# MongoDB and JSP

Table of Contents:

|  |  |  |
| --- | --- | --- |
| Sr No | Topic | Page |
| 0 | Preface | 1 |
| 1 | Introduction to MongoDB | 2 |
| 2 | MongoShell basic commands | 10 |
| 3 | Connecting to MongoDB using Java | 17 |
| 4 | CRUD operations in MongoDB using Java | 20 |
| 5 | Session Management in JSP using MongoDB | 36 |
| 6 | Aggregation Queries in MongoDB using Java | 47 |
| 7 | Web Analytics using Capped Collections | 59 |
| 8 | Storing Large Files using GridFS | 63 |
| 9 | Security and Performance | 71 |
| 10 | MongoDB UI tool- BMONGO and MonjaDB |  |

Preface

Before I present the content to you, there is no copyright on this content by me. Most of this content has been compiled from variety of sources from internet. So, you must read this book total free of cost. The contents aren’t directly copied at all and whenever they are the appropriat e source has been mentioned. This is not an attempt to take credit for somebody’s elses work, rather its a genuine attempt to provide some definite help for using MongoDB and JSP. I looked on web for books on this topic but couldn’t really find one, so this effort. I hope you appreciate the effort along with the intentions.

# Introduction to MongoDB

By the time I am writing this book, its been a hard work to find a book on MongoDB using Java or JSP and i have no idea why? Anyway, we wanted to create some tools/projects in mognodb and java hence we were looking for book, so we decided to write a compiled one ourselves.

Before we begin, I assume you have some background of building web apps using JSP, and you are interested in learning to develop JSP applications with MongoDB as data backend. In case you have never heard of MongoDB before, it is an open source, document-oriented database that supports the concept of flexible schema. In this chapter, we will learn what MongoDB is, and what do we gain from using MongoDB instead of trusted old SQL databases.

Lets talk briefly about the NoSQL databases (alternative database technologies to RDBM systems), the basics of MongoDB, and what distinguishes it from relational databases.

# What is JSP?

JavaServer Pages™ (JSP) is a web-scripting technology similar to Netscape server-side JavaScript (SSJS) or Microsoft Active Server Pages (ASP). However, it's more easily extensible than SSJS or ASP, and it isn't proprietary to any one vendor or any particular web server. Although the JSP specification has been managed by Sun Microsystems, any vendors can implement JSP in their own systems.

If you're working with Netscape technologies today or are planning to in the future, you should start learning about JSP, because Netscape will soon be introducing products that use it. They've already announced that the next version of Netscape Application Server (NAS) will use JSP as a presentation layer technology. In this article, I'll introduce you to JSP: its features, target users, and intended use. I'll compare JSP to some current Netscape technologies, such as SSJS and NAS's presentation markup language.

# JSP BASICS

JSP is a presentation layer technology that sits on top of a Java servlets model and makes working with HTML easier. Like SSJS, it allows you to mix static HTML content with server-side scripting to produce dynamic output. By default, JSP uses Java as its scripting language; however, the specification allows other languages to be used, just as ASP can use other languages (such as JavaScript and VBScript). While JSP with Java will be more flexible and robust than scripting platforms based on simpler languages like JavaScript and VBScript, Java also has a steeper learning curve than simple scripting languages.

To offer the best of both worlds - a robust web application platform and a simple, easy-to-use language and tool set - JSP provides a number of server-side tags that allow developers to perform most dynamic content operations without ever writing a single line of Java code. So developers who are only familiar with scripting, or even those who are simply HTML designers, can use JSP tags for generating simple output without having to learn Java. Advanced scripters or Java developers can also use the tags, or they can use the full Java language if they want to perform advanced operations in JSP pages.

# So what is this MongoDB?

Before we answer that lets talk about this No-SQL database concept, because the MongoDB which you are so much interested in knowing about is a NoSQL database.

# NoSQL Databases:

NoSQL (elaborated "Not only SQL"), is a data storage technology. It is a term used to collectively identify a number of database systems, which are fundamentally different from relational databases. NoSQL databases are increasingly being used in web 2.0 applications, social networking sites where the data is mostly user generated and dynamic. Because of their severe diverse nature, it is difficult to map user-generated content to a relational data model, the schema has to be kept as flexible as possible to reflect the changes in the content.

As the popularity of such a website grows, so does the amount of data and the read-write operations on the data. With a relational database system, dealing with these problems is very hard. The developers of the application and administrators of the database have to deal with the added complexity of scaling the database operations, while keeping its performance optimum. This is why more popular websites— Craigslist, Foursquare ,Facebook, Twitter to name a few—have adopted NoSQL databases to store part or all of their data.

These database systems have been developed with the goal of addressing such problems, and therefore are more suitable for such use cases. They are open source, freely available on the Internet, and their use is increasingly gaining momentum in consumer and enterprise applications.

## Different types of NoSQL databases

The NoSQL databases currently being used can be grouped into four broad categories:

• **Key-value data stores**: Data is stored as key-value pairs. Values are retrieved by keys.

Redis, Dynomite, and Voldemort are examples of such databases.

• **Column-based databases**: These databases organize the data in tables, similar to an

RDBMS, however, they store the content by columns instead of rows. They are good

for data warehousing applications. Examples of column-based databases are Hbase,

Cassandra, Hypertable, and so on.

• **Document-based databases**: Data is stored and organized as a collection of

documents. The documents are flexible; each document can have any number of

fields. Apache CouchDB and MongoDB are prominent document databases.

• **Graph-based data-stores**: These databases apply the computer science graph theory

for storing and retrieving data. They focus on interconnectivity of different parts

of data. Units of data are visualized as nodes and relationships among them are

defined by edges connecting the nodes. Neo4j is an example of such a database.

# Ask again, What is MongoDB?

* MongoDB falls into the group of document-oriented NoSQL databases.
* It is developed and maintained by 10gen (<http://www.10gen.com>).
* It is an open source database, written in the programming language C.
* The source code is licensed under AGPL and freely available at GitHub, anyone can download it from the repo <https://github.com/mongodb/mongo> and customize it to suit his/her needs.
* It is increasingly being used as a data storage layer in different kinds of applications, both web-based and nonweb-based.

