...     'key':{'metadata.site':1,'metadata.page':1,'metadata.date':1}})
{ "collectionsharded" : "stats.monthly", "ok" : 1 }
```

Note: Determine your actual requirements and load before deciding to shard. In many situations a single MongoDB instance may be able to keep track of all events and pages.

2.3 Hierarchical Aggregation

Overview

Background

If you collect a large amount of data, but do not *pre-aggregate* (page xv), and you want to have access to aggregated information and reports, then you need a method to aggregate these data into a usable form. This document provides an overview of these aggregation patterns and processes.

For clarity, this case study assumes that the incoming event data resides in a collection named `events`. For details on how you might get the event data into the events collection, please see "*Storing Log Data* (page v)" document. This document continues using this example.

Solution

The first step in the aggregation process is to aggregate event data into the finest required granularity. Then use this aggregation to generate the next least specific level granularity and this repeat process until you have generated all required views.

The solution uses several collections: the raw data (i.e. `events`) collection as well as collections for aggregated hourly, daily, weekly, monthly, and yearly statistics. All aggregations use the `mapReduce` *command*, in a hierarchical process. The following figure illustrates the input and output of each job:

Note: Aggregating raw events into an hourly collection is qualitatively different from the operation that aggregates hourly statistics into the daily collection.

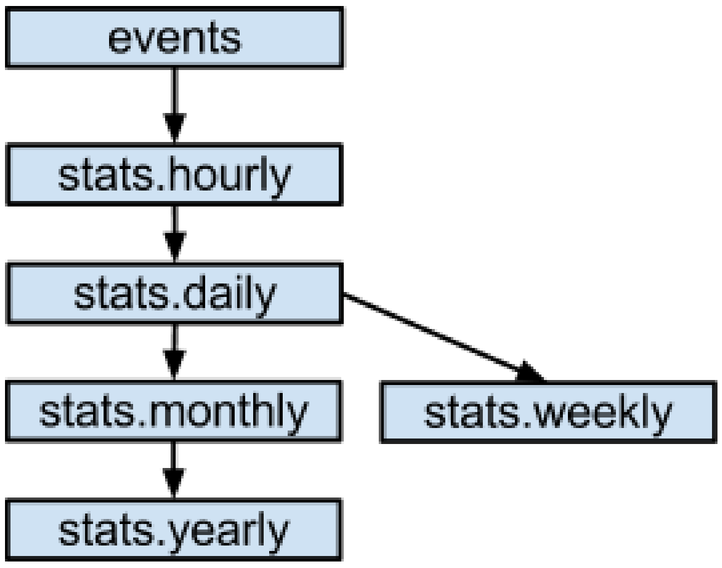


Figure 3: Hierarchy of data aggregation.

See Also:

Map-reduce and the MapReduce wiki page for more information on the Map-reduce data aggregation paradigm.

Schema

When designing the schema for event storage, it's important to track the events included in the aggregation and events that are not yet included.

Relational Approach

A simple tactic from relational database, uses an auto-incremented integer as the primary key. However, this introduces a significant performance penalty for event logging process because the aggregation process must fetch new keys one at a time.

If you can batch your inserts into the `events` collection, you can use an auto-increment primary key by using the `find_and_modify` command to generate the `_id` values, as in the following example:

```
>>> obj = db.my_sequence.find_and_modify(
...     query={'_id':0},
...     update={'$inc': {'inc': 50}}
...     upsert=True,
...     new=True)
>>> batch_of_ids = range(obj['inc']-50, obj['inc'])
```

However, in most cases you can simply include a timestamp with each event that you can use to distinguish processed events from unprocessed events.

This example assumes that you are calculating average session length for logged-in users on a website. The events will have the following form:

```
{
    "userid": "rick",
    "ts": ISODate('2010-10-10T14:17:22Z'),
    "length":95
}
```

The operations described in the next session will calculate total and average session times for each user at the hour, day, week, month and year. For each aggregation you will want to store the number of sessions so that MongoDB can incrementally recompute the average session times. The aggregate document will resemble the following:

```
{
   _id: { u: "rick", d: ISODate("2010-10-10T14:00:00Z") },
   value: {
       ts: ISODate('2010-10-10T15:01:00Z'),
       total: 254,
       count: 10,
       mean: 25.4 }
}
```

Note: The timestamp value in the `_id` sub-document, which will allow you to incrementally update documents at various levels of the hierarchy.

Operations

This section assumes that all events exist in the `events` collection and have a timestamp. The operations, thus are to aggregate from the `events` collection into the smallest aggregate–hourly totals– and then aggregate from the hourly totals into coarser granularity levels. In all cases, these operations will store aggregation time as a `last_run` variable.

Creating Hourly Views from Event Collections

Aggregation

Note: Although this solution uses Python and PyMongo to connect with MongoDB, you must pass JavaScript functions (i.e. `mapf`, `reducef`, and `finalizef`) to the `mapReduce` command.

Begin by creating a map function, as below:

```
mapf_hour = bson.Code('''function() {
    var key = {
        u: this.userid,
        d: new Date(
            this.ts.getFullYear(),
            this.ts.getMonth(),
            this.ts.getDate(),
            this.ts.getHours(),
            0, 0, 0);
    emit(
        key,
        {
            total: this.length,
            count: 1,
            mean: 0,
            ts: new Date(); });
}''')
```

In this case, it emits key-value pairs that contain the data you want to aggregate as you'd expect. The function also emits a `ts` value that makes it possible to cascade aggregations to coarser grained aggregations (i.e. hour to day, etc.)

Consider the following reduce function:

```
reducef = bson.Code('''function(key, values) {
    var r = { total: 0, count: 0, mean: 0, ts: null };
    values.forEach(function(v) {
        r.total += v.total;
        r.count += v.count;
    });
    return r;
}''')
```

The reduce function returns a document in the same format as the output of the map function. This pattern for map and reduce functions makes map-reduce processes easier to test and debug.

While the reduce function ignores the `mean` and `ts` (timestamp) values, the finalize step, as follows, computes these data:

```
finalizef = bson.Code('''function(key, value) {
    if(value.count > 0) {
       value.mean = value.total / value.count;
    }
    value.ts = new Date();
    return value;
}''')
```

With the above function the `map_reduce` operation itself will resemble the following:

```
cutoff = datetime.utcnow() - timedelta(seconds=60)
query = { 'ts': { '$gt': last_run, '$lt': cutoff } }

db.events.map_reduce(
    map=mapf_hour,
    reduce=reducef,
    finalize=finalizef,
    query=query,
    out={ 'reduce': 'stats.hourly' })

last_run = cutoff
```

The `cutoff` variable allows you to process all events that have occurred since the last run but before 1 minute ago. This allows for some delay in logging events. You can safely run this aggregation as often as you like, provided that you update the `last_run` variable each time.

Indexing Create an index on the timestamp (i.e. the `ts` field) to support the query selection of the `map_reduce` operation. Use the following operation at the Python/PyMongo console:

```
>>> db.events.ensure_index('ts')
```

Deriving Day-Level Data

Aggregation To calculate daily statistics, use the hourly statistics as input. Begin with the following map function:

```
mapf_day = bson.Code('''function() {
    var key = {
        u: this._id.u,
        d: new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            this._id.d.getDate(),
            0, 0, 0, 0) };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
            ts: null });
}''')
```

The map function for deriving day-level data differs from the initial aggregation above in the following ways:

- the aggregation key is the (userid, date) rather than (userid, hour) to support daily aggregation.
- the keys and values emitted (i.e. `emit()`) are actually the total and count values from the hourly aggregates rather than properties from event documents.

  This is the case for all the higher-level aggregation operations.

Because the output of this map function is the same as the previous map function, you can use the same reduce and finalize functions.

The actual code driving this level of aggregation is as follows:

```
cutoff = datetime.utcnow() - timedelta(seconds=60)
query = { 'value.ts': { '$gt': last_run, '$lt': cutoff } }

db.stats.hourly.map_reduce(
    map=mapf_day,
    reduce=reducef,
    finalize=finalizef,
    query=query,
    out={ 'reduce': 'stats.daily' })

last_run = cutoff
```

There are a couple of things to note here. First of all, the query is not on `ts` now, but `value.ts`, the timestamp written during the finalization of the hourly aggregates. Also note that you are, in fact, aggregating from the `stats.hourly` collection into the `stats.daily` collection.

Indexing Because you will run the query option regularly which finds on the `value.ts` field, you may wish to create an index to support this. Use the following operation in the Python/PyMongo shell to create this index:

```
>>> db.stats.hourly.ensure_index('value.ts')
```

Weekly and Monthly Aggregation

Aggregation You can use the aggregated day-level data to generate weekly and monthly statistics. A map function for generating weekly data follows:

```
mapf_week = bson.Code('''function() {
    var key = {
        u: this._id.u,
        d: new Date(
            this._id.d.valueOf()
            - dt.getDay()*24*60*60*1000) };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
            ts: null });
}''')
```

Here, to get the group key, the function takes the current and subtracts days until you get the beginning of the week. In the weekly map function, you'll use the first day of the month as the group key, as follows:

```
mapf_month = bson.Code('''function() {
        d: new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            1, 0, 0, 0, 0) };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
            ts: null });
}''')
```

These map functions are identical to each other except for the date calculation.

Indexing Create additional indexes to support the weekly and monthly aggregation options on the `value.ts` field. Use the following operation in the Python/PyMongo shell.

```
>>> db.stats.daily.ensure_index('value.ts')
>>> db.stats.monthly.ensure_index('value.ts')
```

Refactor Map Functions Use Python's string interpolation to refactor the map function definitions as follows:

```
mapf_hierarchical = '''function() {
    var key = {
        u: this._id.u,
        d: %s };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
            ts: null });
}'''

mapf_day = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            this._id.d.getDate(),
            0, 0, 0, 0)''')

mapf_week = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.valueOf()
            - dt.getDay()*24*60*60*1000)''')

mapf_month = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            1, 0, 0, 0, 0)''')

mapf_year = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.getFullYear(),
            1, 1, 0, 0, 0, 0)''')
```

You can create a `h_aggregate` function to wrap the `map_reduce` operation, as below, to reduce code duplication:

```
def h_aggregate(icollection, ocollection, mapf, cutoff, last_run):
     query = { 'value.ts': { '$gt': last_run, '$lt': cutoff } }
     icollection.map_reduce(
         map=mapf,
         reduce=reducef,
         finalize=finalizef,
         query=query,
         out={ 'reduce': ocollection.name })
```

With `h_aggregate` defined, you can perform all aggregation operations as follows:

```
cutoff = datetime.utcnow() - timedelta(seconds=60)

h_aggregate(db.events, db.stats.hourly, mapf_hour, cutoff, last_run)
h_aggregate(db.stats.hourly, db.stats.daily, mapf_day, cutoff, last_run)
h_aggregate(db.stats.daily, db.stats.weekly, mapf_week, cutoff, last_run)
h_aggregate(db.stats.daily, db.stats.monthly, mapf_month, cutoff, last_run)
h_aggregate(db.stats.monthly, db.stats.yearly, mapf_year, cutoff, last_run)

last_run = cutoff
```

As long as you save and restore the `last_run` variable between aggregations, you can run these aggregations as often as you like since each aggregation operation is incremental.

Sharding

Ensure that you choose a *shard key* that is not the incoming timestamp, but rather something that varies significantly in the most recent documents. In the example above, consider using the `userid` as the most significant part of the shard key.

To prevent a single, active user from creating a large, *chunk* that MongoDB cannot split, use a compound shard key with (username, timestamp) on the events collection. Consider the following:

```
>>> db.command('shardCollection','events', {
... 'key' : { 'userid': 1, 'ts' : 1} } )
{ "collectionsharded": "events", "ok" : 1 }
```

To shard the aggregated collections you must use the `_id` field, so you can issue the following group of shard operations in the Python/PyMongo shell:

```
db.command('shardCollection', 'stats.daily', {
     'key': { '_id': 1 } })
db.command('shardCollection', 'stats.weekly', {
     'key': { '_id': 1 } })
db.command('shardCollection', 'stats.monthly', {
     'key': { '_id': 1 } })
db.command('shardCollection', 'stats.yearly', {
     'key': { '_id': 1 } })
```

You should also update the `h_aggregate` map-reduce wrapper to support sharded output Add `'sharded':True` to the `out` argument. See the full sharded `h_aggregate` function:

```
def h_aggregate(icollection, ocollection, mapf, cutoff, last_run):
     query = { 'value.ts': { '$gt': last_run, '$lt': cutoff } }
     icollection.map_reduce(
         map=mapf,
         reduce=reducef,
         finalize=finalizef,
         query=query,
         out={ 'reduce': ocollection.name, 'sharded': True })
```

3 Product Data Management

MongoDB's flexible schema makes it particularly well suited to storing information for product data management and e-commerce websites and solutions. The "*Product Catalog* (page xxx)" document describes methods and practices for modeling and managing a product catalog using MongoDB, while the "*Inventory Management* (page xxxvii)" document introduces a pattern for handling interactions between inventory and users' shopping carts. Finally the "*Category Hierarchy* (page xliv)" document describes methods for interacting with category hierarchies in MongoDB.

3.1 Product Catalog

Overview

This document describes the basic patterns and principles for designing an E-Commerce product catalog system using MongoDB as a storage engine.

Problem

Product catalogs must have the capacity to store many differed types of objects with different sets of attributes. These kinds of data collections are quite compatible with MongoDB's data model, but many important considerations and design decisions remain.

Solution

For relational databases, there are several solutions that address this problem, each with a different performance profile. This section examines several of these options and then describes the preferred MongoDB solution.

SQL and Relational Data Models

Concrete Table Inheritance One approach, in a relational model, is to create a table for each product category. Consider the following example SQL statement for creating database tables:

```
CREATE TABLE `product_audio_album` (
    `sku` char(8) NOT NULL,
    ...
    `artist` varchar(255) DEFAULT NULL,
    `genre_0` varchar(255) DEFAULT NULL,
    `genre_1` varchar(255) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`))
...
CREATE TABLE `product_film` (
    `sku` char(8) NOT NULL,
    ...
    `title` varchar(255) DEFAULT NULL,
    `rating` char(8) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`))
...
```

This approach has limited flexibility for two key reasons:

- You must create a new table for every new category of products.
- You must explicitly tailor all queries for the exact type of product.

Single Table Inheritance Another relational data model uses a single table for all product categories and adds new columns anytime you need to store data regarding a new type of product. Consider the following SQL statement:

```
CREATE TABLE `product` (
    `sku` char(8) NOT NULL,
    ...
    `artist` varchar(255) DEFAULT NULL,
    `genre_0` varchar(255) DEFAULT NULL,
    `genre_1` varchar(255) DEFAULT NULL,
    ...
    `title` varchar(255) DEFAULT NULL,
    `rating` char(8) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`))
```

This approach is more flexible than concrete table inheritance: it allows single queries to span different product types, but at the expense of space.

Multiple Table Inheritance Also in the relational model, you may use a "multiple table inheritance" pattern to represent common attributes in a generic "product" table, with some variations in individual category product tables. Consider the following SQL statement:

```
CREATE TABLE `product` (
    `sku` char(8) NOT NULL,
    `title` varchar(255) DEFAULT NULL,
    `description` varchar(255) DEFAULT NULL,
    `price`, ...
    PRIMARY KEY(`sku`))

CREATE TABLE `product_audio_album` (
    `sku` char(8) NOT NULL,
    ...
    `artist` varchar(255) DEFAULT NULL,
    `genre_0` varchar(255) DEFAULT NULL,
    `genre_1` varchar(255) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`),
    FOREIGN KEY(`sku`) REFERENCES `product`(`sku`))
...
CREATE TABLE `product_film` (
    `sku` char(8) NOT NULL,
    ...
    `title` varchar(255) DEFAULT NULL,
    `rating` char(8) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`),
    FOREIGN KEY(`sku`) REFERENCES `product`(`sku`))
...
```

Multiple table inheritance is more space-efficient than *single table inheritance* (page xxxi) and somewhat more flexible than *concrete table inheritance* (page xxxi). However, this model does require an expensive `JOIN` operation to obtain all relevant attributes relevant to a product.

Entity Attribute Values The final substantive pattern from relational modeling is the entity-attribute-value schema where you would create a meta-model for product data. In this approach, you maintain a table with three columns, e.g. `entity_id`, `attribute_id`, `value`, and these triples describe each product.

Consider the description of an audio recording. You may have a series of rows representing the following relationships:

Entity	Attribute	Value
sku_00e8da9b	type	Audio Album
sku_00e8da9b	title	A Love Supreme
sku_00e8da9b	...	...
sku_00e8da9b	artist	John Coltrane
sku_00e8da9b	genre	Jazz
sku_00e8da9b	genre	General
...	...	...

This schema is totally flexible:

- any entity can have any set of any attributes.
- New product categories do not require *any* changes to the data model in the database.

The downside for these models, is that all nontrivial queries require large numbers of `JOIN` operations that results in large performance penalties.

Avoid Modeling Product Data Additionally some e-commerce solutions with relational database systems avoid choosing one of the data models above, and serialize all of this data into a `BLOB` column. While simple, the details become difficult to access for search and sort.

Non-Relational Data Model Because MongoDB is a non-relational database, the data model for your product catalog can benefit from this additional flexibility. The best models use a single MongoDB collection to store all the product data, which is similar to the *single table inheritance* (page xxxi) relational model. MongoDB's dynamic schema means that each *document* need not conform to the same schema. As a result, the document for each product only needs to contain attributes relevant to that product.

Schema

At the beginning of the document, the schema must contain general product information, to facilitate searches of the entire catalog. Then, a `details` sub-document that contains fields that vary between product types. Consider the following example document for an album product.

```
{
  sku: "00e8da9b",
  type: "Audio Album",
  title: "A Love Supreme",
  description: "by John Coltrane",
  asin: "B0000A118M",

  shipping: {
    weight: 6,
    dimensions: {
      width: 10,
      height: 10,
      depth: 1
    },
  },

  pricing: {
    list: 1200,
    retail: 1100,
    savings: 100,
    pct_savings: 8
  },

  details: {
    title: "A Love Supreme [Original Recording Reissued]",
    artist: "John Coltrane",
    genre: [ "Jazz", "General" ],
        ...
    tracks: [
      "A Love Supreme Part I: Acknowledgement",
      "A Love Supreme Part II - Resolution",
      "A Love Supreme, Part III: Pursuance",
      "A Love Supreme, Part IV-Psalm"
    ],
  },
}
```

A movie item would have the same fields for general product information, shipping, and pricing, but have different details sub-document. Consider the following:

```
{
  sku: "00e8da9d",
  type: "Film",
  ...,
  asin: "B000P0J0AQ",

  shipping: { ... },

  pricing: { ... },

  details: {
    title: "The Matrix",
    director: [ "Andy Wachowski", "Larry Wachowski" ],
    writer: [ "Andy Wachowski", "Larry Wachowski" ],
    ...,
    aspect_ratio: "1.66:1"
  },
}
```

Note: In MongoDB, you can have fields that hold multiple values (i.e. arrays) without any restrictions on the number of fields or values (as with `genre_0` and `genre_1`) and also without the need for a JOIN operation.

Operations

For most deployments the primary use of the product catalog is to perform search operations. This section provides an overview of various types of queries that may be useful for supporting an e-commerce site. All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Find Albums by Genre and Sort by Year Produced

Querying This query returns the documents for the products of a specific genre, sorted in reverse chronological order:

```
query = db.products.find({'type':'Audio Album',
                          'details.genre': 'jazz'})
query = query.sort([('details.issue_date', -1)])
```

Indexing To support this query, create a compound index on all the properties used in the filter and in the sort:

```
db.products.ensure_index([
    ('type', 1),
    ('details.genre', 1),
    ('details.issue_date', -1)])
```

Note: The final component of the index is the sort field. This allows MongoDB to traverse the index in the sorted order to preclude a slow in-memory sort.

Find Products Sorted by Percentage Discount Descending

While most searches will be for a particular type of product (e.g album, movie, etc.,) in some situations you may want to return all products in a certain price range, or discount percentage.

Querying To return this data use the pricing information that exists in all products to find the products with the highest percentage discount:

```
query = db.products.find( { 'pricing.pct_savings': {'$gt': 25 })
query = query.sort([('pricing.pct_savings', -1)])
```

Indexing To support this type of query, you will want to create an index on the `pricing.pct_savings` field:

```
db.products.ensure_index('pricing.pct_savings')
```

Since MongoDB can read indexes in ascending or descending order, the order of the index does not matter.

Note: If you want to preform range queries (e.g. "return all products over $25") and then sort by another property like `pricing.retail`, MongoDB cannot use the index as effectively in this situation.

The field that you want to select a range, or perform sort operations, must be the *last* field in a compound index in order to avoid scanning an entire collection. Using different properties within a single combined range query and sort operation requires some scanning which will limit the speed of your query.

Find Movies Based on Staring Actor

Querying Use the following query to select documents within the details of a specified product type (i.e. `Film`) of product (a movie) to find products that contain a certain value (i.e. a specific actor in the `details.actor` field,) with the results sorted by date descending:

```
query = db.products.find({'type': 'Film',
                          'details.actor': 'Keanu Reeves'})
query = query.sort([('details.issue_date', -1)])
```

Indexing To support this query, you may want to create the following index.

```
db.products.ensure_index([
    ('type', 1),
    ('details.actor', 1),
    ('details.issue_date', -1)])
```

This index begins with the `type` field and then narrows by the other search field, where the final component of the index is the sort field to maximize index efficiency.

Find Movies with a Particular Word in the Title

Regardless of database engine, in order to retrieve this information the system will need to scan some number of documents or records to satisfy this query.

Querying MongoDB supports regular expressions within queries. In Python, you can use the "`re`" module to construct the query:

```
import re
re_hacker = re.compile(r'.*hacker.*', re.IGNORECASE)

query = db.products.find({'type': 'Film', 'title': re_hacker})
query = query.sort([('details.issue_date', -1)])
```

MongoDB provides a special syntax for regular expression queries without the need for the `re` module. Consider the following alternative which is equivalent to the above example:

```
query = db.products.find({
    'type': 'Film',
    'title': {'$regex': '.*hacker.*', '$options':'i'}})
query = query.sort([('details.issue_date', -1)])
```

The `$options` operator specifies a case insensitive match.

Indexing The indexing strategy for these kinds of queries is different from previous attempts. Here, create an index on `{ type: 1, details.issue_date: -1, title: 1 }` using the following command at the Python/PyMongo console:

```
db.products.ensure_index([
    ('type', 1),
    ('details.issue_date', -1),
    ('title', 1)])
```

This index makes it possible to avoid scanning whole documents by using the index for scanning the title rather than forcing MongoDB to scan whole documents for the title field. Additionally, to support the sort on the `details.issue_date` field, by placing this field *before* the `title` field, ensures that the result set is already ordered before MongoDB filters title field.

Scaling

Sharding

Database performance for these kinds of deployments are dependent on indexes. You may use *sharding* to enhance performance by allowing MongoDB to keep larger portions of those indexes in RAM. In sharded configurations, select a *shard key* that allows `mongos` to route queries directly to a single shard or small group of shards.

Since most of the queries in this system include the `type` field, include this in the shard key. Beyond this, the remainder of the shard key is difficult to predict without information about your database's actual activity and distribution. Consider that:

- `details.issue_date` would be a poor addition to the shard key because, although it appears in a number of queries, no query was were *selective* by this field.
- you should include one or more fields in the `detail` document that you query frequently, and a field that has quasi-random features, to prevent large unsplitable chunks.

In the following example, assume that the `details.genre` field is the second-most queried field after `type`. Enable sharding using the following `shardCollection` operation at the Python/PyMongo console:

```
>>> db.command('shardCollection', 'product', {
...     key : { 'type': 1, 'details.genre' : 1, 'sku':1 } })
{ "collectionsharded" : "details.genre", "ok" : 1 }
```

Note: Even if you choose a "poor" shard key that requires `mongos` to broadcast all to all shards, you will still see some benefits from sharding, because:

1. Sharding makes a larger amount of memory available to store indexes, and
2. MongoDB will parallelize queries across shards, reducing latency.

Read Preference

While *sharding* is the best way to scale operations, some data sets make it impossible to partition data so that `mongos` can route queries to specific shards. In these situations `mongos` sends the query to all shards and then combines the results before returning to the client.

In these situations, you can add additional read performance by allowing `mongos` to read from the *secondary* instances in a *replica set* by configuring *read preference* in your client. Read preference is configurable on a per-connection or per-operation basis. In PyMongo, set the `read_preference` argument.

The `SECONDARY` property in the following example, permits reads from a *secondary* (as well as a primary) for the entire connection .

```
conn = pymongo.Connection(read_preference=pymongo.SECONDARY)
```

Conversely, the `SECONDARY_ONLY` read preference means that the client will only send read operation only to the secondary member

```
conn = pymongo.Connection(read_preference=pymongo.SECONDARY_ONLY)
```

You can also specify `read_preference` for specific queries, as follows:

```
results = db.product.find(..., read_preference=pymongo.SECONDARY)
```

or

```
results = db.product.find(..., read_preference=pymongo.SECONDARY_ONLY)
```

See Also:

"*Replica Set Read Preference*."

3.2 Inventory Management

Overview

This case study provides an overview of practices and patterns for designing and developing the inventory management portions of an E-commerce application.

See Also:

"*Product Catalog* (page xxx)."

Problem

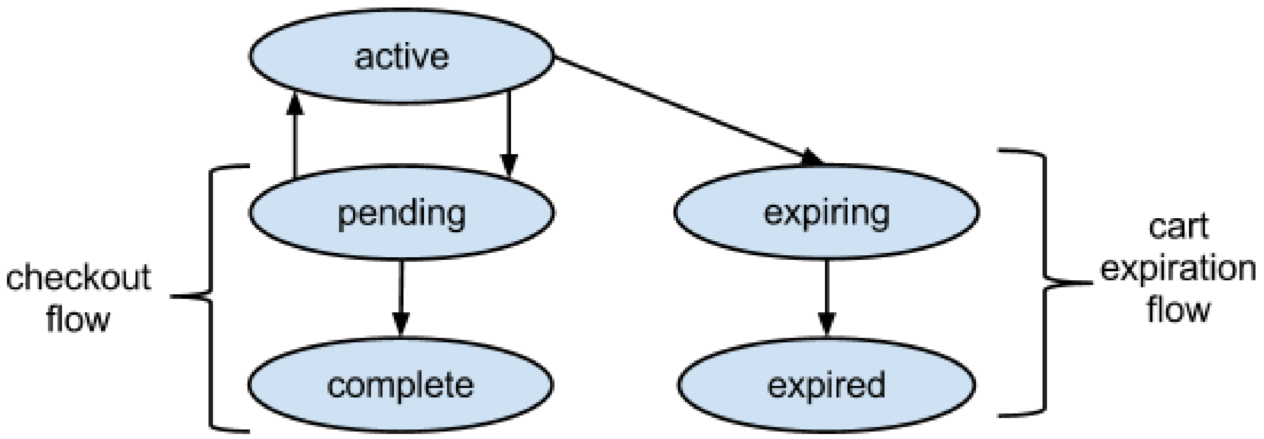
Customers in e-commerce stores regularly add and remove items from their "shopping cart," change quantities multiple times, abandon the cart at any point, and sometimes have problems during and after checkout that require a hold or canceled order. These activities make it difficult to maintain inventory systems and counts and ensure that customers cannot "buy" items that are unavailable while they shop in your store.

Solution

This solution keeps the traditional metaphor of the shopping cart, but the shopping cart will *age*. After a shopping cart has been inactive for a certain period of time, all items in the cart re-enter the available inventory and the cart is empty. The state transition diagram for a shopping cart is below:

Schema

Inventory collections must maintain counts of the current available inventory of each stock-keeping unit (SKU; or item) as well as a list of items in carts that may return to the available inventory if they are in a shopping cart that times out. In the following example, the `_id` field stores the SKU:



```
{
    _id: '00e8da9b',
    qty: 16,
    carted: [
        { qty: 1, cart_id: 42,
          timestamp: ISODate("2012-03-09T20:55:36Z"), },
        { qty: 2, cart_id: 43,
          timestamp: ISODate("2012-03-09T21:55:36Z"), },
    ]
}
```

Note: These examples use a simplified schema. In a production implementation, you may choose to merge this schema with the product catalog schema described in the "*Product Catalog* (page xxx)" document.

The SKU above has 16 items in stock, 1 item a cart, and 2 items in a second cart. This leaves a total of 19 unsold items of merchandise.

To model the shopping cart objects, you need to maintain `sku`, `quantity`, fields embedded in a shopping cart *document*:

```
{
    _id: 42,
    last_modified: ISODate("2012-03-09T20:55:36Z"),
    status: 'active',
    items: [
        { sku: '00e8da9b', qty: 1, item_details: {...} },
        { sku: '0ab42f88', qty: 4, item_details: {...} }
    ]
}
```

Note: The `item_details` field in each line item allows your application to display the cart contents to the user without requiring a second query to fetch details from the catalog collection.

Operations

This section introduces operations that you may use to support an e-commerce site. All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Add an Item to a Shopping Cart

Moving an item from the available inventory to a cart is a fundamental requirement for a shopping cart system. The most important requirement is to ensure that your application will never move an unavailable item from the inventory to the cart.

Ensure that inventory is only updated if there is sufficient inventory to satisfy the request with the following `add_item_to_cart` function operation.

```
def add_item_to_cart(cart_id, sku, qty, details):
    now = datetime.utcnow()

    # Make sure the cart is still active and add the line item
    result = db.cart.update(
        {'_id': cart_id, 'status': 'active' },
        { '$set': { 'last_modified': now },
          '$push': {
              'items': {'sku': sku, 'qty':qty, 'details': details } } },
        w=1)
    if not result['updatedExisting']:
        raise CartInactive()

    # Update the inventory
    result = db.inventory.update(
        {'_id':sku, 'qty': {'$gte': qty}},
        {'$inc': {'qty': -qty},
         '$push': {
             'carted': { 'qty': qty, 'cart_id':cart_id,
                         'timestamp': now } } },
        w=1)
    if not result['updatedExisting']:
        # Roll back our cart update
        db.cart.update(
            {'_id': cart_id },
            { '$pull': { 'items': {'sku': sku } } })
        raise InadequateInventory()
```

The system does not trust that the available inventory can satisfy a request

First this operation checks to make sure that the cart is "active" before adding a item. Then, it verifies that the available inventory to satisfy the request before decrementing inventory.

If there is not adequate inventory, the system removes the cart update: by specifying `w=1` and checking the result allows the application to report an error if the cart is inactive or available quantity is insufficient to satisfy the request.

Note: This operation requires no *indexes* beyond the default index on the `_id` field.

Modifying the Quantity in the Cart

The following process underlies adjusting the quantity of items in a users cart. The application must ensure that when a user increases the quantity of an item, in addition to updating the `carted` entry for the user's cart, that the inventory exists to cover the modification.

```python
def update_quantity(cart_id, sku, old_qty, new_qty):
    now = datetime.utcnow()
    delta_qty = new_qty - old_qty

    # Make sure the cart is still active and add the line item
    result = db.cart.update(
        {'_id': cart_id, 'status': 'active', 'items.sku': sku },
        {'$set': {
             'last_modified': now,
             'items.$.qty': new_qty },
        },
        w=1)
    if not result['updatedExisting']:
        raise CartInactive()

    # Update the inventory
    result = db.inventory.update(
        {'_id':sku,
         'carted.cart_id': cart_id,
         'qty': {'$gte': delta_qty} },
        {'$inc': {'qty': -delta_qty },
         '$set': { 'carted.$.qty': new_qty, 'timestamp': now } },
        w=1)
    if not result['updatedExisting']:
        # Roll back our cart update
        db.cart.update(
            {'_id': cart_id, 'items.sku': sku },
            {'$set': { 'items.$.qty': old_qty } })
        raise InadequateInventory()
```

Note: That the positional operator `$` updates the particular `carted` entry and item that matched the query.

This allows the application to update the inventory and keep track of the data needed to "rollback" the cart in a single atomic operation. The code also ensures that the cart is active.

Note: This operation requires no *indexes* beyond the default index on the `_id` field.

Checking Out

The checkout operation must: validate the method of payment and remove the `carted` items after the transaction succeeds. Consider the following procedure:

```python
def checkout(cart_id):
    now = datetime.utcnow()

    # Make sure the cart is still active and set to 'pending'. Also
    #     fetch the cart details so we can calculate the checkout price
    cart = db.cart.find_and_modify(
        {'_id': cart_id, 'status': 'active' },
        update={'$set': { 'status': 'pending','last_modified': now } } )
    if cart is None:
        raise CartInactive()

    # Validate payment details; collect payment
    try:
        collect_payment(cart)
        db.cart.update(
            {'_id': cart_id },
            {'$set': { 'status': 'complete' } } )
        db.inventory.update(
            {'carted.cart_id': cart_id},
            {'$pull': {'cart_id': cart_id} },
            multi=True)
    except:
        db.cart.update(
            {'_id': cart_id },
            {'$set': { 'status': 'active' } } )
        raise
```

Begin by "locking" the cart by setting its status to "pending" Then the system will verify that the cart is still active and collect payment data. Then, the `findAndModify` *command* makes it possible to update the cart atomically and return its details to capture payment information. Then:

- If the payment is successful, then the application will remove the `carted` items from the inventory documents and set the cart to `complete`.
- If payment is unsuccessful, the application will unlock the cart by setting its status to `active` and report a payment error.

Note: This operation requires no *indexes* beyond the default index on the `_id` field.

Returning Inventory from Timed-Out Carts

Process Periodically, your application must "expire" inactive carts and return their items to available inventory. In the example that follows the variable `timeout` controls the length of time before a cart expires:

```
def expire_carts(timeout):
    now = datetime.utcnow()
    threshold = now - timedelta(seconds=timeout)

    # Lock and find all the expiring carts
    db.cart.update(
        {'status': 'active', 'last_modified': { '$lt': threshold } },
        {'$set': { 'status': 'expiring' } },
        multi=True )

    # Actually expire each cart
    for cart in db.cart.find({'status': 'expiring'}):

        # Return all line items to inventory
        for item in cart['items']:
            db.inventory.update(
                { '_id': item['sku'],
                  'carted.cart_id': cart['id'],
                  'carted.qty': item['qty']
                },
                {'$inc': { 'qty': item['qty'] },
                 '$pull': { 'carted': { 'cart_id': cart['id'] } } })

        db.cart.update(
            {'_id': cart['id'] },
            {'$set': { status': 'expired' })
```

This procedure:

1. finds all carts that are older than the `threshold` and are due for expiration.
2. for each "expiring" cart, return all items to the available inventory.
3. once the items return to the available inventory, set the `status` field to `expired`.

Indexing To support returning inventory from timed-out cart, create an index to support queries on their `status` and `last_modified` fields. Use the following operations in the Python/PyMongo shell:

```
db.cart.ensure_index([('status', 1), ('last_modified', 1)])
```

Error Handling

The above operations do not account for one possible failure situation: if an exception occurs after updating the shopping cart but before updating the inventory collection. This would result in a shopping cart that may be absent or expired but items have not returned to available inventory.

To account for this case, your application will need a periodic cleanup operation that finds inventory items that have `carted` items and check that to ensure that they exist in a user's cart, and return them to available inventory if they do not.