## Why should one use MongoDB?

Features that make learning and using MongoDB a win, include:

* Easy to learn, at least easier than learning other NoSQL systems at least.

Column-oriented or graph-based databases introduce radical ideas that many

developers struggle to grasp. However, there is a lot of similarity in the basic

concepts of MongoDB and a relational database. Developers coming from an RDBMS

background, find little trouble adapting to MongoDB.

* It implements the idea of flexible schema. You don't have to define the structure

of the data before you start storing it, which makes it very suitable for storing nonstructured data.

* It is highly scalable. It comes with great features to help keep performance

optimum, while the size and traffic of data grows, with little or no change in the

application layer.

Furthermore it is free, it can be downloaded and used without charge. It has excellent documentation and an active and co-operative online community who participate in mailing lists, forums, and IRC chat rooms.

## Some popular users of MongoDB:

MetLife, SAP, The Guardian,Craigslist, Foursquare,eBay, SourceForge,NewYork Times etc. MongoDB is the most popular NoSQL database system.

# MongoDB concepts—Databases, collections, and documents

A MongoDB server hosts a number of databases. The databases act as containers of data

and they are independent of each other. A MongoDB database contains one or more

collections. For example, a database for a blogging application named myblogsite may

typically have the collections articles, authors, comments, categories, and so on.

**A collection** is a set of documents. It is logically analogous to the concept of a table in a

relational database. But unlike tables, you don't have to define the structure of the data

that is going to be stored in the collection beforehand.

**A document** stored in a collection is a unit of data. A document contains a set of fields or

key-value pairs. The keys are strings, the values can be of various types: strings, integers,

floats, timestamps, and so on. You can even store a document as the value of a field in

another document.

## Example of an MongoDB document:

Let's take a closer look at a MongoDB document. The following is an example of a document

that stores certain information about a user in a web application:

{

\_id : ObjectId("9eff39fa1ba3ava58146d651a"),

username : "ksjoshi88",

email : “ksjoshi88@gmail.com” ,

age : 25,

is\_admin : true,

created : "Mon Sep 30 2013 01:52:58 GMT+0700 (BDST)"

}

The previous document has six fields. If you have some JavaScript experience, you would

recognize the structure as JSON or JavaScript Object Notation.

The value for the first field,\_id, is autogenerated. MongoDB automatically generates an ObjectId for each document you create in a collection and assigns it as \_id for that document. This is also unique; that means no two documents in the same collection will have the same values for ID, just like aprimary key of a table in a relational database.

The next two fields, username and email are strings, age is an integer, and is\_admin is boolean. Finally, created is a JavaScript DateTime object, represented as a string.

## BSON: Data exchange format for MongoDB

When you store this document in the database, it is serialized into a special binary encoded format, known as BSON, short for binary JSON. BSON is the default data exchange format for MongoDB. The key advantage of BSON is that it is more efficient than conventional formats such as XML and JSON, both in terms of memory consumption and processing time.

Also, BSON supports all the data types supported by JSON (string, integer, double, Boolean, array, object, null) plus some special data types such as regular expression, object ID, date, binary data, and code. Programming languages such as PHP, Python, Java, and so on have libraries that manage conversion of language-specific data structures (for example, the associative array in PHP) to and from BSON. This enables the languages to easily communicate with MongoDB and manipulate the data in it.

# Analogy to relational databases:

|  |  |
| --- | --- |
| **RDBMS** | **MongoDB** |
| Database | Database |
| Table | Collection |
| Row | Document |
| Column | Field |

Task crate a graphical table for the document:

{

\_id: ObjectId(“45ght6723ok78”),

Name: “Kaustubh Joshi”,

Gender: “Male”,

Country: “India”

}

## Differences between RDBMS and MongoDB:

So much for the similarities: now let's talk briefly about the differences. The key thing that

distinguishes MongoDB from a relational model is the absence of relationship constraints.

There are no foreign keys in a collection and as a result there are no JOIN queries. Constraint

management is typically handled in the application layer. Also, because of its flexible schema

property, there is no expensive ALTER TABLE statement in MongoDB.

# How to Install MongoDB?

Well, the answer is simple. Just Google it. But these are the popular steps to install it on windows.

1. Go to <http://www.mongodb.org/downloads>
2. Select the type of download appropriate for your operating system:



1. Download the MongoDB to your machine. Visit <http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/>

for installing it on Windows.

1. Once the download is finished, move the ZIP archive to the C:\ drive and extract

it. Rename the extracted folder (mongodb-win32-i386-x.y.z where x.y.z isthe version number) to mongodb.

1. Create the folder C:\data\db.
2. Start MongoDB

To start MongoDB, execute from the *Command Prompt*:

**C:\mongodb\bin\mongod.exe**

1. This will start the main MongoDB database process. The waiting for connections message in the console output indicates that the mongod.exe process is running successfully.
2. Connect to MongoDB using the [mongo.exe](http://docs.mongodb.org/manual/reference/program/mongo/#bin.mongo) shell. Open another *Command Prompt* and issue the following command:

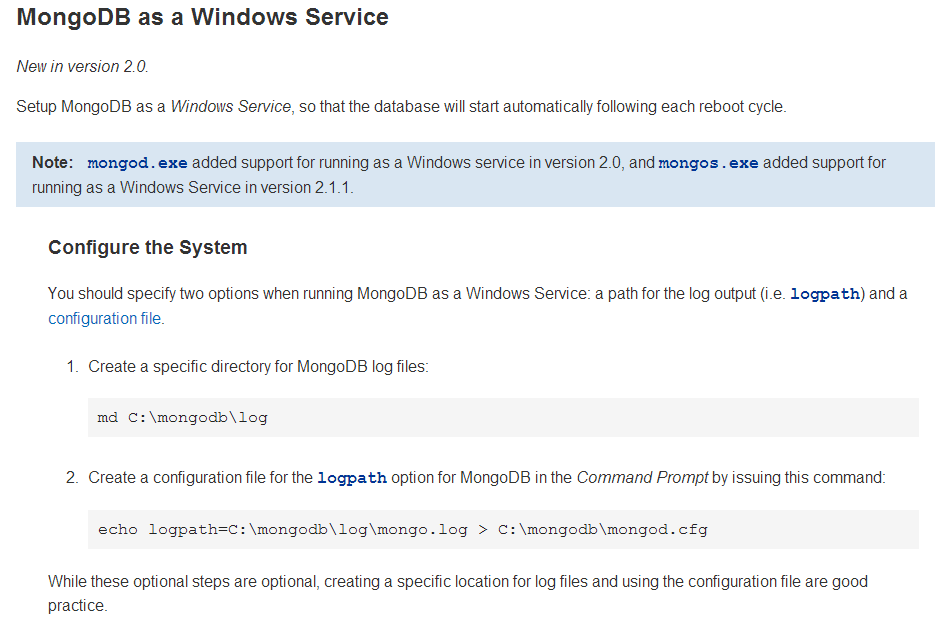
**C:\mongodb\bin\mongo.exe**

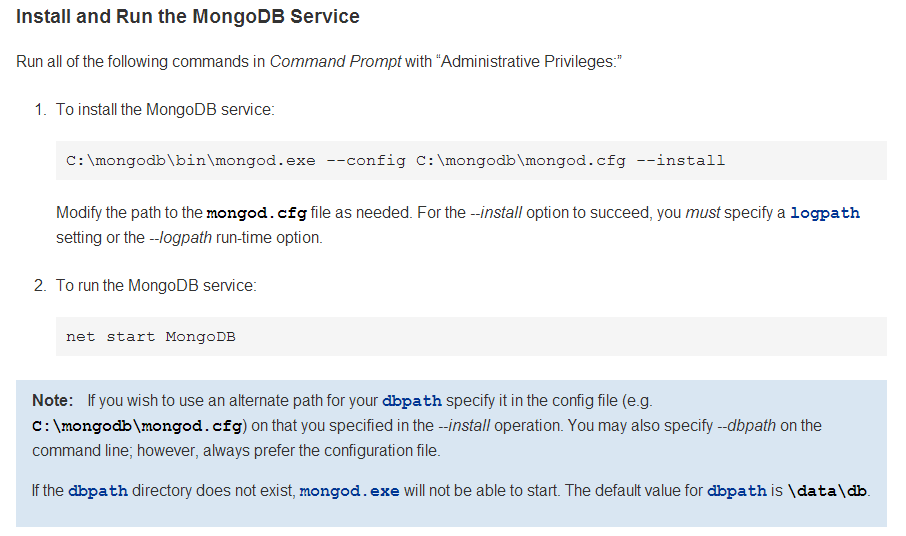
1. The [mongo.exe](http://docs.mongodb.org/manual/reference/program/mongo/#bin.mongo) shell will connect to [mongod.exe](http://docs.mongodb.org/manual/reference/program/mongod.exe/#bin.mongod.exe) running on the localhost interface and port 27017 by default. At the[mongo.exe](http://docs.mongodb.org/manual/reference/program/mongo/#bin.mongo) prompt, issue the following two commands to insert a record in the test [collection](http://docs.mongodb.org/manual/reference/glossary/#term-collection) of the default test database and then retrieve that record:

**db.test.save( { a: 1 } )**

**db.test.find()**

## MongoDB as windows service:





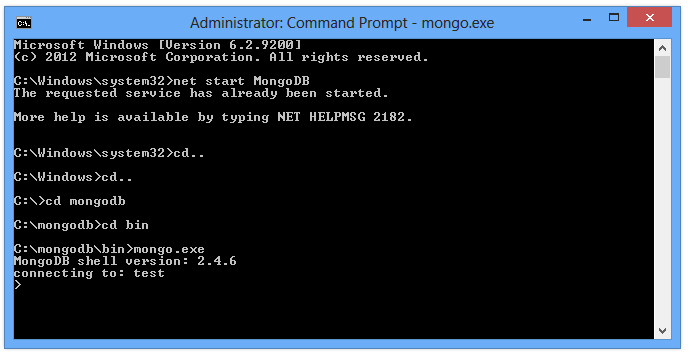
### Stop or Remove the MongoDB Service

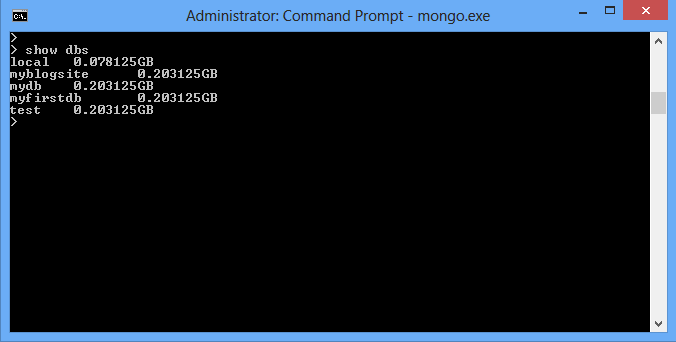
* To stop the MongoDB service:
* net stop MongoDB
* To remove the MongoDB service:

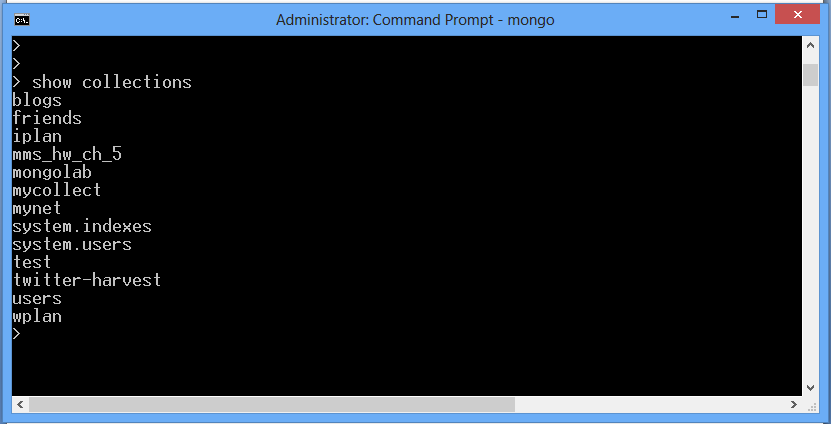
C:\mongodb\bin\mongod.exe --remove

# MongoShell Basic Commands

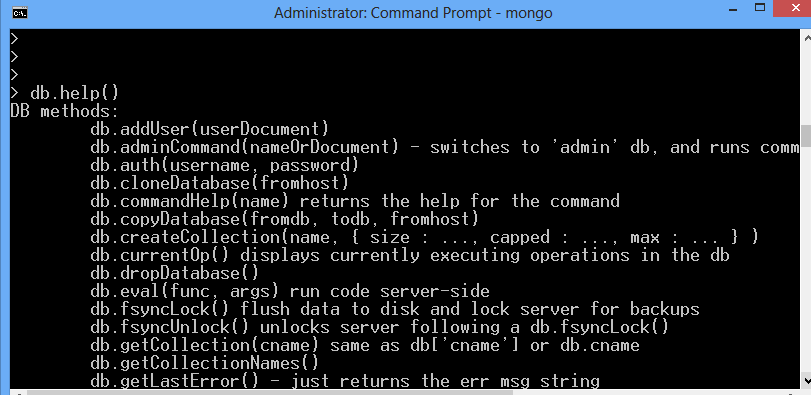
The [mongo](http://docs.mongodb.org/v2.2/reference/mongo/#bin.mongo) shell is an interactive JavaScript shell for MongoDB, and is part of all [MongoDB distributions](http://www.mongodb.org/downloads). This section provides an introduction to the shell, and outlines key functions, operations, and use of the [mongo](http://docs.mongodb.org/v2.2/reference/mongo/#bin.mongo) shell.