```
def cleanup_inventory(timeout):
    now = datetime.utcnow()
    threshold = now - timedelta(seconds=timeout)

    # Find all the expiring carted items
    for item in db.inventory.find(
        {'carted.timestamp': {'$lt': threshold }}):

        # Find all the carted items that matched
        carted = dict(
                  (carted_item['cart_id'], carted_item)
                  for carted_item in item['carted']
                  if carted_item['timestamp'] < threshold)

        # First Pass: Find any carts that are active and refresh the carted items
        for cart in db.cart.find(
            { '_id': {'$in': carted.keys() },
            'status':'active'}):
            cart = carted[cart['_id']]

            db.inventory.update(
                { '_id': item['_id'],
                  'carted.cart_id': cart['_id'] },
                { '$set': {'carted.$.timestamp': now } })
            del carted[cart['_id']]

        # Second Pass: All the carted items left in the dict need to now be
        #    returned to inventory
        for cart_id, carted_item in carted.items():
            db.inventory.update(
                { '_id': item['_id'],
                  'carted.cart_id': cart_id,
                  'carted.qty': carted_item['qty'] },
                { '$inc': { 'qty': carted_item['qty'] },
                  '$pull': { 'carted': { 'cart_id': cart_id } } })
```

To summarize: This operation finds all "carted" items that have time stamps older than the threshold. Then, the process makes two passes over these items:

1. Of the items with time stamps older than the threshold, if the cart is still active, it resets the time stamp to maintain the carts.
2. Of the stale items that remain in inactive carts, the operation returns these items to the inventory.

Note: The function above is safe for use because it checks to ensure that the cart has expired before returning items from the cart to inventory. However, it could be long-running and slow other updates and queries.

Use judiciously.

Sharding

If you need to *shard* the data for this system, the `_id` field is an ideal *shard key* for both carts and products because most update operations use the `_id` field. This allows `mongos` to route all updates that select on `_id` to a single `mongod` process.

There are two drawbacks for using `_id` as a shard key:

- If the cart collection's `_id` is an incrementing value, all new carts end up on a single shard.

  You can mitigate this effect by choosing a random value upon the creation of a cart, such as a hash (i.e. MD5 or SHA-1) of an ObjectID, as the `_id`. The process for this operation would resemble the following:

  ```
  import hashlib
  import bson

  cart_id = bson.ObjectId()
  cart_id_hash = hashlib.md5(str(cart_id)).hexdigest()

  cart = { "_id": cart_id, "cart_hash": cart_id_hash }
  db.cart.insert(cart)
  ```

- Cart expiration and inventory adjustment requires update operations and queries to broadcast to all shards when using `_id` as a shard key.

  This may be less relevant as the expiration functions run relatively infrequently and you can queue them or artificially slow them down (as with judicious use of `sleep()`) to minimize server load.

Use the following commands in the Python/PyMongo console to shard the cart and inventory collections:

```
>>> db.command('shardCollection', 'inventory'
...             'key': { '_id': 1 } )
{ "collectionsharded" : "inventory", "ok" : 1 }
>>> db.command('shardCollection', 'cart')
...             'key': { '_id': 1 } )
{ "collectionsharded" : "cart", "ok" : 1 }
```

3.3 Category Hierarchy

Overview

This document provides the basic design for modeling a product hierarchy stored in MongoDB as well as a collection of common operations for interacting with this data that will help you begin to write an E-commerce product category hierarchy.

See Also:

"*Product Catalog* (page xxx)"

Solution

To model a product category hierarchy, this solution keeps each category in its own document that also has a list of its ancestors or "parents." This document uses music genres as the basis of its examples:

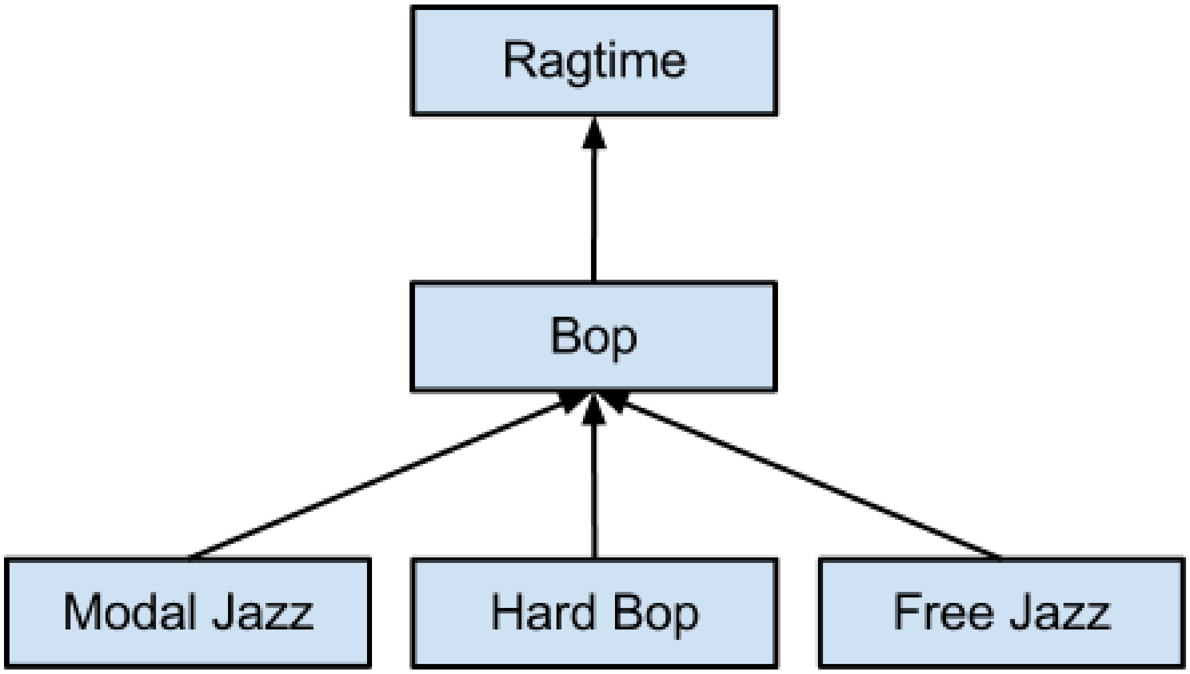


Figure 4: Initial category hierarchy

Because these kinds of categories change infrequently, this model focuses on the operations needed to keep the hierarchy up-to-date rather than the performance profile of update operations.

Schema

This schema has the following properties:

- A single document represents each category in the hierarchy.
- An `ObjectId` identifies each category document for internal cross-referencing.
- Each category document has a human-readable name and a URL compatible `slug` field.

- The schema stores a list of ancestors for each category to facilitate displaying a query and its ancestors using only a single query.

Consider the following prototype:

```
{ "_id" : ObjectId("4f5ec858eb03303a11000002"),
  "name" : "Modal Jazz",
  "parent" : ObjectId("4f5ec858eb03303a11000001"),
  "slug" : "modal-jazz",
  "ancestors" : [
         { "_id" : ObjectId("4f5ec858eb03303a11000001"),
        "slug" : "bop",
        "name" : "Bop" },
         { "_id" : ObjectId("4f5ec858eb03303a11000000"),
           "slug" : "ragtime",
           "name" : "Ragtime" } ]
}
```

Operations

This section outlines the category hierarchy manipulations that you may need in an E-Commerce site. All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Read and Display a Category

Querying Use the following option to read and display a category hierarchy. This query will use the slug field to return the category information and a "bread crumb" trail from the current category to the top level category.

```
category = db.categories.find(
    {'slug':slug},
    {'_id':0, 'name':1, 'ancestors.slug':1, 'ancestors.name':1 })
```

Indexing Create a unique index on the slug field with the following operation on the Python/PyMongo console:

```
>>> db.categories.ensure_index('slug', unique=True)
```

Add a Category to the Hierarchy

To add a category you must first determine its ancestors. Take adding a new category "Swing" as a child of "Ragtime", as below:

The insert operation would be trivial except for the ancestors. To define this array, consider the following helper function:

```
def build_ancestors(_id, parent_id):
    parent = db.categories.find_one(
        {'_id': parent_id},
        {'name': 1, 'slug': 1, 'ancestors':1})
    parent_ancestors = parent.pop('ancestors')
    ancestors = [ parent ] + parent_ancestors
    db.categories.update(
        {'_id': _id},
        {'$set': { 'ancestors': ancestors } })
```

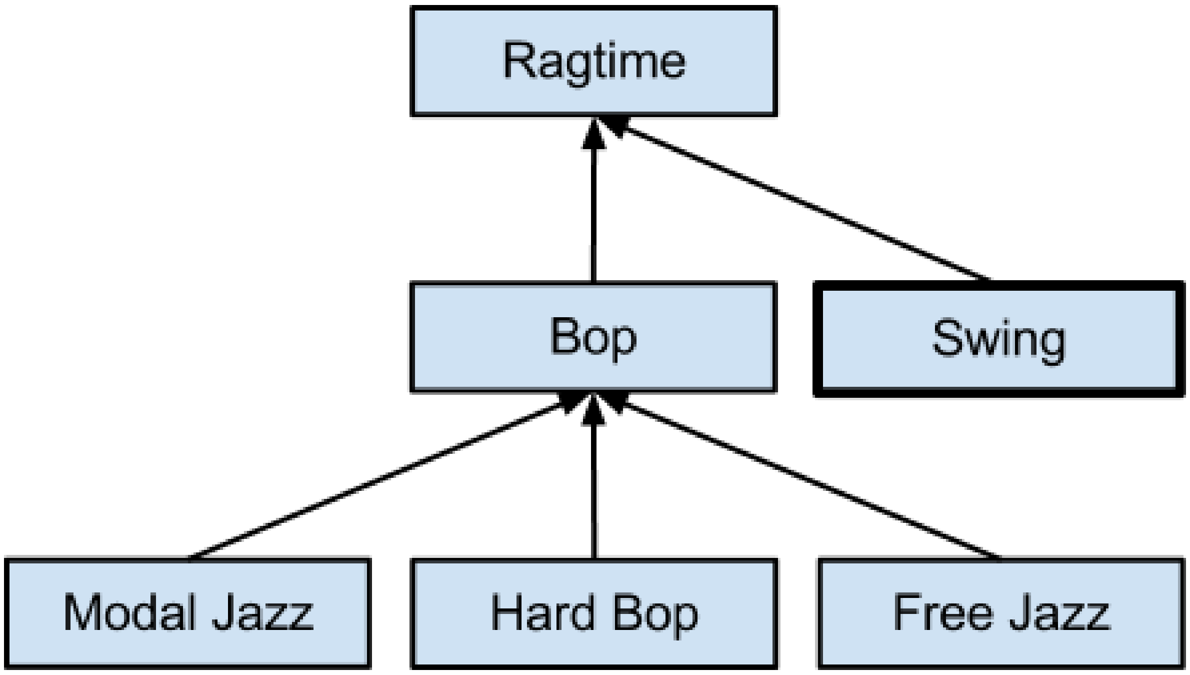


Figure 5: Adding a category

You only need to travel "up" one level in the hierarchy to get the ancestor list for "Ragtime" that you can use to build the ancestor list for "Swing." Then create a document with the following set of operations:

```
doc = dict(name='Swing', slug='swing', parent=ragtime_id)
swing_id = db.categories.insert(doc)
build_ancestors(swing_id, ragtime_id)
```

Note: Since these queries and updates all selected based on `_id`, you only need the default MongoDB-supplied index on `_id` to support this operation efficiently.

Change the Ancestry of a Category

This section address the process for reorganizing the hierarchy by moving "bop" under "swing" as follows:

Procedure Update the `bop` document to reflect the change in ancestry with the following operation:

```
db.categories.update(
    {'_id':bop_id}, {'$set': { 'parent': swing_id } } )
```

The following helper function, rebuilds the ancestor fields to ensure correctness. [7]

```
def build_ancestors_full(_id, parent_id):
    ancestors = []
    while parent_id is not None:
        parent = db.categories.find_one(
            {'_id': parent_id},
```

[7] Your application cannot guarantee that the ancestor list of a parent category is correct, because MongoDB may process the categories out-of-order.

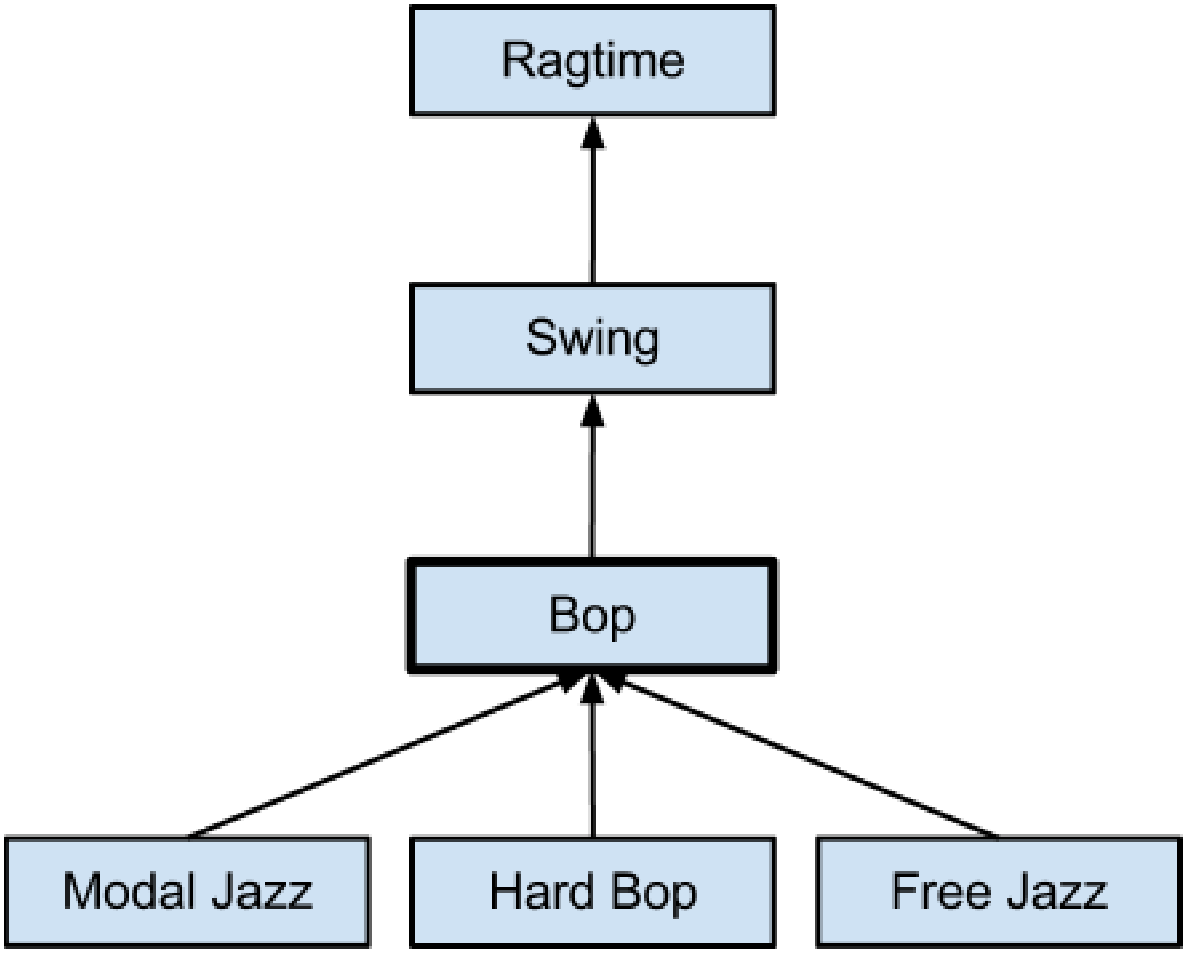


Figure 6: Change the parent of a category

```
                {'parent': 1, 'name': 1, 'slug': 1, 'ancestors':1})
            parent_id = parent.pop('parent')
            ancestors.append(parent)
        db.categories.update(
            {'_id': _id},
            {'$set': { 'ancestors': ancestors } })
```

You can use the following loop to reconstruct all the descendants of the "bop" category:

```
for cat in db.categories.find(
    {'ancestors._id': bop_id},
    {'parent_id': 1}):
    build_ancestors_full(cat['_id'], cat['parent_id'])
```

Indexing Create an index on the `ancestors._id` field to support the update operation.

```
db.categories.ensure_index('ancestors._id')
```

Rename a Category

To a rename a category you need to both update the category itself and also update all the descendants. Consider renaming "Bop" to "BeBop" as in the following figure:

First, you need to update the category name with the following operation:

```
db.categories.update(
    {'_id':bop_id}, {'$set': { 'name': 'BeBop' } } )
```

Next, you need to update each descendant's ancestors list:

```
db.categories.update(
    {'ancestors._id': bop_id},
    {'$set': { 'ancestors.$.name': 'BeBop' } },
    multi=True)
```

This operation uses:

- the positional operation `$` to match the exact "ancestor" entry that matches the query, and
- the `multi` option to update all documents that match this query.

Note: In this case, the index you have already defined on `ancestors._id` is sufficient to ensure good performance.

Sharding

For most deployments, *sharding* this collection has limited value because the collection will be very small. If you do need to shard, because most updates query the `_id` field, this field is a suitable *shard key*. Shard the collection with the following operation in the Python/PyMongo console.

```
>>> db.command('shardCollection', 'categories', {
...     'key': {'_id': 1} })
{ "collectionsharded" : "categories", "ok" : 1 }
```

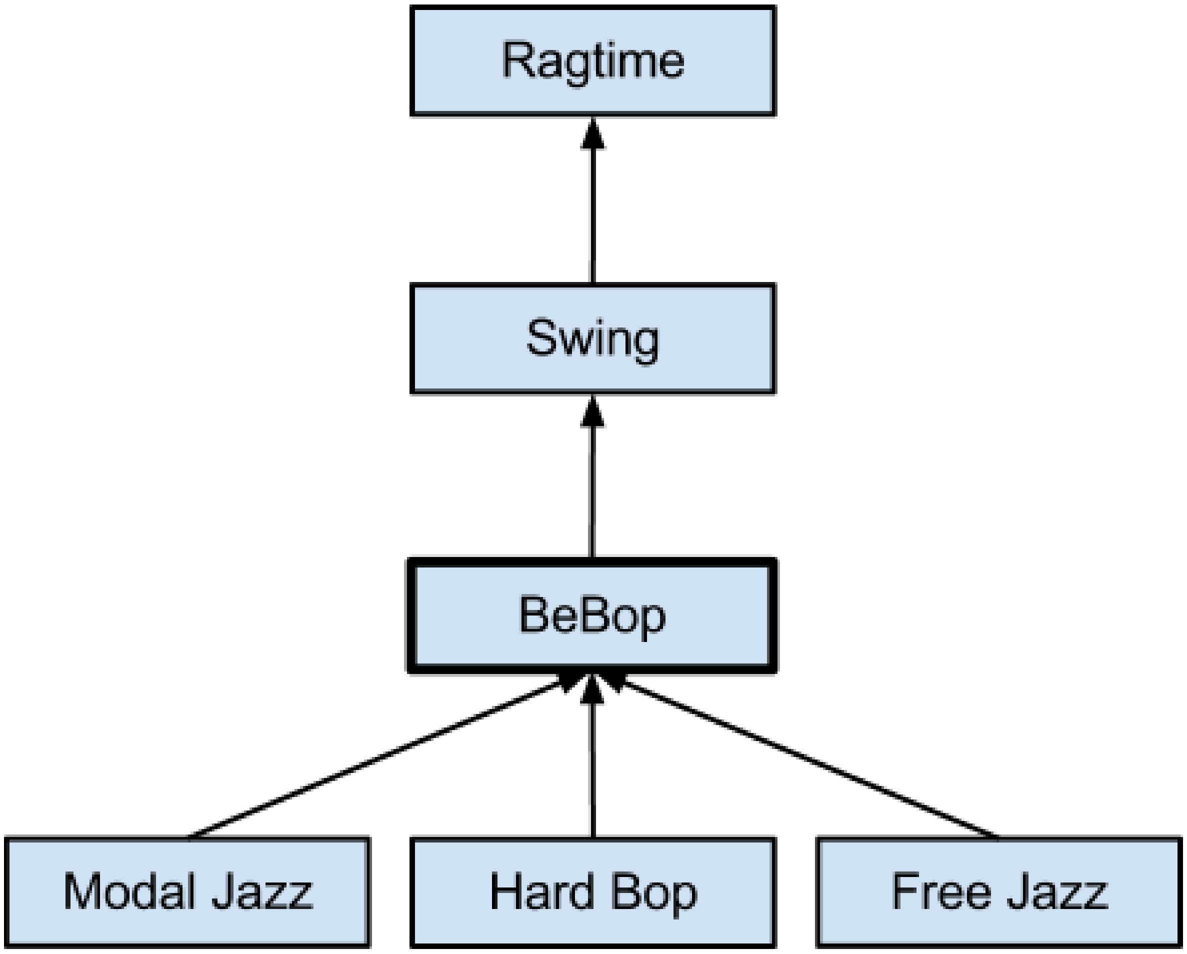


Figure 7: Rename a category

4 Content Management Systems

The content management use cases introduce fundamental MongoDB practices and approaches, using familiar problems and simple examples. The "*Metadata and Asset Management* (page l)" document introduces a model that you may use when designing a web site content management system, while "*Storing Comments* (page lvi)" introduces the method for modeling user comments on content, like blog posts, and media, in MongoDB.

4.1 Metadata and Asset Management

Overview

This document describes the design and pattern of a content management system using MongoDB modeled on the popular Drupal CMS.

Problem

You are designing a content management system (CMS) and you want to use MongoDB to store the content of your sites.

Solution

To build this system you will use MongoDB's flexible schema to store all content "nodes" in a single collection regardless of type. This guide will provide prototype schema and describe common operations for the following primary node types:

Basic Page Basic pages are useful for displaying infrequently-changing text such as an 'about' page. With a basic page, the salient information is the title and the content.

Blog entry Blog entries record a "stream" of posts from users on the CMS and store title, author, content, and date as relevant information.

Photo Photos participate in photo galleries, and store title, description, author, and date along with the actual photo binary data.

This solution does not describe schema or process for storing or using navigational and organizational information.

Schema

Although *documents* in the `nodes` collection contain content of different times, all documents have a similar structure and a set of common fields. Consider the following prototype document for a "basic page" node type:

```
{
    _id: ObjectId(...),
    nonce: ObjectId(...),
    metadata: {
        type: 'basic-page'
        section: 'my-photos',
        slug: 'about',
        title: 'About Us',
        created: ISODate(...),
        author: { _id: ObjectId(...), name: 'Rick' },
        tags: [ ... ],
        detail: { text: '# About Us\n...' }
    }
}
```

Most fields are descriptively titled. The `section` field identifies groupings of items, as in a photo gallery, or a particular blog . The `slug` field holds a URL-friendly unique representation of the node, usually that is unique within its section for generating URLs.

All documents also have a `detail` field that varies with the document type. For the basic page above, the detail field might hold the text of the page. For a blog entry, the `detail` field might hold a sub-document. Consider the following prototype:

```
{
    ...
    metadata: {
        ...
        type: 'blog-entry',
        section: 'my-blog',
        slug: '2012-03-noticed-the-news',
        ...
        detail: {
            publish_on: ISODate(...),
            text: 'I noticed the news from Washington today...'
        }
    }
}
```

Photos require a different approach. Because photos can be potentially larger than these documents, it's important to separate the binary photo storage from the nodes metadata.

GridFS provides the ability to store larger files in MongoDB. GridFS stores data in two collections, in this case, `cms.assets.files`, which stores metadata, and `cms.assets.chunks` which stores the data itself. Consider the following prototype document from the `cms.assets.files` collection:

```
{
    _id: ObjectId(...),
    length: 123...,
    chunkSize: 262144,
    uploadDate: ISODate(...),
    contentType: 'image/jpeg',
    md5: 'ba49a...',
    metadata: {
        nonce: ObjectId(...),
        slug: '2012-03-invisible-bicycle',
        type: 'photo',
        section: 'my-album',
        title: 'Kitteh',
        created: ISODate(...),
        author: { _id: ObjectId(...), name: 'Jared' },
        tags: [ ... ],
        detail: {
            filename: 'kitteh_invisible_bike.jpg',
            resolution: [ 1600, 1600 ], ... }
    }
}
```

Note: This document embeds the basic node document fields, which allows you to use the same code to manipulate nodes, regardless of type.

Operations

This section outlines a number of common operations for building and interacting with the metadata and asset layer of the cms for all node types. All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Create and Edit Content Nodes

Procedure The most common operations inside of a CMS center on creating and editing content. Consider the following `insert()` operation:

```
db.cms.nodes.insert({
    'nonce': ObjectId(),
    'metadata': {
        'section': 'myblog',
        'slug': '2012-03-noticed-the-news',
        'type': 'blog-entry',
        'title': 'Noticed in the News',
        'created': datetime.utcnow(),
        'author': { 'id': user_id, 'name': 'Rick' },
        'tags': [ 'news', 'musings' ],
        'detail': {
            'publish_on': datetime.utcnow(),
            'text': 'I noticed the news from Washington today...' }
        }
     })
```

Once inserted, your application must have some way of preventing multiple concurrent updates. The schema uses the special `nonce` field to help detect concurrent edits. By using the `nonce` field in the query portion of the `update` operation, the application will generate an error if there is an editing collision. Consider the following `update`

```
def update_text(section, slug, nonce, text):
    result = db.cms.nodes.update(
        { 'metadata.section': section,
          'metadata.slug': slug,
          'nonce': nonce },
        { '$set':{'metadata.detail.text': text, 'nonce': ObjectId() } },
        w=1)
    if not result['updatedExisting']:
        raise ConflictError()
```

You may also want to perform metadata edits to the item such as adding tags:

```
db.cms.nodes.update(
    { 'metadata.section': section, 'metadata.slug': slug },
    { '$addToSet': { 'tags': { '$each': [ 'interesting', 'funny' ] } } })
```

In this example the `$addToSet` operator will only add values to the `tags` field if they do not already exist in the `tags` array, there's no need to supply or update the nonce.

Index Support To support updates and queries on the `metadata.section`, and `metadata.slug`, fields *and* to ensure that two editors don't create two documents with the same section name or slug. Use the following operation at the Python/PyMongo console:

```
>>> db.cms.nodes.ensure_index([
...     ('metadata.section', 1), ('metadata.slug', 1)], unique=True)
```

The unique=True option prevents to documents from colliding. If you want an index to support queries on the above fields and the nonce field create the following index:

```
>>> db.cms.nodes.ensure_index([
...     ('metadata.section', 1), ('metadata.slug', 1), ('nonce', 1) ])
```

However, in most cases, the first index will be sufficient to support these operations.

Upload a Photo

Procedure To update a photo object, use the following operation, which builds upon the basic update procedure:

```
def upload_new_photo(
    input_file, section, slug, title, author, tags, details):
    fs = GridFS(db, 'cms.assets')
    with fs.new_file(
        content_type='image/jpeg',
        metadata=dict(
            type='photo',
            locked=datetime.utcnow(),
            section=section,
            slug=slug,
            title=title,
            created=datetime.utcnow(),
            author=author,
            tags=tags,
            detail=detail)) as upload_file:
        while True:
            chunk = input_file.read(upload_file.chunk_size)
            if not chunk: break
            upload_file.write(chunk)
    # unlock the file
    db.assets.files.update(
        {'_id': upload_file._id},
        {'$set': { 'locked': None } } )
```

Because uploading the photo spans multiple documents and is a non-atomic operation, you must "lock" the file during upload by writing datetime.utcnow() in the record. This helps when there are multiple concurrent editors and lets the application detect stalled file uploads. This operation assumes that, for photo upload, the last update will succeed:

```
def update_photo_content(input_file, section, slug):
    fs = GridFS(db, 'cms.assets')

    # Delete the old version if it's unlocked or was locked more than 5
    #    minutes ago
    file_obj = db.cms.assets.find_one(
        { 'metadata.section': section,
          'metadata.slug': slug,
          'metadata.locked': None })
    if file_obj is None:
        threshold = datetime.utcnow() - timedelta(seconds=300)
        file_obj = db.cms.assets.find_one(
            { 'metadata.section': section,
              'metadata.slug': slug,
              'metadata.locked': { '$lt': threshold } })
    if file_obj is None: raise FileDoesNotExist()
    fs.delete(file_obj['_id'])
    # update content, keep metadata unchanged
    file_obj['locked'] = datetime.utcnow()
    with fs.new_file(**file_obj):
        while True:
            chunk = input_file.read(upload_file.chunk_size)
            if not chunk: break
            upload_file.write(chunk)
    # unlock the file
    db.assets.files.update(
        {'_id': upload_file._id},
        {'$set': { 'locked': None } } )
```

As with the basic operations, you can use a much more simple operation to edit the tags:

```
db.cms.assets.files.update(
    { 'metadata.section': section, 'metadata.slug': slug },
    { '$addToSet': { 'metadata.tags': { '$each': [ 'interesting', 'funny' ] } } })
```

Index Support Create a unique index on `{ metadata.section: 1, metadata.slug: 1 }` to support the above operations and prevent users from creating or updating the same file concurrently. Use the following operation in the Python/PyMongo console:

```
>>> db.cms.assets.files.ensure_index([
...     ('metadata.section', 1), ('metadata.slug', 1)], unique=True)
```

Locate and Render a Node

To locate a node based on the value of `metadata.section` and `metadata.slug`, use the following `find_one` operation.

```
node = db.nodes.find_one({'metadata.section': section, 'metadata.slug': slug })
```

Note: The index defined (`section`, `slug`) created to support the update operation, is sufficient to support this operation as well.

Locate and Render a Photo

To locate an image based on the value of `metadata.section` and `metadata.slug`, use the following `find_one` operation.

```
fs = GridFS(db, 'cms.assets')
with fs.get_version({'metadata.section': section, 'metadata.slug': slug }) as img_fpo:
     # do something with the image file
```

Note: The index defined (`section`, `slug`) created to support the update operation, is sufficient to support this operation as well.

Search for Nodes by Tag

Querying To retrieve a list of nodes based on their tags, use the following query:

```
nodes = db.nodes.find({'metadata.tags': tag })
```

Indexing Create an index on the `tags` field in the `cms.nodes` collection, to support this query:

```
>>> db.cms.nodes.ensure_index('tags')
```

Search for Images by Tag

Procedure To retrieve a list of images based on their tags, use the following operation:

```
image_file_objects = db.cms.assets.files.find({'metadata.tags': tag })
fs = GridFS(db, 'cms.assets')
for image_file_object in db.cms.assets.files.find(
    {'metadata.tags': tag }):
    image_file = fs.get(image_file_object['_id'])
    # do something with the image file
```

Indexing Create an index on the `tags` field in the `cms.assets.files` collection, to support this query:

```
>>> db.cms.assets.files.ensure_index('tags')
```

Generate a Feed of Recently Published Blog Articles

Querying Use the following operation to generate a list of recent blog posts sorted in descending order by date, for use on the index page of your site, or in an `.rss` or `.atom` feed.

```
articles = db.nodes.find({
    'metadata.section': 'my-blog'
    'metadata.published': { '$lt': datetime.utcnow() } })
articles = articles.sort({'metadata.published': -1})
```

Note: In many cases you will want to limit the number of nodes returned by this query.

Indexing Create a compound index on the `{ metadata.section: 1, metadata.published: 1 }` fields to support this query and sort operation.

```
>>> db.cms.nodes.ensure_index(
...     [ ('metadata.section', 1), ('metadata.published', -1) ])
```

Note: For all sort or range queries, ensure that field with the sort or range operation is the final field in the index.

Sharding

In a CMS, read performance is more critical than write performance. To achieve the best read performance in a *sharded cluster*, ensure that the `mongos` can route queries to specific *shards*.

Also remember that MongoDB can not enforce unique indexes across shards. Using a compound *shard key* that consists of `metadata.section` and `metadata.slug`, will provide the same semantics as describe above.

Warning: Consider the actual use and workload of your cluster before configuring sharding for your cluster.

Use the following operation at the Python/PyMongo shell:

```
>>> db.command('shardCollection', 'cms.nodes', {
...     key : { 'metadata.section': 1, 'metadata.slug' : 1 } })
{ "collectionsharded": "cms.nodes", "ok": 1}
>>> db.command('shardCollection', 'cms.assets.files', {
...     key : { 'metadata.section': 1, 'metadata.slug' : 1 } })
{ "collectionsharded": "cms.assets.files", "ok": 1}
```

To shard the `cms.assets.chunks` collection, you must use the `_id` field as the *shard key*. The following operation will shard the collection

```
>>> db.command('shardCollection', 'cms.assets.chunks', {
...     key : { 'files_id': 1 } })
{ "collectionsharded": "cms.assets.chunks", "ok": 1}
```

Sharding on the `files_id` field ensures routable queries because all reads from GridFS must first look up the document in `cms.assets.files` and then look up the chunks separately.

4.2 Storing Comments

This document outlines the basic patterns for storing user-submitted comments in a content management system (CMS.)

Overview

MongoDB provides a number of different approaches for storing data like users-comments on content from a CMS. There is no correct implementation, but there are a number of common approaches and known considerations for each approach. This case study explores the implementation details and trade offs of each option. The three basic patterns are:

1. Store each comment in its own *document*.

   This approach provides the greatest flexibility at the expense of some additional application level complexity.

   These implementations make it possible to display comments in chronological or threaded order, and place no restrictions on the number of comments attached to a specific object.

2. Embed all comments in the "parent" document.

   This approach provides the greatest possible performance for displaying comments at the expense of flexibility: the structure of the comments in the document controls the display format.

   Note: Because of the *limit on document size*, documents, including the original content and all comments, cannot grow beyond 16 megabytes.

3. A hybrid design, stores comments separately from the "parent," but aggregates comments into a small number of documents, where each contains many comments.

Also consider that comments can be *threaded*, where comments are always replies to "parent" item or to another comment, which carries certain architectural requirements discussed below.

One Document per Comment

Schema

If you store each comment in its own document, the documents in your `comments` collection, would have the following structure:

```
{
    _id: ObjectId(...),
    discussion_id: ObjectId(...),
    slug: '34db',
    posted: ISODateTime(...),
    author: {
              id: ObjectId(...),
              name: 'Rick'
             },
    text: 'This is so bogus ... '
}
```

This form is only suitable for displaying comments in chronological order. Comments store:

- the `discussion_id` field that references the discussion parent,
- a URL-compatible `slug` identifier,
- a `posted` timestamp,
- an `author` sub-document that contains a reference to a user's profile in the `id` field and their name in the `name` field, and
- the full `text` of the comment.

To support threaded comments, you might use a slightly different structure like the following:

```
{
    _id: ObjectId(...),
    discussion_id: ObjectId(...),
    parent_id: ObjectId(...),
    slug: '34db/8bda'
    full_slug: '2012.02.08.12.21.08:34db/2012.02.09.22.19.16:8bda',
    posted: ISODateTime(...),
    author: {
              id: ObjectId(...),
              name: 'Rick'
             },
    text: 'This is so bogus ... '
}
```

This structure:

- adds a `parent_id` field that stores the contents of the `_id` field of the parent comment,
- modifies the `slug` field to hold a path composed of the parent or parent's slug and this comment's unique slug, and
- adds a `full_slug` field that that combines the slugs and time information to make it easier to sort documents in a threaded discussion by date.

Warning: MongoDB can only index *1024 bytes*. This includes all field data, the field name, and the namespace (i.e. database name and collection name.) This may become an issue when you create an index of the `full_slug` field to support sorting.

Operations

This section contains an overview of common operations for interacting with comments represented using a schema where each comment is its own *document*.

All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose. Issue the following commands at the interactive Python shell to load the required libraries:

```
>>> import bson
>>> import pymongo
```

Post a New Comment To post a new comment in a chronologically ordered (i.e. without threading) system, use the following `insert()` operation:

```
slug = generate_pseudorandom_slug()
db.comments.insert({
    'discussion_id': discussion_id,
    'slug': slug,
    'posted': datetime.utcnow(),
    'author': author_info,
    'text': comment_text })
```

To insert a comment for a system with threaded comments, you must generate the `slug` path and `full_slug` at insert. See the following operation:

```
posted = datetime.utcnow()

# generate the unique portions of the slug and full_slug
slug_part = generate_pseudorandom_slug()
full_slug_part = posted.strftime('%Y.%m.%d.%H.%M.%S') + ':' + slug_part
# load the parent comment (if any)
if parent_slug:
    parent = db.comments.find_one(
        {'discussion_id': discussion_id, 'slug': parent_slug })
    slug = parent['slug'] + '/' + slug_part
    full_slug = parent['full_slug'] + '/' + full_slug_part
else:
    slug = slug_part
    full_slug = full_slug_part

# actually insert the comment
db.comments.insert({
    'discussion_id': discussion_id,
    'slug': slug,
    'full_slug': full_slug,
    'posted': posted,
    'author': author_info,
    'text': comment_text })
```

View Paginated Comments To view comments that are not threaded, select all comments participating in a discussion and sort by the `posted` field. For example:

```
cursor = db.comments.find({'discussion_id': discussion_id})
cursor = cursor.sort('posted')
cursor = cursor.skip(page_num * page_size)
cursor = cursor.limit(page_size)
```

Because the `full_slug` field contains both hierarchical information (via the path) and chronological information, you can use a simple sort on the `full_slug` field to retrieve a threaded view:

```
cursor = db.comments.find({'discussion_id': discussion_id})
cursor = cursor.sort('full_slug')
cursor = cursor.skip(page_num * page_size)
cursor = cursor.limit(page_size)
```

See Also:

`cursor.limit`, `cursor.skip`, and `cursor.sort`

Indexing To support the above queries efficiently, maintain two compound indexes, on:

1. `(``discussion_id, posted)`` and
2. `(``discussion_id, full_slug)``

Issue the following operation at the interactive Python shell.