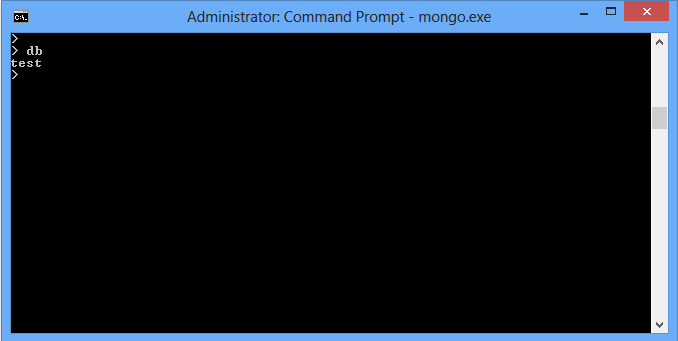


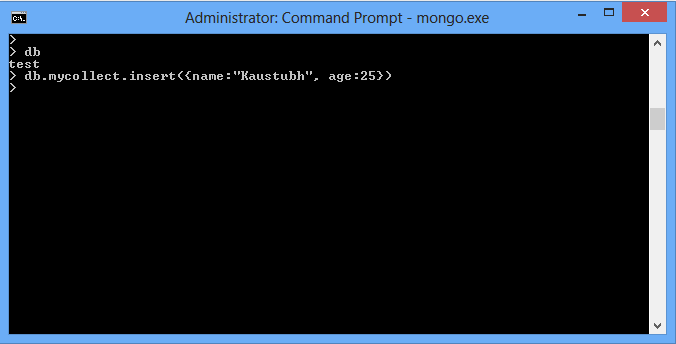


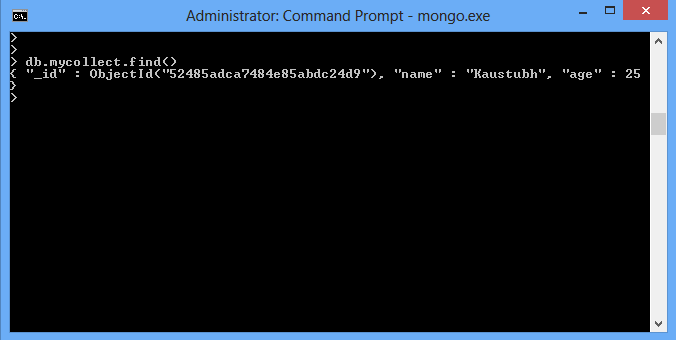


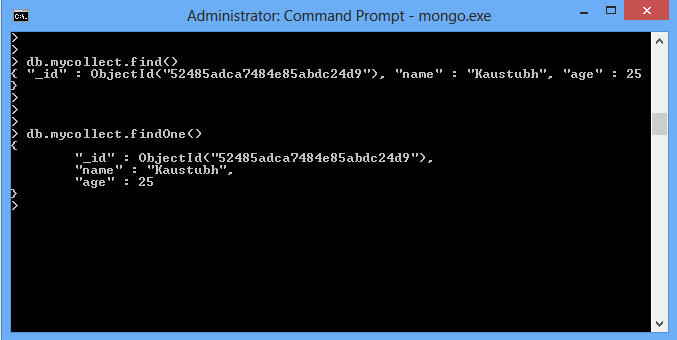


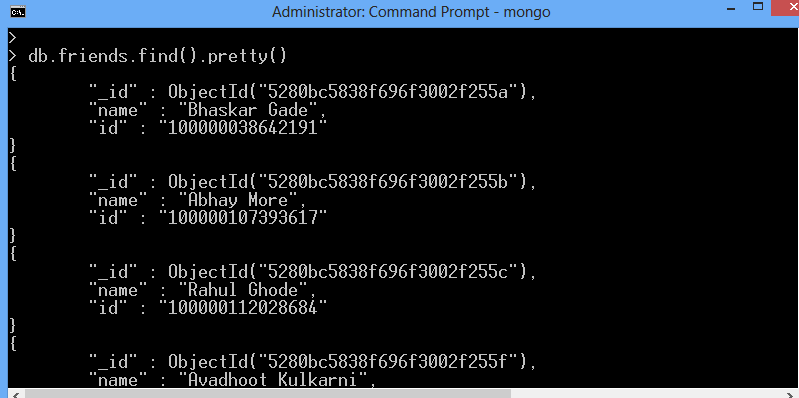


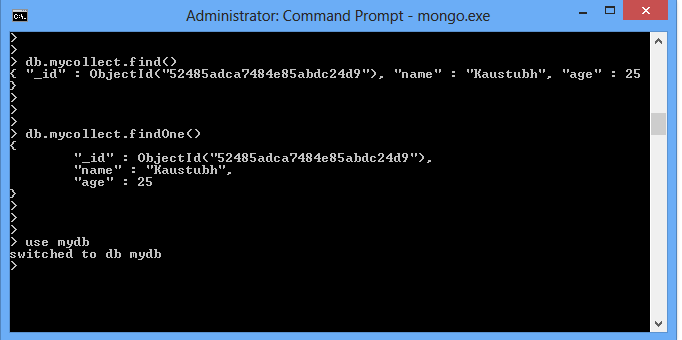


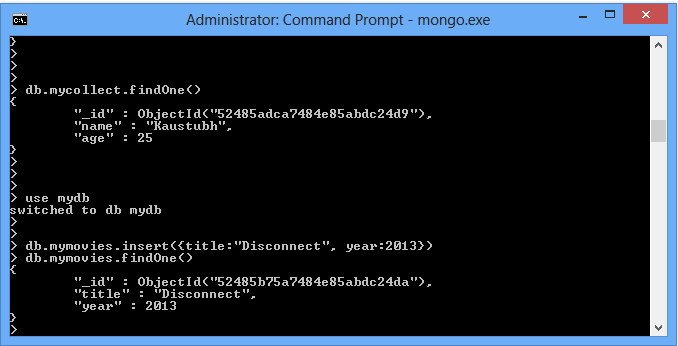


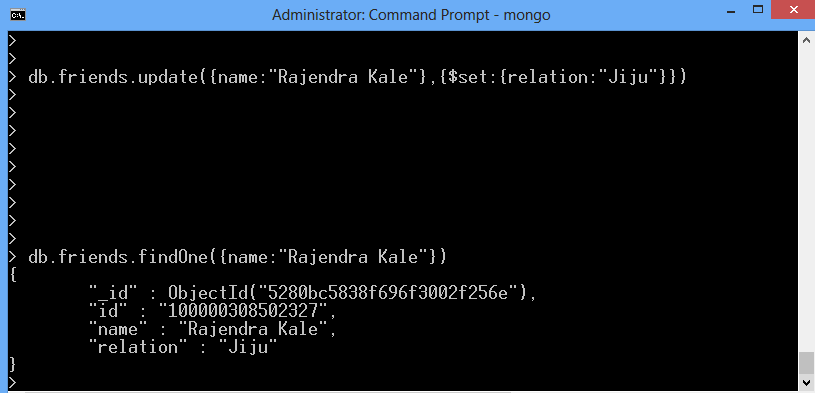


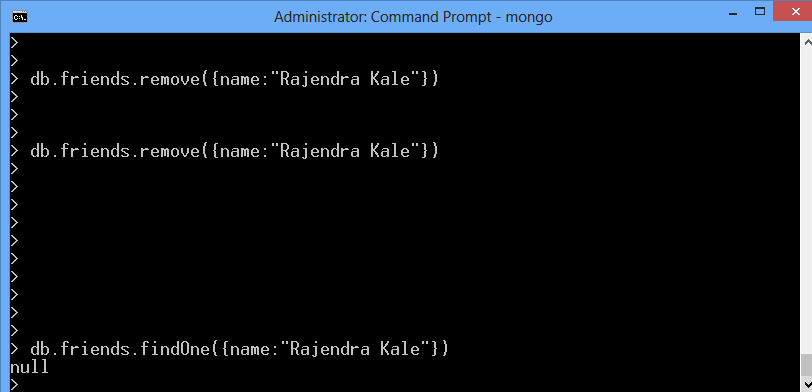












# How to install mongodb Java driver

Step 1)

Download the latest java driver from:

<http://central.maven.org/maven2/org/mongodb/mongo-java-driver/>

Step 2)

You need to add the jar to your project's classpath in order to use it. These are not executable jars. Take a look at the getting started docs here:

<http://docs.mongodb.org/ecosystem/tutorial/getting-started-with-java-driver/#getting-started-with-java-driver>

There is no "installing" the driver. You just add it as a dependency in your project and you use it in your code.

# Connecting to MongoDB in JSP

After installing the required jar, start a new dynamic web project in eclipse. Create a JSP file connect.jsp and write the following code in it.

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*

import=*"com.mongodb.DB"*

import=*"com.mongodb.DBCollection"*

import=*"com.mongodb.DBCursor"*

import=*"com.mongodb.ServerAddress"*

import=*"com.mongodb.DBObject"*

import=*"com.mongodb.BasicDBObject"*

import=*"com.mongodb.WriteConcern"*

import=*"com.mongodb.Mongo"*

import=*"com.mongodb.MongoException"*

import=*"java.util.Arrays"*

import=*"java.util.\*"*

%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

<%

Mongo mongo=**new** Mongo("Localhost",27017); //to establish a connection

DB db=mongo.getDB("test"); // select the databse test

Set<String> colls=db.getCollectionNames(); //get all the collection names in "test"

**for**(String s:colls)

System.out.println(s); //printing the collection names on to the console

%>

</body>

</html>

## Code Explanation:

The Mongo class from the package com.mongodb.Mongo, represents a database connection with internal connection pooling. For most applications, you should have one Mongo instance for the entire JVM.

The following creations of Mongo instance are equivalent:

**Mongo m1=new Mongo( );**

**Mongo m2=new Mongo(“localhost”);**

**Mongo m3=new Mongo(“localhost”,27017);**

**Mongo m4=new Mongo(new ServerAddress(“localhost”));**

You can also connect to a replica set here by passing a ServerAddress list to the Mongo constructor, like this:

**Mongo repmongo=new Mongo(Arrays.asList( new ServerAddress(“localhost”,27017), new ServerAddress(“localhost”,27018), new ServerAddress(“localhost”,27019));**

You can also connect to a sharded cluster using the same constructor. We will possibly talk more about replication and sharding in due course of time. The next line of code:

**DB db=mongo.getDB("test");**

getDB( ) here, returns a database object, which is of type DB. For that you must pass the database name as string, remember if the database does not exist it will create one. DB is an abstract class which represents a logical database on the server. db.getCollectionNames( ), returns a set containing the names of all collections in this database.