```
>>> db.comments.ensure_index([
...     ('discussion_id', 1), ('posted', 1)])
>>> db.comments.ensure_index([
...     ('discussion_id', 1), ('full_slug', 1)])
```

Note: Ensure that you always sort by the final element in a compound index to maximize the performance of these queries.

Retrieve Comments via Direct Links

Queries To directly retrieve a comment, without needing to page through all comments, you can select by the `slug` field:

```
comment = db.comments.find_one({
    'discussion_id': discussion_id,
    'slug': comment_slug})
```

You can retrieve a "sub-discussion," or a comment and all of its descendants recursively, by performing a regular expression prefix query on the `full_slug` field:

```
import re

subdiscussion = db.comments.find_one({
    'discussion_id': discussion_id,
    'full_slug': re.compile('^' + re.escape(parent_slug)) })
subdiscussion = subdiscussion.sort('full_slug')
```

Indexing Since you have already created indexes on `{ discussion_id: 1, full_slug: }` to support retrieving sub-discussions, you can add support for the above queries by adding an index on `{ discussion_id: 1 , slug: 1 }`. Use the following operation in the Python shell:

```
>>> db.comments.ensure_index([
...     ('discussion_id', 1), ('slug', 1)])
```

Embedding All Comments

This design embeds the entire discussion of a comment thread inside of the topic *document*. In this example, the "topic," document holds the total content for whatever content you're managing.

Schema

Consider the following prototype `topic` document:

```
{
    _id: ObjectId(...),
    ... lots of topic data ...
    comments: [
        { posted: ISODateTime(...),
          author: { id: ObjectId(...), name: 'Rick' },
          text: 'This is so bogus ... ' },
       ... ]
}
```

This structure is only suitable for a chronological display of all comments because it embeds comments in chronological order. Each document in the array in the `comments` contains the comment's date, author, and text.

Note: Since you're storing the comments in sorted order, there is no need to maintain per-comment slugs.

To support threading using this design, you would need to embed comments within comments, using a structure that resembles the following:

```
{
    _id: ObjectId(...),
    ... lots of topic data ...
    replies: [
        { posted: ISODateTime(...),
          author: { id: ObjectId(...), name: 'Rick' },
          text: 'This is so bogus ... ',
          replies: [
              { author: { ... }, ... },
       ... ]
}
```

Here, the `replies` field in each comment holds the sub-comments, which can intern hold sub-comments.

Note: In the embedded document design, you give up some flexibility regarding display format, because it is difficult to display comments *except* as you store them in MongoDB.

If, in the future, you want to switch from chronological to threaded or from threaded to chronological, this design would make that migration quite expensive.

Warning: Remember that *BSON* documents have a *16 megabyte size limit*. If popular discussions grow larger than 16 megabytes, additional document growth will fail.
Additionally, when MongoDB documents grow significantly after creation you will experience greater storage fragmentation and degraded update performance while MongoDB migrates documents internally.

Operations

This section contains an overview of common operations for interacting with comments represented using a schema that embeds all comments the *document* of the "parent" or topic content.

Note: For all operations below, there is no need for any new indexes since all the operations are function within documents. Because you would retrieve these documents by the `_id` field, you can rely on the index that MongoDB creates automatically.

Post a new comment To post a new comment in a chronologically ordered (i.e unthreaded) system, you need the following `update()`:

```
db.discussion.update(
    { 'discussion_id': discussion_id },
    { '$push': { 'comments': {
        'posted': datetime.utcnow(),
        'author': author_info,
        'text': comment_text } } } )
```

The `$push` operator inserts comments into the `comments` array in correct chronological order. For threaded discussions, the `update()` operation is more complex. To reply to a comment, the following code assumes that it can retrieve the 'path' as a list of positions, for the parent comment:

```
if path != []:
    str_path = '.'.join('replies.%d' % part for part in path)
    str_path += '.replies'
else:
    str_path = 'replies'
db.discussion.update(
    { 'discussion_id': discussion_id },
    { '$push': {
        str_path: {
            'posted': datetime.utcnow(),
            'author': author_info,
            'text': comment_text } } } )
```

This constructs a field name of the form `replies.0.replies.2...` as `str_path` and then uses this value with the `$push` operator to insert the new comment into the parent comment's `replies` array.

View Paginated Comments To view the comments in a non-threaded design, you must use the `$slice` operator:

```
discussion = db.discussion.find_one(
    {'discussion_id': discussion_id},
    { ... some fields relevant to your page from the root discussion ...,
      'comments': { '$slice': [ page_num * page_size, page_size ] }
    })
```

To return paginated comments for the threaded design, you must retrieve the whole document and paginate the comments within the application:

```
discussion = db.discussion.find_one({'discussion_id': discussion_id})

def iter_comments(obj):
    for reply in obj['replies']:
        yield reply
        for subreply in iter_comments(reply):
            yield subreply

paginated_comments = itertools.slice(
    iter_comments(discussion),
    page_size * page_num,
    page_size * (page_num + 1))
```

Retrieve a Comment via Direct Links Instead of retrieving comments via slugs as above, the following example retrieves comments using their position in the comment list or tree.

For chronological (i.e. non-threaded) comments, just use the `$slice` operator to extract a comment, as follows:

```
discussion = db.discussion.find_one(
    {'discussion_id': discussion_id},
    {'comments': { '$slice': [ position, position ] } })
comment = discussion['comments'][0]
```

For threaded comments, you must find the correct path through the tree in your application, as follows:

```
discussion = db.discussion.find_one({'discussion_id': discussion_id})
current = discussion
for part in path:
    current = current.replies[part]
comment = current
```

Note: Since parent comments embed child replies, this operation actually retrieves the entire sub-discussion for the comment you queried for.

See Also:

`find_one()`.

Hybrid Schema Design

Schema

In the "hybrid approach" you will store comments in "buckets" that hold about 100 comments. Consider the following example document:

```
{
    _id: ObjectId(...),
    discussion_id: ObjectId(...),
    page: 1,
    count: 42,
    comments: [ {
        slug: '34db',
        posted: ISODateTime(...),
        author: { id: ObjectId(...), name: 'Rick' },
        text: 'This is so bogus ... ' },
    ... ]
}
```

Each document maintains `page` and `count` data that contains meta data regarding the page, the page number and the comment count, in addition to the `comments` array that holds the comments themselves.

Note: Using a hybrid format makes storing threaded comments complex, and this specific configuration is not covered in this document.

Also, 100 comments is a *soft* limit for the number of comments per page. This value is arbitrary: choose a value that will prevent the maximum document size from growing beyond the 16MB *BSON document size limit*, but large enough to ensure that most comment threads will fit in a single document. In some situations the number of comments per document can exceed 100, but this does not affect the correctness of the pattern.

Operations

This section contains a number of common operations that you may use when building a CMS using this hybrid storage model with documents that hold 100 comment "pages."

All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Post a New Comment

Updating In order to post a new comment, you need to `$push` the comment onto the last page and `$inc` that page's comment count. Consider the following example that queries on the basis of a `discussion_id` field:

```
page = db.comment_pages.find_and_modify(
    { 'discussion_id': discussion['_id'],
      'page': discussion['num_pages'] },
    { '$inc': { 'count': 1 },
      '$push': {
          'comments': { 'slug': slug, ... } } },
    fields={'count':1},
    upsert=True,
    new=True )
```

The `find_and_modify()` operation is an *upsert*,: if MongoDB cannot find a document with the correct `page` number, the `find_and_modify()` will create it and initialize the new document with appropriate values for `count` and `comments`.

To limit the number of comments per page to roughly 100, you will need to create new pages as they become necessary. Add the following logic to support this:

```
if page['count'] > 100:
    db.discussion.update(
        { 'discussion_id: discussion['_id'],
          'num_pages': discussion['num_pages'] },
        { '$inc': { 'num_pages': 1 } } )
```

This `update()` operation includes the last known number of pages in the query to prevent a race condition where the number of pages inciments twice, that would result in a nearly or totally empty document. If another process increments the number of pages, then update above does nothing.

Indexing To support the `find_and_modify()` and `update()` operations, maintain a compound index on (`discussion_id`, `page`) in the `comment_pages` collection, by issuing the following operation at the Python/PyMongo console:

```
>>> db.comment_pages.ensure_index([
...     ('discussion_id', 1), ('page', 1)])
```

View Paginated Comments The following function defines how to paginate comments with a *fixed* page size (i.e. not with the roughly 100 comment documents in the above example,) as en example:

```
def find_comments(discussion_id, skip, limit):
    result = []
    page_query = db.comment_pages.find(
        { 'discussion_id': discussion_id },
        { 'count': 1, 'comments': { '$slice': [ skip, limit ] } })
    page_query = page_query.sort('page')
    for page in page_query:
        result += page['comments']
        skip = max(0, skip - page['count'])
        limit -= len(page['comments'])
        if limit == 0: break
    return result
```

Here, the `$slice` operator pulls out comments from each page, but *only* when this satisfies the `skip` requirement. For example: if you have 3 pages with 100, 102, 101, and 22 comments on each page, and you wish to retrieve comments where `skip=300` and `limit=50`. Use the following algorithm:

Skip	Limit	Discussion
300	50	`{$slice: [ 300, 50 ] }` matches nothing in page #1; subtract page #1's `count` from `skip` and continue.
200	50	`{$slice: [ 200, 50 ] }` matches nothing in page #2; subtract page #2's `count` from `skip` and continue.
98	50	`{$slice: [ 98, 50 ] }` matches 2 comments in page #3; subtract page #3's `count` from `skip` (saturating at 0), subtract 2 from limit, and continue.
0	48	`{$slice: [ 0, 48 ] }` matches all 22 comments in page #4; subtract 22 from `limit` and continue.
0	26	There are no more pages; terminate loop.

Note: Since you already have an index on (`discussion_id`, `page`) in your `comment_pages` collection, MongoDB can satisfy these queries efficiently.

Retrieve a Comment via Direct Links

Query To retrieve a comment directly without paging through all preceding pages of commentary, use the slug to find the correct page, and then use application logic to find the correct comment:

```
page = db.comment_pages.find_one(
    { 'discussion_id': discussion_id,
      'comments.slug': comment_slug},
    { 'comments': 1 })
for comment in page['comments']:
    if comment['slug'] = comment_slug:
        break
```

Indexing To perform this query efficiently you'll need a new index on the `discussion_id` and `comments.slug` fields (i.e. `{ discussion_id: 1 comments.slug: 1 }`.) Create this index using the following operation in the Python/PyMongo console:

```
>>> db.comment_pages.ensure_index([
...     ('discussion_id', 1), ('comments.slug', 1)])
```

Sharding

For all of the architectures discussed above, you will want to the `discussion_id` field to participate in the shard key, if you need to shard your application.

For applications that use the "one document per comment" approach, consider using `slug` (or `full_slug`, in the case of threaded comments) fields in the shard key to allow the `mongos` instances to route requests by `slug`. Issue the following operation at the Python/PyMongo console:

```
>>> db.command('shardCollection', 'comments', {
...      'key' : { 'discussion_id' : 1, 'full_slug': 1 } })
```

This will return the following response:

```
{ "collectionsharded" : "comments", "ok" : 1 }
```

In the case of comments that fully-embedded in parent content *documents* the determination of the shard key is outside of the scope of this document.

For hybrid documents, use the page number of the comment page in the shard key along with the `discussion_id` to allow MongoDB to split popular discussions between, while grouping discussions on the same shard. Issue the following operation at the Python/PyMongo console:

```
>>> db.command('shardCollection', 'comment_pages', {
...      key : { 'discussion_id' : 1, 'page': 1 } })
{ "collectionsharded" : "comment_pages", "ok" : 1 }
```

5 Python Application Development

5.1 Write a Tumblelog Application with Django MongoDB Engine

Introduction

In this tutorial, you will learn how to create a basic tumblelog application using the popular Django Python web-framework and the *MongoDB* database.

The tumblelog will consist of two parts:

1. A public site that lets people view posts and comment on them.
2. An admin site that lets you add, change and delete posts and publish comments.

This tutorial assumes that you are already familiar with Django and have a basic familiarity with MongoDB operation and have *installed MongoDB*.

Where to get help

If you're having trouble going through this tutorial, please post a message to mongodb-user or join the IRC chat in *#mongodb* on irc.freenode.net to chat with other MongoDB users who might be able to help.

Note: Django MongoDB Engine uses a forked version of Django 1.3 that adds non-relational support.

Installation

Begin by installing packages required by later steps in this tutorial.

Prerequisite

This tutorial uses pip to install packages and virtualenv to isolate Python environments. While these tools and this configuration are not required as such, they ensure a standard environment and are strongly recommended. Issue the following commands at the system prompt:

```
pip install virtualenv
virtualenv myproject
```

Respectively, these commands: install the `virtualenv` program (using `pip`) and create a isolated python environment for this project (named `myproject`.)

To activate `myproject` environment at the system prompt, use the following commands:

```
source myproject/bin/activate
```

Installing Packages

Django MongoDB Engine directly depends on:

- Django-nonrel, a fork of Django 1.3 that adds support for non-relational databases
- djangotoolbox, a bunch of utilities for non-relational Django applications and backends

Install by issuing the following commands:

```
pip install https://bitbucket.org/wkornewald/django-nonrel/get/tip.tar.gz
pip install https://bitbucket.org/wkornewald/djangotoolbox/get/tip.tar.gz
pip install https://github.com/django-nonrel/mongodb-engine/tarball/master
```

Continue with the tutorial to begin building the "tumblelog" application.

Build a Blog to Get Started

In this tutorial you will build a basic blog as the foundation of this application and use this as the basis of your tumblelog application. You will add the first post using the shell and then later use the Django administrative interface.

Call the `startproject` command, as with other Django projects, to get started and create the basic project skeleton:

```
django-admin.py startproject tumblelog
```

Configuring Django

Configure the database in the `tumblelog/settings.py` file:

```
DATABASES = {
   'default': {
      'ENGINE': 'django_mongodb_engine',
      'NAME': 'my_tumble_log'
   }
}
```

See Also:

The Django MongoDB Engine Settings documentation for more configuration options.

Define the Schema

The first step in writing a tumblelog in Django is to define the "models" or in MongoDB's terminology *documents*.

In this application, you will define posts and comments, so that each `Post` can contain a list of `Comments`. Edit the `tumblelog/models.py` file so it resembles the following:

```
from django.db import models
from django.core.urlresolvers import reverse

from djangotoolbox.fields import ListField, EmbeddedModelField


class Post(models.Model):
    created_at = models.DateTimeField(auto_now_add=True, db_index=True)
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    body = models.TextField()
    comments = ListField(EmbeddedModelField('Comment'), editable=False)

    def get_absolute_url(self):
        return reverse('post', kwargs={"slug": self.slug})

    def __unicode__(self):
        return self.title

    class Meta:
        ordering = ["-created_at"]


class Comment(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    body = models.TextField(verbose_name="Comment")
    author = models.CharField(verbose_name="Name", max_length=255)
```

The Django "nonrel" code looks the same as vanilla Django, however there is no built in support for some of MongoDB's native data types like Lists and Embedded data. `djangotoolbox` handles these definitions.

See Also:

The Django MongoDB Engine fields documentation for more.

The models declare an index to the `Post` class. One for the `created_at` date as our frontpage will order by date: there is no need to add `db_index` on `SlugField` because there is a default index on `SlugField`.

Add Data with the Shell

The `manage.py` provides a shell interface for the application that you can use to insert data into the tumblelog. Begin by issuing the following command to load the Python shell:

```
python manage.py shell
```

Create the first post using the following sequence of operations:

```
>>> from tumblelog.models import *
>>> post = Post(
... title="Hello World!",
... slug="hello-world",
... body = "Welcome to my new shiny Tumble log powered by MongoDB and Django-MongoDB!"
... )
>>> post.save()
```

Add comments using the following sequence of operations:

```
>>> post.comments
[]
>>> comment = Comment(
... author="Joe Bloggs",
... body="Great post! I'm looking forward to reading your blog")
>>> post.comments.append(comment)
>>> post.save()
```

Finally, inspect the post:

```
>>> post = Post.objects.get()
>>> post
<Post: Hello World!>
>>> post.comments
[<Comment: Comment object>]
```

Add the Views

Because django-mongodb provides tight integration with Django you can use generic views to display the frontpage and post pages for the tumblelog. Insert the following content into the `urls.py` file to add the views:

```
from django.conf.urls.defaults import patterns, include, url
from django.views.generic import ListView, DetailView
from tumblelog.models import Post

urlpatterns = patterns('',
    url(r'^$', ListView.as_view(
        queryset=Post.objects.all(),
        context_object_name="posts_list"),
        name="home"
    ),
    url(r'^post/(?P<slug>[a-zA-Z0-9-]+)/$', PostDetailView.as_view(
        queryset=Post.objects.all(),
        context_object_name="post"),
        name="post"
    ),
)
```

Add Templates

In the tumblelog directory add the following directories `templates` and `templates/tumblelog` for storing the tumblelog templates:

```
mkdir -p templates/tumblelog
```

Configure Django so it can find the templates by updating `TEMPLATE_DIRS` in the `settings.py` file to the following:

```
import os.path
TEMPLATE_DIRS = (
    os.path.join(os.path.realpath(__file__), ’../templates’),
)
```

Then add a base template that all others can inherit from. Add the following to `templates/base.html`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My Tumblelog</title>
    <link href="http://twitter.github.com/bootstrap/1.4.0/bootstrap.css" rel="stylesheet">
    <style>.content {padding-top: 80px;}</style>
  </head>

  <body>

    <div class="topbar">
      <div class="fill">
        <div class="container">
          <h1><a href="/" class="brand">My Tumblelog</a>! <small>Starring MongoDB and Django-MongoDB
        </div>
      </div>
    </div>

    <div class="container">
      <div class="content">
        {% block page_header %}{% endblock %}
        {% block content %}{% endblock %}
      </div>
    </div>

  </body>
</html>
```

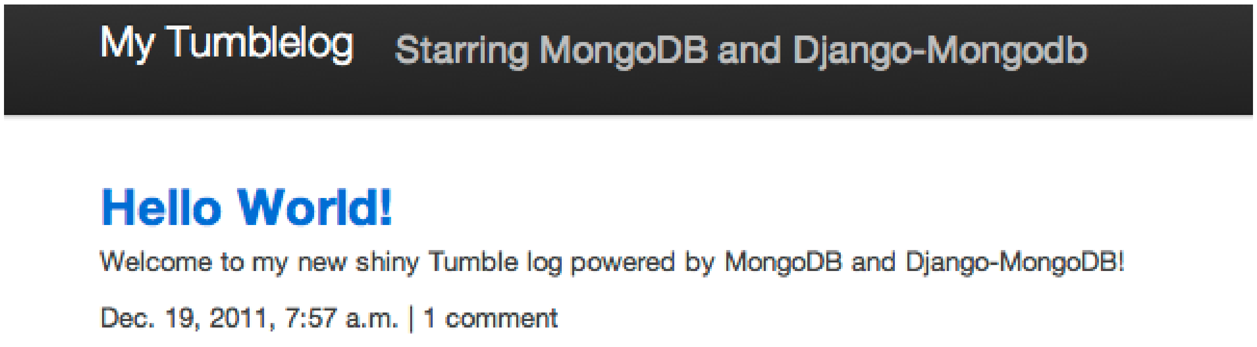
Create the frontpage for the blog, which should list all the posts. Add the following template to the `templates/tumblelog/post_list.html`:

```
{% extends "base.html" %}

{% block content %}
    {% for post in posts_list %}
      <h2><a href="{% url post slug=post.slug %}">{{ post.title }}</a></h2>
      <p>{{ post.body|truncatewords:20 }}</p>
      <p>
        {{ post.created_at }} |
        {% with total=post.comments|length %}
            {{ total }} comment{{ total|pluralize }}
        {% endwith %}
      </p>
    {% endfor %}
{% endblock %}
```

Finally, add `templates/tumblelog/post_detail.html` for the individual posts:

```
{% extends "base.html" %}

{% block page_header %}
  <div class="page-header">
    <h1>{{ post.title }}</h1>
  </div>
{% endblock %}

{% block content %}
  <p>{{ post.body }}<p>
  <p>{{ post.created_at }}</p>
  <hr>
  <h2>Comments</h2>
  {% if post.comments %}
    {% for comment in post.comments %}
       <p>{{ comment.body }}</p>
       <p><strong>{{ comment.author }}</strong> <small>on {{ comment.created_at }}</small></p>
      {{ comment.text }}
    {% endfor %}
  {% endif %}
{% endblock %}
```

Run `python manage.py runserver` to see your new tumblelog! Go to http://localhost:8000/ and you should see:



Hello World!
Welcome to my new shiny Tumble log powered by MongoDB and Django-MongoDB!
Dec. 19, 2011, 7:57 a.m. | 1 comment

Add Comments to the Blog

In the next step you will provide the facility for readers of the tumblelog to comment on posts. This a requires custom form and view to handle the form, and data. You will also update the template to include the form.