# CRUD operations in MongoDB using JSP

### Let’s now study in detail how to use Java to manipulate data in mongodb.

## Inserting a Document

Once you have the collection object, you can insert documents into the collection. For example, lets make a little document that in JSON would be represented as:

{

"name" : "Sahil",

"type" : "student",

"rn" : 28,

"info" : {

maths : 99,

english : 98

}

}

Notice that the above has an “inner” document embedded within it. To do this, we can use the [BasicDBObject](http://api.mongodb.org/java/current/com/mongodb/BasicDBObject.html) class to create the document (including the inner document), and then just simply insert it into the collection using the insert() method.

BasicDBObject doc = **new** BasicDBObject("name", "Sahil").

append("type", "student").

append("rn", 28).

append("info", **new** BasicDBObject("maths", 99).append("english", 98));

coll.insert(doc);

## The First web Application

For our first MongoDB powered web application, we are going to build a very simple blogging website. The reason I chose to build a blog as our first example application is because it is a CRUD (Create, Read, Update, Delete) application and it is very suitable in our case to ease into JSP and MongoDB web development. We will build plain user interfaces using HTML/CSS with simple textboxes and buttons. A MongoDB database will store all the content. JSP will take care of moving the data back and forth between the frontend and the database. By building tools to create/read/update/delete articles in the blog site, we will learn how to save, read, or remove documents in MongoDB.

First things first, get your mongodb server running.

## CREATING a new blog:

Here we will be building a UI to create a blog and then put into mongodb collection in the form of a document (record). For UI: create a simple HTML/CSS page as shown below in the diagram. ( Instead you can also download the source code for the book from bbarters.com)



Now, we need to make sure when the form is submitted, the content of the HTML form goes into the mongodb collection. For that we will write a new JSP page, called as **insert.jsp**

<body>

<%

//this is the code to get the form contents from createBlog.jsp

String blogTitle=request.getParameter("title");

out.println(blogTitle);

//This is the code to insert the blog details into mongo collection

Mongo mg = new Mongo("Localhost",27017);

DB db = mg.getDB("javablogs");

DBCollection collection = db.getCollection("articles");

BasicDBObject doc = new BasicDBObject();

doc = new BasicDBObject();

doc.append("Title",request.getParameter("title"));

doc.append("Content",request.getParameter("content"));

Date cdate=new Date();

doc.append("CreatedAt",cdate);

collection.insert(doc);

mg.close();

response.sendRedirect("viewBlogs.jsp");

%>

</body>

## Explanation:

The code above is nothing but the C part of the CRUD operations we are discussing in this chapter. The first three lines are self-explanatory, we are connecting to mongodb hosted on a local machine with standard port 27017, then we are selecting the Database “javablogs” and collection “articles”. Remember if the database doesnt exist, it will be created for you, same goes for the collection.

Now we proceed to the code where we take the content from html form and put it into articles collections as a new document. For that we need a BasicDBObject, this class is a basic implementation of BSON object that is mongo specific. It can be created as follows:

**BasicDBObject obj=new BasicDBObject( );**

**obj.append(“name”,”Sahil”);**

The append method adds a key/value pair to the object. So, in short we will create a new MongoDB document using number of key/value pairs in this BasicDBObject object. And then

**collection.insert(doc);**

and then we will use the insert method of DBCollection class to save the document in the mongodb collection. We can optinally close the connection so as to release the database resources.

## Counting documents in the collection

We can count how many documents we have in a collection using the getCount() method.

System.out.println(collection.getCount());

# Reading the documents from the mongodb collection

In order to get all the documents in the collection, we will use the find() method. The find() method returns a **DBCursor** object which allows us to iterate over the set of documents that matched our query. So to query all of the documents and print them out :

DBCursor cursor = collection.find();

**try** {

**while**(cursor.hasNext()) {

System.out.println(cursor.next());

}

} **finally** {

cursor.close();}

## Getting a Single Document with A Query

We can create a query to pass to the find() method to get a subset of the documents in our collection. For example, if we wanted to find the document for which the value of the “i” field is 71, we would do the following:

BasicDBObject query = **new** BasicDBObject("i", 71);

cursor = collection.find(query);

**try** {

**while**(cursor.hasNext()) {

System.out.println(cursor.next());

}

} **finally** {

cursor.close();

}

You may commonly see examples and documentation in MongoDB which use $ Operators, such as this: ($ne => not equal to, $get => greater than)

db.things.find({j: {$ne: 3}, k: {$gt: 10} });

These are represented as regular String keys in the Java driver, using embedded DBObjects:

BasicDBObject query = **new** BasicDBObject("j", **new** BasicDBObject("$ne", 3)).append("k", **new** BasicDBObject("$gt", 10));

cursor = collection.find(query);

**try** {

**while**(cursor.hasNext()) {

System.out.println(cursor.next());

}

} **finally** {

cursor.close();}

## Getting a Set of Documents with a Query

We can use the query to get a set of documents from our collection. For example, if we wanted to get all documents where "i"> 50, we could write:

query = **new** BasicDBObject("i", **new** BasicDBObject("$gt", 50));

*// e.g. find all where i > 50*

cursor = collection.find(query);

We could also get a range, say 20 < i <= 30:

query = **new** BasicDBObject("i", **new** BasicDBObject("$gt", 20).

append ("$lte", 30));

*// i.e. 20 < i <= 30*

cursor = coll.find(query);

**while**(cursor.hasNext()) {

System.out.println(cursor.next());

}

Now that we have successfully put the documents into the collection in mongodb its time to view the blog. That way we will also be able to confirm if the blog got stored inside the mongodb at first place or not? As the last line of code in insert.jsp we redirect the user to **viewBlogs.jsp** page. Lets have a look how this viewBlogs.jsp looks:



Ok, now let’s have a look at the code which has produced this not so nice looking UI.



## Code Explanation:

Again the first few lines are pretty self-explanatory, I guess these lines are getting repeated all over again on every page, so we will have to do something about it soon. Anyway, we connect to a mongodb instance on a local machine on standard port, then we go on to select the “Javablogs” database and “articles” collection.