Create the Comments Form

You must customize form handling to deal with embedded comments. By extending `ModelForm`, it is possible to append the comment to the post on save. Create and add the following to `forms.py`:

```
from django.forms import ModelForm
from tumblelog.models import Comment


class CommentForm(ModelForm):

    def __init__(self, object, *args, **kwargs):
        """Override the default to store the original document
        that comments are embedded in.
        """
        self.object = object
        return super(CommentForm, self).__init__(*args, **kwargs)

    def save(self, *args):
        """Append to the comments list and save the post"""
        self.object.comments.append(self.instance)
        self.object.save()
        return self.object

    class Meta:
        model = Comment
```

Handle Comments in the View

You must extend the generic views need to handle the form logic. Add the following to the `views.py` file:

```
from django.http import HttpResponseRedirect
from django.views.generic import DetailView
from tumblelog.forms import CommentForm


class PostDetailView(DetailView):
    methods = ['get', 'post']

    def get(self, request, *args, **kwargs):
        self.object = self.get_object()
        form = CommentForm(object=self.object)
        context = self.get_context_data(object=self.object, form=form)
        return self.render_to_response(context)

    def post(self, request, *args, **kwargs):
        self.object = self.get_object()
        form = CommentForm(object=self.object, data=request.POST)

        if form.is_valid():
            form.save()
            return HttpResponseRedirect(self.object.get_absolute_url())

        context = self.get_context_data(object=self.object, form=form)
        return self.render_to_response(context)
```

Note: The `PostDetailView` class extends the `DetailView` class so that it can handle `GET` and `POST` requests. On `POST`, `post()` validates the comment: if valid, `post()` appends the comment to the post.

Don't forget to update the `urls.py` file and import the `PostDetailView` class to replace the `DetailView` class.

Add Comments to the Templates

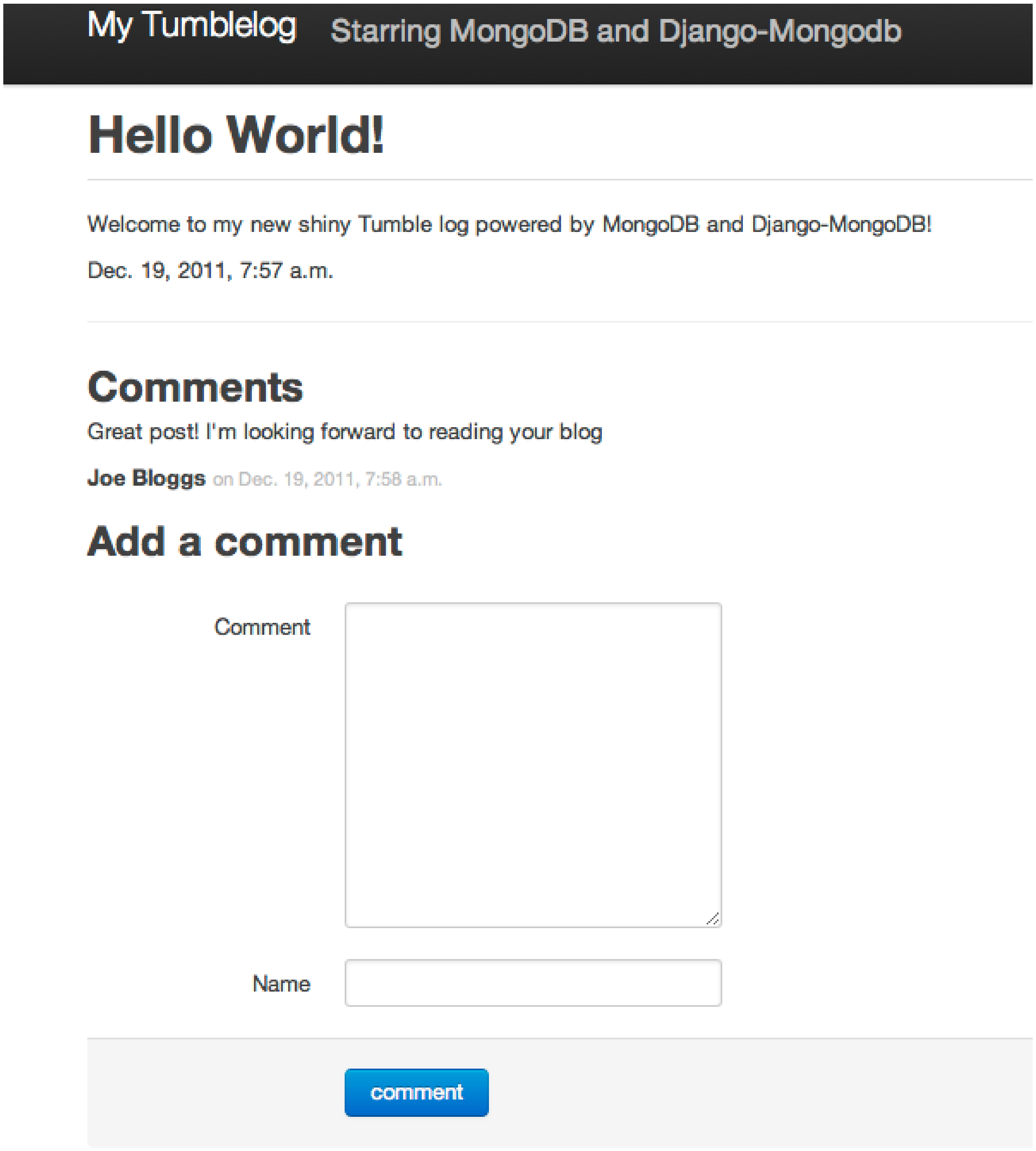
Finally, you can add the form to the templates, so that readers can create comments. Splitting the template for the forms out into `templates/_forms.html` will allow maximum reuse of forms code:

```
<fieldset>
{% for field in form.visible_fields %}
<div class="clearfix {% if field.errors %}error{% endif %}">
  {{ field.label_tag }}
  <div class="input">
    {{ field }}
    {% if field.errors or field.help_text %}
      <span class="help-inline">
      {% if field.errors %}
        {{ field.errors|join:' ' }}
      {% else %}
        {{ field.help_text }}
      {% endif %}
      </span>
    {% endif %}
  </div>
</div>
{% endfor %}
{% csrf_token %}
<div style="display:none">{% for h in form.hidden_fields %} {{ h }}{% endfor %}</div>
</fieldset>
```

After the comments section in `post_detail.html` add the following code to generate the comments form:

```
<h2>Add a comment</h2>
<form action="." method="post">
  {% include "_forms.html" %}
  <div class="actions">
    <input type="submit" class="btn primary" value="comment">
  </div>
</form>
```

Your tumblelog's readers can now comment on your posts! Run `python manage.py runserver` to see the changes. Run `python manage.py runserver` and go to http://localhost:8000/hello-world/ to see the following:



Hello World!

Welcome to my new shiny Tumble log powered by MongoDB and Django-MongoDB!

Dec. 19, 2011, 7:57 a.m.

Comments

Great post! I'm looking forward to reading your blog

Joe Bloggs on Dec. 19, 2011, 7:58 a.m.

Add a comment

Add Site Administration Interface

While you may always add posts using the shell interface as above, you can easily create an administrative interface for posts with Django. Enable the admin by adding the following apps to `INSTALLED_APPS` in `settings.py`.

- `django.contrib.admin`
- `djangomongodbengine`
- `djangotoolbox`

- `tumblelog`

Warning: This application does not require the `Sites` framework. As a result, remove `django.contrib.sites` from `INSTALLED_APPS`. If you need it later please read SITE_ID issues document.

Create a `admin.py` file and register the `Post` model with the admin app:

```
from django.contrib import admin
from tumblelog.models import Post

admin.site.register(Post)
```

Note: The above modifications deviate from the default django-nonrel and `djangotoolbox` mode of operation. Django's administration module will not work unless you exclude the `comments` field. By making the `comments` field non-editable in the "admin" model definition, you will allow the administrative interface to function.

If you need an administrative interface for a ListField you must write your own Form / Widget.

See Also:

The Django Admin documentation docs for additional information.

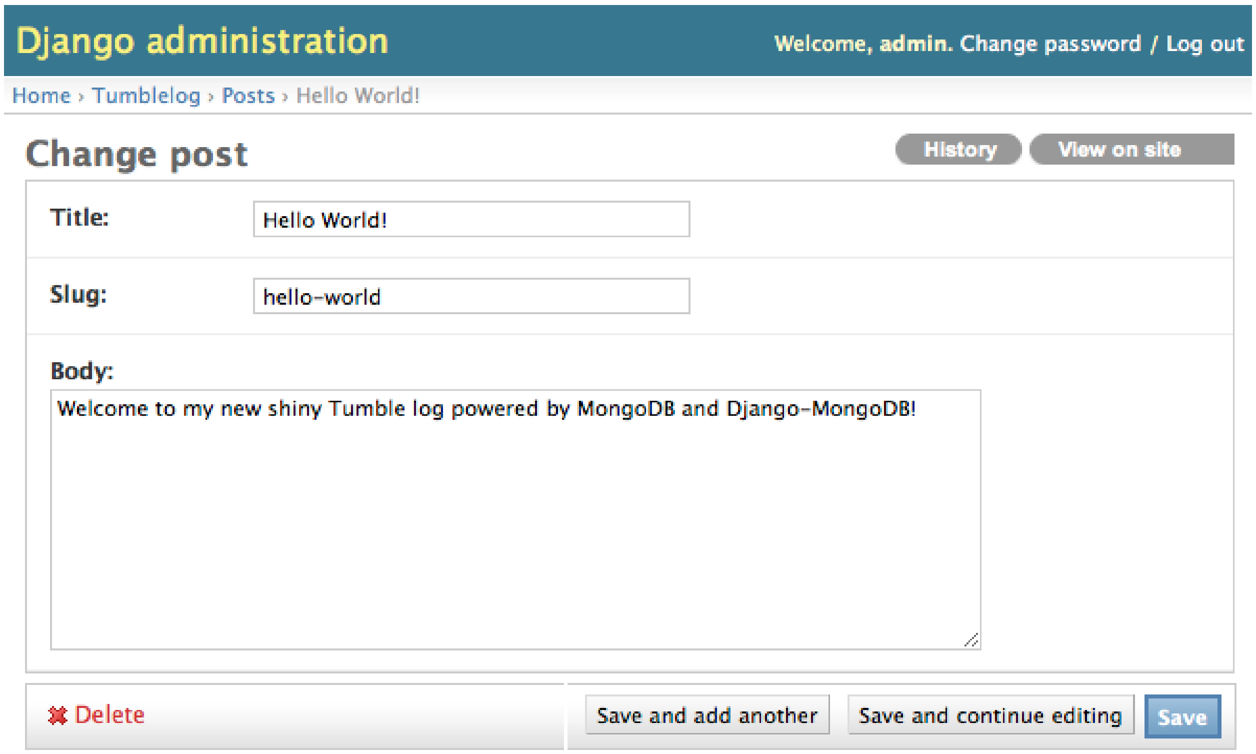
Update the `urls.py` to enable the administrative interface. Add the import and discovery mechanism to the top of the file and then add the admin import rule to the `urlpatterns`:

```
# Enable admin
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',

    # ...

    url(r'^admin/', include(admin.site.urls)),
)
```

Finally, add a superuser and setup the indexes by issuing the following command at the system prompt:

```
python manage.py syncdb
```

Once done run the server and you can login to admin by going to http://localhost:8000/admin/.


Django administration
Welcome, admin. Change password / Log out
Home › Tumblelog › Posts › Hello World!
Change post
History
View on site
Delete
Save and add another
Save and continue editing
Save

Title: Hello World!

Slug: hello-world

Body: Welcome to my new shiny Tumble log powered by MongoDB and Django-MongoDB!

Convert the Blog to a Tumblelog

Currently, the application only supports posts. In this section you will add special post types including: *Video*, *Image* and *Quote* to provide a more traditional tumblelog application. Adding this data requires no migration.

In `models.py` update the `Post` class to add new fields for the new post types. Mark these fields with `blank=True` so that the fields can be empty.

Update `Post` in the `models.py` files to resemble the following:

```
POST_CHOICES = (
    ('p', 'post'),
    ('v', 'video'),
    ('i', 'image'),
    ('q', 'quote'),
)


class Post(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    title = models.CharField(max_length=255)
    slug = models.SlugField()

    comments = ListField(EmbeddedModelField('Comment'), editable=False)

    post_type = models.CharField(max_length=1, choices=POST_CHOICES, default='p')

    body = models.TextField(blank=True, help_text="The body of the Post / Quote")
    embed_code = models.TextField(blank=True, help_text="The embed code for video")
    image_url = models.URLField(blank=True, help_text="Image src")
    author = models.CharField(blank=True, max_length=255, help_text="Author name")
    def get_absolute_url(self):
        return reverse('post', kwargs={"slug": self.slug})

    def __unicode__(self):
        return self.title
```

Note: Django-Nonrel doesn't support multi-table inheritance. This means that you will have to manually create an administrative form to handle data validation for the different post types.

The "Abstract Inheritance" facility means that the view logic would need to merge data from multiple collections.

The administrative interface should now handle adding multiple types of post. To conclude this process, you must update the frontend display to handle and output the different post types.

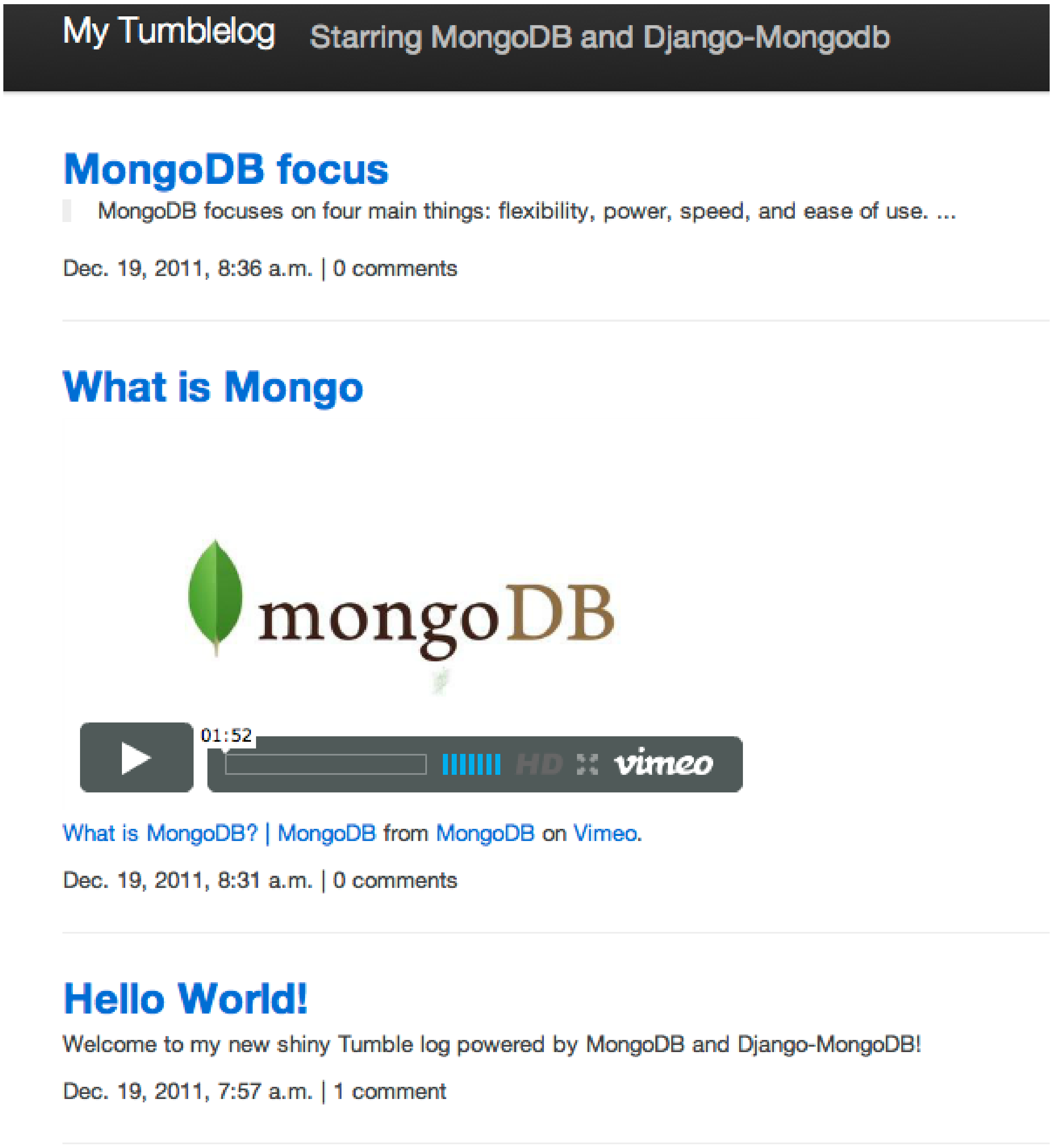
In the `post_list.html` file, change the post output display to resemble the following:

```
{% if post.post_type == 'p' %}
  <p>{{ post.body|truncatewords:20 }}</p>
{% endif %}
{% if post.post_type == 'v' %}
  {{ post.embed_code|safe }}
{% endif %}
{% if post.post_type == 'i' %}
  <p><img src="{{ post.image_url }}" /><p>
{% endif %}
{% if post.post_type == 'q' %}
  <blockquote>{{ post.body|truncatewords:20 }}</blockquote>
  <p>{{ post.author }}</p>
{% endif %}
```

In the `post_detail.html` file, change the output for full posts:

```
{% if post.post_type == 'p' %}
  <p>{{ post.body }}<p>
{% endif %}
{% if post.post_type == 'v' %}
  {{ post.embed_code|safe }}
{% endif %}
{% if post.post_type == 'i' %}
  <p><img src="{{ post.image_url }}" /><p>
{% endif %}
{% if post.post_type == 'q' %}
  <blockquote>{{ post.body }}</blockquote>
  <p>{{ post.author }}</p>
{% endif %}
```

Now you have a fully fledged tumbleblog using Django and MongoDB!