We use DBCursor to iterate through the documents in the articles collection. This class is an iterator over database results. Doing a find( ) query on any collection returns a DBCursor object.

**DBCursor cursor=collection.find( );**

The DBCursor class has some important methods like:

**hasNext() –** checks if there is another document in the cursor, returns true if does

**next() –** returns DBObject, the document the cursor is at and moves the cursor ahead by one

Once we get a individual document using next method on cursor, we will store that in DBObject str, and after that we will use get( ) method on that DBObject instance to get the individual values of a field, remember to typecast the values to a specific java type before using it.

So, for example you want to read the value of “Name” column in a document returned by the DBCursor next() method, you will write:  
**String data=(String)str.get(“Name”)** // str here is a DBObject instance

In order to read all documents returned by a cursor we will of course run the loop till cursor.hasNext( ) method keeps returning true.

# Updating a document in MongoDB using JSP

Let’s now try and update the documents inserted in the mongodb. Here in our context we will try to update the blog that we have saved in mongodb. We will create a new JSP page editBlog.jsp.

The link for this page has been given on viewBlogs.jsp page as discussed earlier. Lets see how this editBlog.jsp looks like:



Let’s now take a look at the code for editBlog.jsp but before we do that lets just discuss the way you can update a document in mongodb and by using Java as a language.

The sample data:

{

"company" : "SEED",

"type" : "training",

"clients" : 1000

},

{

"company" : "Accenture",

"type" : "Outsourcing",

"clients" : 25000

},

{

"company" : "Bbarters",

"type" : "Social Sharing",

"clients" : 900

}

## DBCollection.update() with $set

Find document where company = ‘Bbarters’, and update it’s clients values from 900 to 9000.

BasicDBObject newDocument = **new** BasicDBObject();

newDocument.put("clients", 9000);

BasicDBObject searchQuery = **new** BasicDBObject().append("company", "Bbarters");

  collection.update(searchQuery, newDocument);

**The document is replaced!?**  
Wait, the entire “company” document is replaced with another new document, this is not what we want.To update a particular value only, uses $set update modifier.

BasicDBObject newDocument = **new** BasicDBObject();

newDocument.append("$set", **new** BasicDBObject().append("clients", 9000));

BasicDBObject searchQuery = **new** BasicDBObject().append("company", "Bbarters");

collection.update(searchQuery, newDocument);

## 2. DBCollection.update() with $inc

This example show the use of $inc modifier to increase a particular value. Find document where company = ‘Bbarters’, update it’s ‘clients’ value by increasing the value from 9000 to 9199, (9000 + 199) = 9199.

BasicDBObject newDocument = **new** BasicDBObject().append("$inc",

**new** BasicDBObject().append("total clients", 199));

collection.update(**new** BasicDBObject().append("company", "Bbarters"), newDocument);

## 3. DBCollection.update() with multi

This example show the use of multi parameter to update a set of matched documents. Find document where type = ‘social sharing’, update all the matched documents’ ‘clients’ value to 888.

BasicDBObject updateQuery = **new** BasicDBObject();

updateQuery.append("$set", **new** BasicDBObject().append("clients", "888"));

BasicDBObject searchQuery = **new** BasicDBObject();

searchQuery.append("type", "social sharing");

collection.updateMulti(searchQuery, updateQuery);

**Note**  
If update without the multi set to true.

BasicDBObject updateQuery = **new** BasicDBObject();

updateQuery.append("$set", **new** BasicDBObject().append("clients", "888"));

BasicDBObject searchQuery = **new** BasicDBObject();

searchQuery.append("type", "social sharing");

collection.update(searchQuery, updateQuery);

You should notice that only the first matched document is updated. To update a set of matched documents, you need to set “multi” to true.

Now that we have covered the basics of updating methods in mongodb using java, lets proceed to the logic of editBlog.jsp



Again the first few lines are trivial by now, establishing the connection and selecting the database and collection. Now via querystring we have passed the value of \_id field in a document. The \_id field is stored as type ObjectId in mongodb the java equivalent to this is: ObjectId class.

So,we will create an instance of ObjectId class and will use the passed on value from request to instantiate it. Once we do that, we will build our query’s, where part using this newly created ObjectId instance.

**BasicDBObject query=new BasicDBObject();**

**query.put("\_id",myid);**

**This is similar to:**

**where \_id=myid; // in SQL update query**

After this we will create a new document , or in other words the updated version of the doucment. We do this using thie piece of code:

**BasicDBObject doc = new BasicDBObject();**

**doc = new BasicDBObject();**

**doc.append("Title",request.getParameter("title"));**

**doc.append("Content",request.getParameter("content"));**

**Date cdate=new Date();**

**doc.append("CreatedAt",cdate);**

Then we build the update command using this code, here we use the $set directive which is equivalent to set directive in update query in SQL:

**BasicDBObject updateObj = new BasicDBObject();**

**updateObj.put("$set", doc);**

After we do this, finally we will fire the update query on the mongodb collection:

**collection.update(query,updateObj,false,true);**

here the first parameter is the where part, second is the updation part, third parameter is upsert set to false, which means if the record doesnt exists, do not insert it. The last parameter is for multiple updates that is if the query matches more than one documents shall we update them all or not, it is set to true here.

## Deleting documents from MongoDB collection:

In this section, we show you how to use collection.remove() to delete documents from the collection.

## DBCollection.remove()

Get first document and delete it.

DBObject doc = collection.findOne(); *//get first document*

collection.remove(doc);

Puts query in a BasicDBObject. In this case, type = “Outsourcing” is deleted.

BasicDBObject document = **new** BasicDBObject();

document.put("type", “Outsourcing”);

collection.remove(document);

And Operator?

A query like this only delete type = “training”.

BasicDBObject document = **new** BasicDBObject();

document.put("type", “training”);

document.put("type", “Outsourcing”);

collection.remove(document);

use BasicDBObject directly. In this case, type = “training” is deleted.

collection.remove(**new** BasicDBObject().append("type", “training”));

put a $in operator in a BasicDBObject object, constructs the query in ArrayList. In this case, type=”training” and type=”Outsourcing” are deleted.