My Tumblelog Starring MongoDB and Django-Mongodb

MongoDB focus

MongoDB focuses on four main things: flexibility, power, speed, and ease of use. ...

Dec. 19, 2011, 8:36 a.m. | 0 comments

What is Mongo

What is MongoDB? | MongoDB from MongoDB on Vimeo.

Dec. 19, 2011, 8:31 a.m. | 0 comments

Hello World!

Welcome to my new shiny Tumble log powered by MongoDB and Django-MongoDB!

Dec. 19, 2011, 7:57 a.m. | 1 comment

5.2 Write a Tumblelog Application with Flask and MongoEngine

Introduction

This tutorial describes the process for creating a basic tumblelog application using the popular Flask Python web-framework in conjunction with the *MongoDB* database.

The tumblelog will consist of two parts:

1. A public site that lets people view posts and comment on them.
2. An admin site that lets you add and change posts.

This tutorial assumes that you are already familiar with Flask and have a basic familiarity with MongoDB and have *installed MongoDB*. This tutorial uses MongoEngine as the Object Document Mapper (ODM,) this component may simplify the interaction between Flask and MongoDB.

Where to get help

If you're having trouble going through this tutorial, please post a message to mongodb-user or join the IRC chat in *#mongodb* on irc.freenode.net to chat with other MongoDB users who might be able to help.

Installation

Begin by installing packages required by later steps in this tutorial.

Prerequisite

This tutorial uses pip to install packages and virtualenv to isolate Python environments. While these tools and this configuration are not required as such, they ensure a standard environment and are strongly recommended. Issue the following command at the system prompt:

```
pip install virtualenv
virtualenv myproject
```

Respectively, these commands: install the `virtualenv` program (using `pip`) and create a isolated python environment for this project (named `myproject`.)

To activate `myproject` environment at the system prompt, use the following command:

```
source myproject/bin/activate
```

Install Packages

Flask is a "microframework," because it provides a small core of functionality and is highly extensible. For the "tumblelog" project, this tutorial includes task and the following extension:

- WTForms provides easy form handling.
- Flask-MongoEngine provides integration between MongoEngine, Flask, and WTForms.
- Flask-Script for an easy to use development server

Install with the following commands:

```
pip install flask
pip install flask-script
pip install WTForms
pip install mongoengine
pip install flask_mongoengine
```

Continue with the tutorial to begin building the "tumblelog" application.

Build a Blog to Get Started

First, create a simple "bare bones" application. Make a directory named `tumblelog` for the project and then, add the following content into a file named `__init__.py`:

```
from flask import Flask
app = Flask(__name__)


if __name__ == '__main__':
    app.run()
```

Next, create the `manage.py` file. [8] Use this file to load additional Flask-scripts in the future. Flask-scripts provides a development server and shell:

```
# Set the path
import os, sys
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from flask.ext.script import Manager, Server
from tumblelog import app

manager = Manager(app)

# Turn on debugger by default and reloader
manager.add_command("runserver", Server(
    use_debugger = True,
    use_reloader = True,
    host = '0.0.0.0')
)

if __name__ == "__main__":
    manager.run()
```

You can run this application with a test server, by issuing the following command at the system prompt:

```
python manage.py runserver
```

There should be no errors, and you can visit http://localhost:5000/ in a web browser to view a page with a "404" message.

Configure MongoEngine and Flask

Install the Flask extension and add the configuration. Update `tumblelog/__init__.py` so that it resembles the following:

```
from flask import Flask
from flask.ext.mongoengine import MongoEngine

app = Flask(__name__)
app.config["MONGODB_DB"] = "my_tumble_log"
app.config["SECRET_KEY"] = "KeepThisS3cr3t"

db = MongoEngine(app)
```

[8] This concept will be familiar to users of Django.

```
if __name__ == '__main__':
    app.run()
```

See Also:

The MongoEngine Settings documentation for additional configuration options.

Define the Schema

The first step in writing a tumblelog in Flask is to define the "models" or in MongoDB's terminology *documents*.

In this application, you will define posts and comments, so that each `Post` can contain a list of `Comments`. Edit the `models.py` file so that it resembles the following:

```
import datetime
from flask import url_for
from tumblelog import db


class Post(db.Document):
    created_at = db.DateTimeField(default=datetime.datetime.now, required=True)
    title = db.StringField(max_length=255, required=True)
    slug = db.StringField(max_length=255, required=True)
    body = db.StringField(required=True)
    comments = db.ListField(db.EmbeddedDocumentField('Comment'))

    def get_absolute_url(self):
        return url_for('post', kwargs={"slug": self.slug})

    def __unicode__(self):
        return self.title

    meta = {
        'allow_inheritance': True,
        'indexes': ['-created_at', 'slug'],
        'ordering': ['-created_at']
    }


class Comment(db.EmbeddedDocument):
    created_at = db.DateTimeField(default=datetime.datetime.now, required=True)
    body = db.StringField(verbose_name="Comment", required=True)
    author = db.StringField(verbose_name="Name", max_length=255, required=True)
```

As above, MongoEngine syntax is simple and declarative. If you have a Django background, the syntax may look familiar. This example defines indexes for `Post`: one for the `created_at` date as our frontpage will order by date and another for the individual post `slug`.

Add Data with the Shell

The `manage.py` provides a shell interface for the application that you can use to insert data into the tumblelog. Before configuring the "urls" and "views" for this application, you can use this interface to interact with your the tumblelog. Begin by issuing the following command to load the Python shell:

```
python manage.py shell
```

Create the first post using the following sequence of operations:

```
>>> from tumblelog.models import *
>>> post = Post(
... title="Hello World!",
... slug="hello-world",
... body="Welcome to my new shiny Tumble log powered by MongoDB, MongoEngine, and Flask"
... )
>>> post.save()
```

Add comments using the following sequence of operations:

```
>>> post.comments
[]
>>> comment = Comment(
... author="Joe Bloggs",
... body="Great post! I'm looking forward to reading your blog!"
... )
>>> post.comments.append(comment)
>>> post.save()
```

Finally, inspect the post:

```
>>> post = Post.objects.get()
>>> post
<Post: Hello World!>
>>> post.comments
[<Comment: Comment object>]
```

Add the Views

Using Flask's class-based views system allows you to produce `List` and `Detail` views for tumblelog posts. Add `views.py` and create a *posts* blueprint:

```
from flask import Blueprint, request, redirect, render_template, url_for
from flask.views import MethodView
from tumblelog.models import Post, Comment

posts = Blueprint('posts', __name__, template_folder='templates')


class ListView(MethodView):

    def get(self):
        posts = Post.objects.all()
        return render_template('posts/list.html', posts=posts)


class DetailView(MethodView):

    def get(self, slug):
        post = Post.objects.get_or_404(slug=slug)
        return render_template('posts/detail.html', post=post)


# Register the urls
posts.add_url_rule('/', view_func=ListView.as_view('list'))
posts.add_url_rule('/<slug>/', view_func=DetailView.as_view('detail'))
```

Now in `__init__.py` register the blueprint, avoiding a circular dependency by registering the blueprints in a method. Add the following code to the module:

```
def register_blueprints(app):
    # Prevents circular imports
    from tumblelog.views import posts
    app.register_blueprint(posts)

register_blueprints(app)
```

Add this method and method call to the main body of the module and not in the main block.

Add Templates

In the `tumblelog` directory add the `templates` and `templates/posts` directories to store the tumblelog templates:

```
mkdir -p templates/posts
```

Create a base template. All other templates will inherit from this template, which should exist in the `templates/base.html` file:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My Tumblelog</title>
    <link href="http://twitter.github.com/bootstrap/1.4.0/bootstrap.css" rel="stylesheet">
    <style>.content {padding-top: 80px;}</style>
  </head>

  <body>

    {%- block topbar -%}
    <div class="topbar">
      <div class="fill">
        <div class="container">
          <h2>
              <a href="/" class="brand">My Tumblelog</a> <small>Starring Flask, MongoDB and MongoEng
          </h2>
        </div>
      </div>
    </div>
    {%- endblock -%}

    <div class="container">
      <div class="content">
        {% block page_header %}{% endblock %}
        {% block content %}{% endblock %}
      </div>
    </div>
    {% block js_footer %}{% endblock %}
  </body>
</html>
```

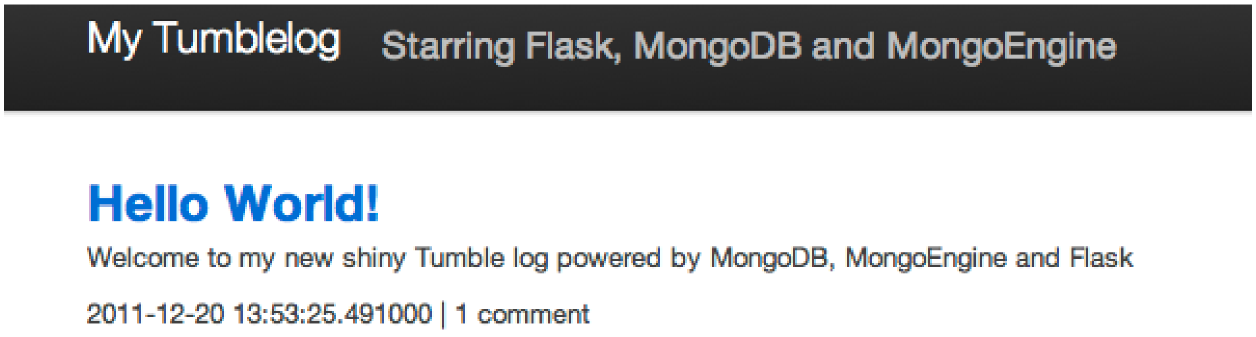
Continue by creating a landing page for the blog that will list all posts. Add the following to the `templates/posts/list.html` file:

```
{% extends "base.html" %}

{% block content %}
    {% for post in posts %}
      <h2><a href="{{ url_for('posts.detail', slug=post.slug) }}">{{ post.title }}</a></h2>
      <p>{{ post.body|truncate(100) }}</p>
      <p>
        {{ post.created_at.strftime('%H:%M %Y-%m-%d') }} |
        {% with total=post.comments|length %}
            {{ total }} comment {%- if total > 1 %}s{%- endif -%}
        {% endwith %}
      </p>
    {% endfor %}
{% endblock %}
```

Finally, add `templates/posts/detail.html` template for the individual posts:

```
{% extends "base.html" %}

{% block page_header %}
  <div class="page-header">
    <h1>{{ post.title }}</h1>
  </div>
{% endblock %}

{% block content %}
  <p>{{ post.body }}<p>
  <p>{{ post.created_at.strftime('%H:%M %Y-%m-%d') }}</p>
  <hr>
  <h2>Comments</h2>
  {% if post.comments %}
    {% for comment in post.comments %}
       <p>{{ comment.body }}</p>
       <p><strong>{{ comment.author }}</strong> <small>on {{ comment.created_at.strftime('%H:%M %Y-%
     {{ comment.text }}
    {% endfor %}
  {% endif %}
{% endblock %}
```

At this point, you can run the `python manage.py runserver` command again to see your new tumblelog! Go to http://localhost:5000 to see something that resembles the following:


My Tumblelog Starring Flask, MongoDB and MongoEngine

Hello World!

Welcome to my new shiny Tumble log powered by MongoDB, MongoEngine and Flask

2011-12-20 13:53:25.491000 | 1 comment

Add Comments to the Blog

In the next step you will provide the facility for readers of the tumblelog to comment on posts. To provide commenting, you will create a form using WTForms that will update the view to handle the form data and update the template to include the form.

Handle Comments in the View

Begin by updating and refactoring the `views.py` file so that it can handle the form. Begin by adding the `import` statement and the `DetailView` class to this file:

```
from flask.ext.mongoengine.wtf import model_form

...

class DetailView(MethodView):

    form = model_form(Comment, exclude=['created_at'])

    def get_context(self, slug):
        post = Post.objects.get_or_404(slug=slug)
        form = self.form(request.form)

        context = {
            "post": post,
            "form": form
        }
        return context

    def get(self, slug):
        context = self.get_context(slug)
        return render_template('posts/detail.html', **context)

    def post(self, slug):
        context = self.get_context(slug)
        form = context.get('form')

        if form.validate():
            comment = Comment()
            form.populate_obj(comment)

            post = context.get('post')
            post.comments.append(comment)
            post.save()

            return redirect(url_for('posts.detail', slug=slug))

        return render_template('posts/detail.html', **context)
```

Note: `DetailView` extends the default Flask `MethodView`. This code remains DRY by defining a *get_context* method to get the default context for both `GET` and `POST` requests. On `POST`, `post()` validates the comment: if valid, `post()` appends the comment to the post.

Add Comments to the Templates

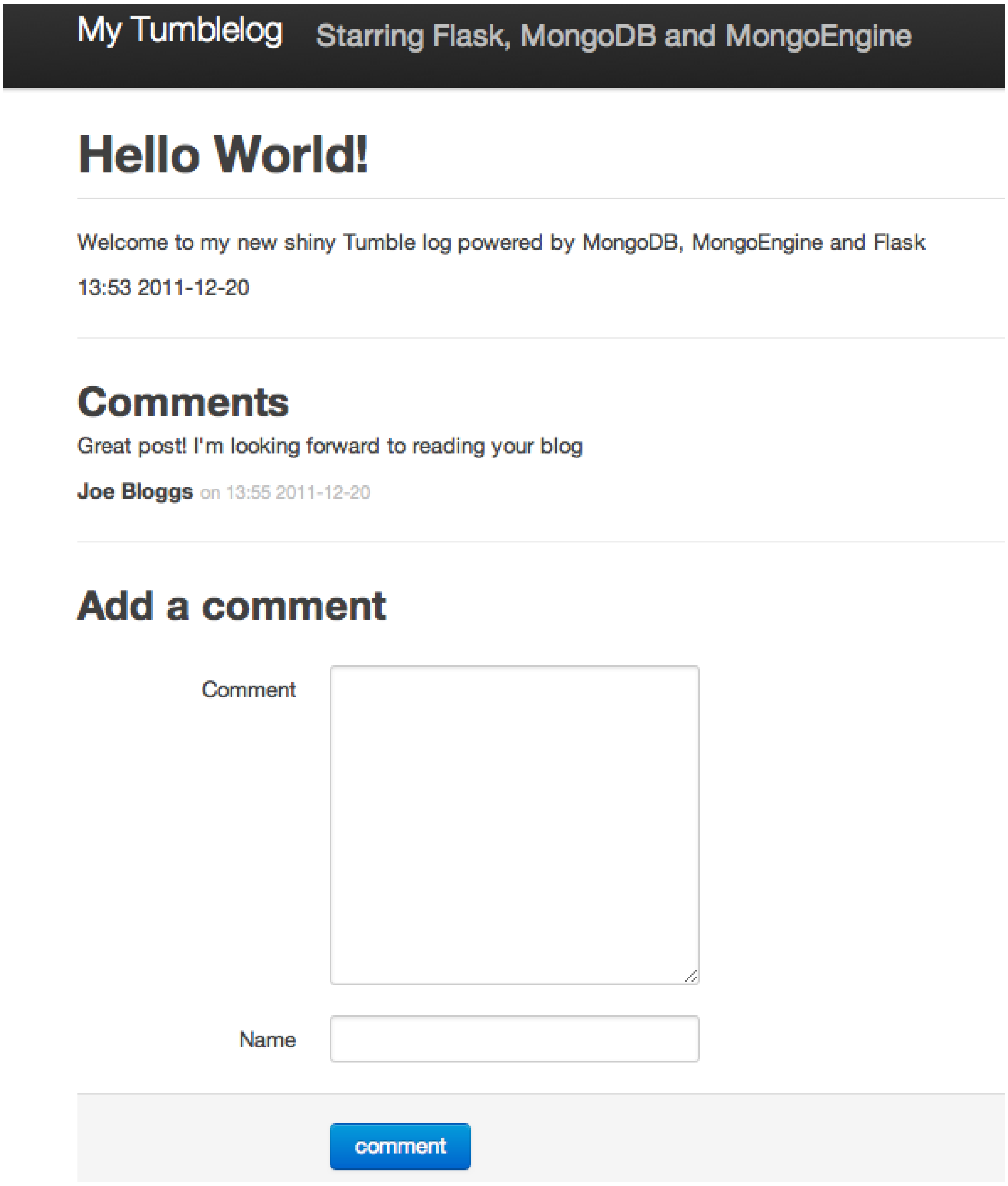
Finally, you can add the form to the templates, so that readers can create comments. Create a macro for the forms in `templates/_forms.html` will allow you to reuse the form code:

```
{% macro render(form) -%}
<fieldset>
{% for field in form %}
{% if field.type in ['CSRFTokenField', 'HiddenField'] %}
  {{ field() }}
{% else %}
  <div class="clearfix {% if field.errors %}error{% endif %}">
    {{ field.label }}
    <div class="input">
      {% if field.name == "body" %}
        {{ field(rows=10, cols=40) }}
      {% else %}
        {{ field() }}
      {% endif %}
      {% if field.errors or field.help_text %}
        <span class="help-inline">
        {% if field.errors %}
          {{ field.errors|join(' ') }}
        {% else %}
          {{ field.help_text }}
        {% endif %}
        </span>
      {% endif %}
    </div>
  </div>
{% endif %}
{% endfor %}
</fieldset>
{% endmacro %}
```

Add the comments form to `templates/posts/detail.html`. Insert an `import` statement at the top of the page and then output the form after displaying comments:

```
{% import "_forms.html" as forms %}

...

<hr>
<h2>Add a comment</h2>
<form action="." method="post">
  {{ forms.render(form) }}
  <div class="actions">
    <input type="submit" class="btn primary" value="comment">
  </div>
</form>
```

Your tumblelog's readers can now comment on your posts! Run `python manage.py runserver` to see the changes.


My Tumblelog Starring Flask, MongoDB and MongoEngine

Hello World!

Welcome to my new shiny Tumble log powered by MongoDB, MongoEngine and Flask

13:53 2011-12-20

Comments

Great post! I'm looking forward to reading your blog

Joe Bloggs on 13:55 2011-12-20

Add a comment

Add a Site Administration Interface

While you may always add posts using the shell interface as above, in this step you will add an administrative interface for the tumblelog site. To add the administrative interface you will add authentication and an additional view. This tutorial only addresses adding and editing posts: a “delete” view and detection of slug collisions are beyond the scope of this tutorial.

Add Basic Authentication

For the purposes of this tutorial all we need is a very basic form of authentication. The following example borrows from the an example Flask "Auth snippet". Create the file `auth.py` with the following content:

```python
from functools import wraps
from flask import request, Response


def check_auth(username, password):
    """This function is called to check if a username /
    password combination is valid.
    """
    return username == 'admin' and password == 'secret'


def authenticate():
    """Sends a 401 response that enables basic auth"""
    return Response(
    'Could not verify your access level for that URL.\n'
    'You have to login with proper credentials', 401,
    {'WWW-Authenticate': 'Basic realm="Login Required"'})


def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            return authenticate()
        return f(*args, **kwargs)
    return decorated
```

Note: This creates a *requires_auth* decorator: provides basic authentication. Decorate any view that needs authentication with this decorator. The username is `admin` and password is `secret`.

Write an Administrative View

Create the views and admin blueprint in `admin.py`. The following view is deliberately generic, to facilitate customization.

```python
from flask import Blueprint, request, redirect, render_template, url_for
from flask.views import MethodView

from flask.ext.mongoengine.wtf import model_form

from tumblelog.auth import requires_auth
from tumblelog.models import Post, Comment

admin = Blueprint('admin', __name__, template_folder='templates')


class List(MethodView):
    decorators = [requires_auth]
    cls = Post
    def get(self):
        posts = self.cls.objects.all()
        return render_template('admin/list.html', posts=posts)


class Detail(MethodView):

    decorators = [requires_auth]

    def get_context(self, slug=None):
        form_cls = model_form(Post, exclude=('created_at', 'comments'))

        if slug:
            post = Post.objects.get_or_404(slug=slug)
            if request.method == 'POST':
                form = form_cls(request.form, inital=post._data)
            else:
                form = form_cls(obj=post)
        else:
            post = Post()
            form = form_cls(request.form)

        context = {
            "post": post,
            "form": form,
            "create": slug is None
        }
        return context

    def get(self, slug):
        context = self.get_context(slug)
        return render_template('admin/detail.html', **context)

    def post(self, slug):
        context = self.get_context(slug)
        form = context.get('form')

        if form.validate():
            post = context.get('post')
            form.populate_obj(post)
            post.save()

            return redirect(url_for('admin.index'))
        return render_template('admin/detail.html', **context)


# Register the urls
admin.add_url_rule('/admin/', view_func=List.as_view('index'))
admin.add_url_rule('/admin/create/', defaults={'slug': None}, view_func=Detail.as_view('create'))
admin.add_url_rule('/admin/<slug>/', view_func=Detail.as_view('edit'))
```

Note: Here, the `List` and `Detail` views are similar to the frontend of the site; however, `requires_auth` decorates both views.

The “Detail” view is slightly more complex: to set the context, this view checks for a slug and if there is no slug, `Detail` uses the view for creating a new post. If a slug exists, `Detail` uses the view for editing an existing post.

In the `__init__.py` file update the `register_blueprints()` method to import the new admin blueprint.

```
def register_blueprints(app):
    # Prevents circular imports
    from tumblelog.views import posts
    from tumblelog.admin import admin
    app.register_blueprint(posts)
    app.register_blueprint(admin)
```

Create Administrative Templates

Similar to the user-facing portion of the site, the administrative section of the application requires three templates: a base template a list view, and a detail view.

Create an `admin` directory for the templates. Add a simple main index page for the admin in the `templates/admin/base.html` file:

```
{% extends "base.html" %}

{%- block topbar -%}
<div class="topbar" data-dropdown="dropdown">
  <div class="fill">
    <div class="container">
      <h2>
           <a href="{{ url_for('admin.index') }}" class="brand">My Tumblelog Admin</a>
      </h2>
      <ul class="nav secondary-nav">
        <li class="menu">
           <a href="{{ url_for("admin.create") }}" class="btn primary">Create new post</a>
        </li>
      </ul>
    </div>
  </div>
</div>
{%- endblock -%}
```

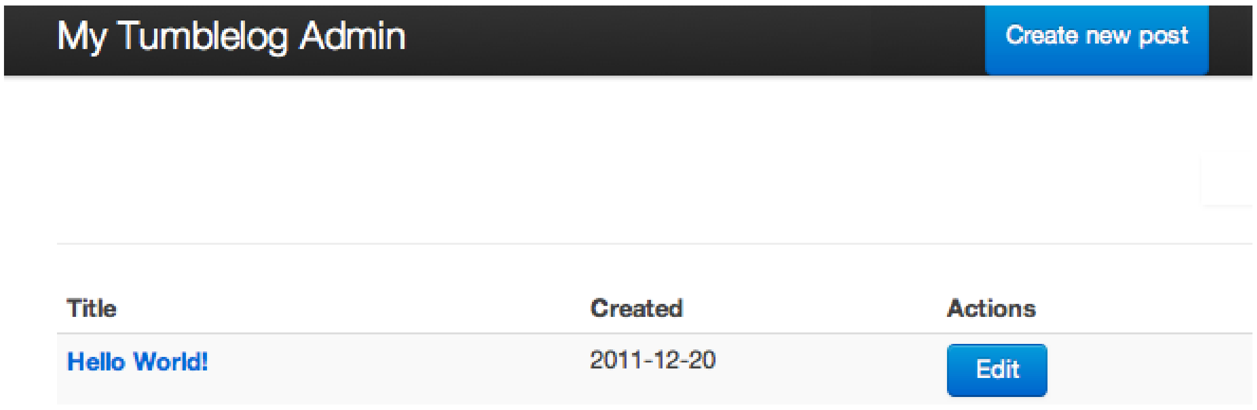
List all the posts in the `templates/admin/list.html` file:

```
{% extends "admin/base.html" %}

{% block content %}
  <table  class="condensed-table zebra-striped">
    <thead>
      <th>Title</th>
      <th>Created</th>
      <th>Actions</th>
    </thead>
    <tbody>
    {% for post in posts %}
      <tr>
        <th><a href="{{ url_for('admin.edit', slug=post.slug) }}">{{ post.title }}</a></th>
        <td>{{ post.created_at.strftime('%Y-%m-%d') }}</td>
        <td><a href="{{ url_for("admin.edit", slug=post.slug) }}" class="btn primary">Edit</a></td>
      </tr>
    {% endfor %}
    </tbody>
  </table>
{% endblock %}
```

Add a temple to create and edit posts in the `templates/admin/detail.html` file:

```
{% extends "admin/base.html" %}
{% import "_forms.html" as forms %}

{% block content %}
  <h2>
    {% if create %}
      Add new Post
    {% else %}
      Edit Post
    {% endif %}
  </h2>

  <form action="?{{ request.query_string }}" method="post">
    {{ forms.render(form) }}
    <div class="actions">
      <input type="submit" class="btn primary" value="save">
      <a href="{{ url_for("admin.index") }}" class="btn secondary">Cancel</a>
    </div>
  </form>
{% endblock %}
```

The administrative interface is ready for use. Restart the test server (i.e. `runserver`) so that you can log in to the administrative interface located at http://localhost:5000/admin/. (The username is `admin` and the password is `secret`.)



Title	Created	Actions
Hello World!	2011-12-20	Edit

Converting the Blog to a Tumblelog

Currently, the application only supports posts. In this section you will add special post types including: *Video*, *Image* and *Quote* to provide a more traditional tumblelog application. Adding this data requires no migration because MongoEngine supports document inheritance.

Begin by refactoring the `Post` class to operate as a base class and create new classes for the new post types. Update the `models.py` file to include the code to replace the old `Post` class:

```
class Post(db.DynamicDocument):
    created_at = db.DateTimeField(default=datetime.datetime.now, required=True)
    title = db.StringField(max_length=255, required=True)
    slug = db.StringField(max_length=255, required=True)
    comments = db.ListField(db.EmbeddedDocumentField('Comment'))
    def get_absolute_url(self):
        return url_for('post', kwargs={"slug": self.slug})

    def __unicode__(self):
        return self.title

    @property
    def post_type(self):
        return self.__class__.__name__

    meta = {
        'allow_inheritance': True,
        'indexes': ['-created_at', 'slug'],
        'ordering': ['-created_at']
    }


class BlogPost(Post):
    body = db.StringField(required=True)


class Video(Post):
    embed_code = db.StringField(required=True)


class Image(Post):
    image_url = db.StringField(required=True, max_length=255)


class Quote(Post):
    body = db.StringField(required=True)
    author = db.StringField(verbose_name="Author Name", required=True, max_length=255)
```

Note: In the `Post` class the `post_type` helper returns the class name, which will make it possible to render the various different post types in the templates.

As MongoEngine handles returning the correct classes when fetching `Post` objects you do not need to modify the interface view logic: only modify the templates.

Update the `templates/posts/list.html` file and change the post output format as follows:

```
{% if post.body %}
  {% if post.post_type == 'Quote' %}
    <blockquote>{{ post.body|truncate(100) }}</blockquote>
    <p>{{ post.author }}</p>
  {% else %}
    <p>{{ post.body|truncate(100) }}</p>
  {% endif %}
{% endif %}
{% if post.embed_code %}
  {{ post.embed_code|safe() }}
{% endif %}
{% if post.image_url %}
  <p><img src="{{ post.image_url }}" /><p>
{% endif %}
```

In the `templates/posts/detail.html` change the output for full posts as follows:

```
{% if post.body %}
  {% if post.post_type == 'Quote' %}
    <blockquote>{{ post.body }}</blockquote>
    <p>{{ post.author }}</p>
  {% else %}
    <p>{{ post.body }}</p>
  {% endif %}
{% endif %}
{% if post.embed_code %}
  {{ post.embed_code|safe() }}
{% endif %}
{% if post.image_url %}
  <p><img src="{{ post.image_url }}" /><p>
{% endif %}
```

Updating the Administration

In this section you will update the administrative interface to support the new post types.

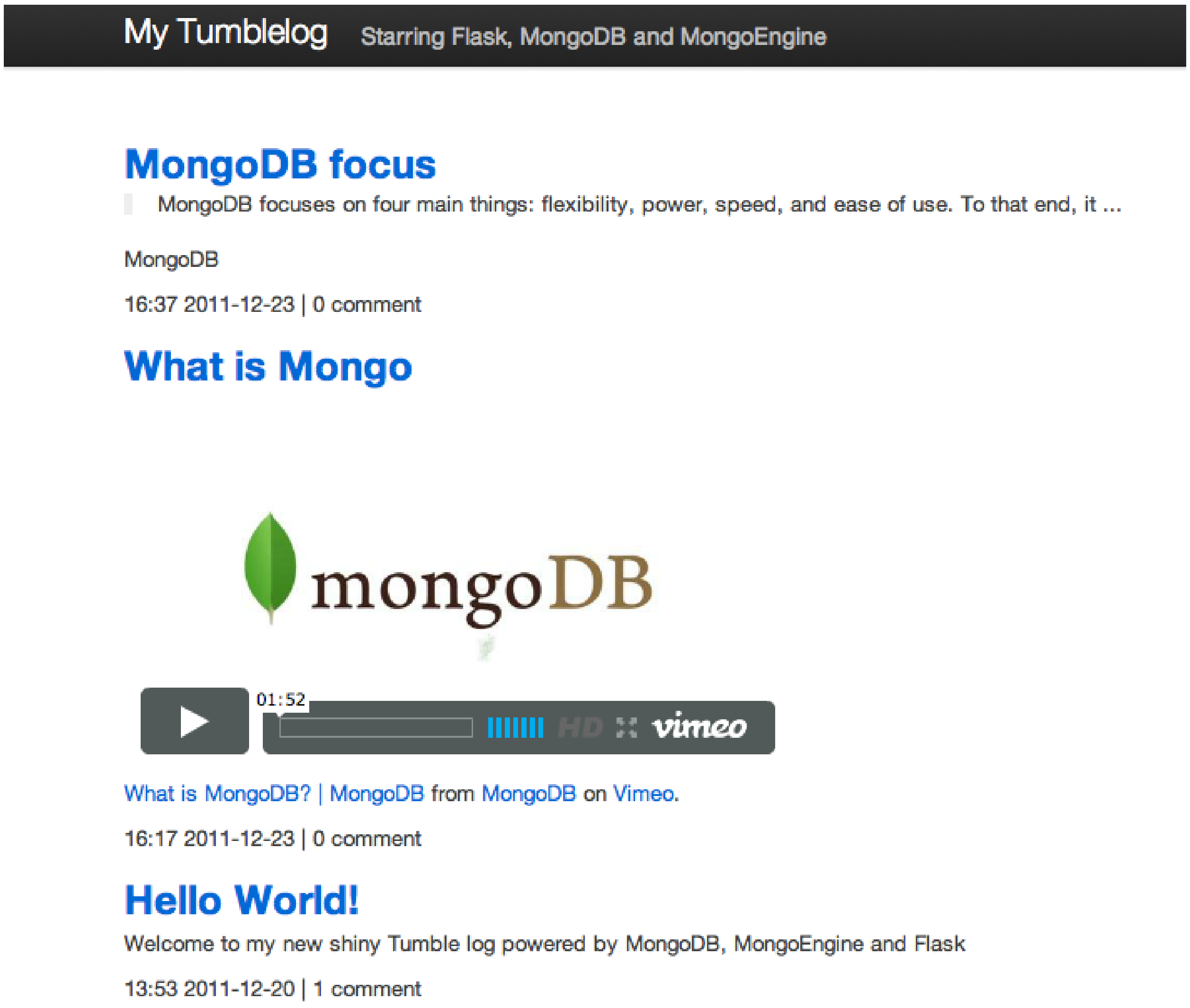
Begin by, updating the `admin.py` file to import the new document models and then update `get_context()` in the `Detail` class to dynamically create the correct model form to use:

```
from tumblelog.models import Post, BlogPost, Video, Image, Quote, Comment

# ...

class Detail(MethodView):

    decorators = [requires_auth]
    # Map post types to models
    class_map = {
        'post': BlogPost,
        'video': Video,
        'image': Image,
        'quote': Quote,
    }

    def get_context(self, slug=None):

        if slug:
            post = Post.objects.get_or_404(slug=slug)
            # Handle old posts types as well
            cls = post.__class__ if post.__class__ != Post else BlogPost
            form_cls = model_form(cls,  exclude=('created_at', 'comments'))
            if request.method == 'POST':
                form = form_cls(request.form, inital=post._data)
            else:
                form = form_cls(obj=post)
        else:
            # Determine which post type we need
            cls = self.class_map.get(request.args.get('type', 'post'))
            post = cls()
            form_cls = model_form(cls,  exclude=('created_at', 'comments'))
            form = form_cls(request.form)
        context = {
            "post": post,
            "form": form,
            "create": slug is None
        }
        return context

    # ...
```

Update the `template/admin/base.html` file to create a new post drop down menu in the toolbar:

```
{% extends "base.html" %}

{%- block topbar -%}
<div class="topbar" data-dropdown="dropdown">
  <div class="fill">
    <div class="container">
      <h2>
           <a href="{{ url_for('admin.index') }}" class="brand">My Tumblelog Admin</a>
      </h2>
      <ul class="nav secondary-nav">
        <li class="menu">
          <a href="#" class="menu">Create new</a>
          <ul class="menu-dropdown">
            {% for type in ('post', 'video', 'image', 'quote') %}
                <li><a href="{{ url_for("admin.create", type=type) }}">{{ type|title }}</a></li>
            {% endfor %}
          </ul>
        </li>
      </ul>
    </div>
  </div>
</div>
{%- endblock -%}

{% block js_footer %}
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
  <script src="http://twitter.github.com/bootstrap/1.4.0/bootstrap-dropdown.js"></script>
{% endblock %}
```

Now you have a fully fledged tumbleblog using Flask and MongoEngine!



My Tumblelog Starring Flask, MongoDB and MongoEngine

MongoDB focus

MongoDB focuses on four main things: flexibility, power, speed, and ease of use. To that end, it ...

MongoDB

16:37 2011-12-23 | 0 comment

What is Mongo

What is MongoDB? | MongoDB from MongoDB on Vimeo.

16:17 2011-12-23 | 0 comment

Hello World!

Welcome to my new shiny Tumble log powered by MongoDB, MongoEngine and Flask

13:53 2011-12-20 | 1 comment

Additional Resources

The complete source code is available on Github: <https://github.com/rozza/flask-tumblelog>