BasicDBObject query2 = **new** BasicDBObject();

List<String> list = **new** ArrayList<String>();

list.add(“training”);

list.add(“Outsourcing”);

query2.put("type", **new** BasicDBObject("$in", list));

collection.remove(query2);

Use cursor to delete all available documents.

DBCursor cursor = collection.find();

**while** (cursor.hasNext()) {

collection.remove(cursor.next());

}

Pass an empty BasicDBObject, and the entire documents will be deleted.

collection.remove(**new** BasicDBObject());

It deletes the entire documents and drop the collection.

collection.drop();

The remove() will returns a WrireResult object, it contains useful information about the remove operation. And you can uses getN() to get the number of documents affected.

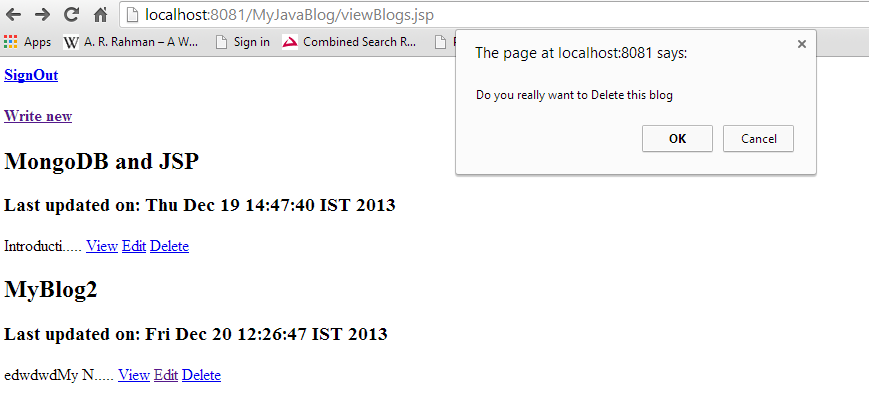
WriteResult result = collection.remove(query2);

System.out.println("Number of documents are deleted : " + result.getN());

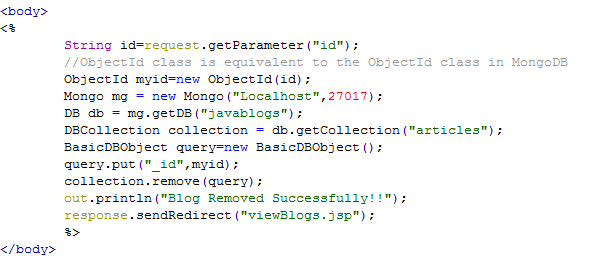
Now that we have seen the basics of Delete operation in mongoDB using JSP, lets write the functionality of deleting a blog in our blogging application.

For that we have already provided a link on **viewBlogs.jsp**, now lets have a look what happens when we click delete link:

Remember you can download the source code for this book for free at bbarters.com.



As you can see in case of delete operation, we have shown a javascript confirm box just to make sure that the user really wants to delete the blog and he/she hasn’t clicked the delete link accidently. Once user confirms the delete operation by cliking OK in confirm box, we will execute the code written in **deleteBlog.jsp:**



If you look at this code, like in case of updation, we are passing the value of \_id field via querystring which will be used for our query clause ( where condition).We build our query using these statements:

**BasicDBObject query=new BasicDBObject();**

**query.put("\_id",myid);**

Now using this query we will perform the remove operation:

**collection.remove(query);**

# Session Mangement, Authentication and Authorization

In this chapter we will discuss how to implement the session, how to create users and authenticate them. Also how to make sure only authorized users can view some of the administration specific pages in our blogging web application.

### Creating a Singleton class for MongoDB connectivity

But before we move on to those topics, as I mentioned in the last chapter we were quite clearly repeating the database connectivity code in just about every page that we created.

So now as our database is fixed(“javablogs”), server is fixed ( localhost) also the standard port number is also the same, why create and destroy a new database connection on every JSP page?

Instaed we can have only one instance of Mongo alive in lifetime of the web application, we can very easily achieve this using dconnect.java file.

dconnect is a singleton java class which is used to connect to mongodb on every page from now-onwards. So, lets have a look at this singleton class dconnect:

package myconnection.test;

import com.mongodb.\*;

import java.net.UnknownHostException;

public class dconnect

{

final String HOST="localhost";

final int PORT=27017;

final String DBNAME="javablogs";

public static dconnect instance;

public Mongo connection;

public DB database;

private dconnect() throws UnknownHostException

{

this.connection=new Mongo(this.HOST,this.PORT);

this.database=this.connection.getDB(this.DBNAME);

}

public static dconnect createInstance() throws UnknownHostException

{

if(dconnect.instance==null)

{

dconnect.instance=new dconnect();

}

return dconnect.instance;

}

public DBCollection getCollection(String name)

{

return this.database.getCollection(name);

}

}

### Code Explanation:

As with any singleton pattern logic, the constructor is private, there is one static data memeber which keeps track of the fact that if there is any instance already alive. We have addded some useful properties like HOST, PORT, DBNAME for ourseleves.

The ***instance*** data member is static and also of type dconnect only, if this static instance exists then a new instance will never get created.

The constructor is private and it does its job of initializing the data members. The constructor can’t be called from outside, so only way to create an object is to call the static method createInstance(). This method checks if any instance already exists, if yes then returns it else creates a new one and returns it.

We also have a public method getCollection(string), this method will be used to access a specific collection from the database, it returns an object of type DBConnection.

## Session Handling basics in JSP

Session Handling becomes mandatory when a requested data need to be sustained for further use. Since http protocol considers every request as a new one, session handling becomes important.

form1.jsp:

<html>

<body>

<form method = "post" action="form2.jsp">

<font>Username<input type = "text" name = "name"></font>gt;

</font><br><br>

<input type = "submit" name = "submit" value = "submit" >

</form>

</body>

</html>

form2.jsp

<%

String name = request.getParameter("name");

if((name!=null))

{

session.setAttribute("username",name);

}

%>

<a href="form3.jsp">Continue</a>

**form3.jsp**

<html>

<head>

<title>Welcome to session continued Page</title>

</head>

<body>

<font>Welcome</font> <%= session.getAttribute("username") %>

</body>

</html>

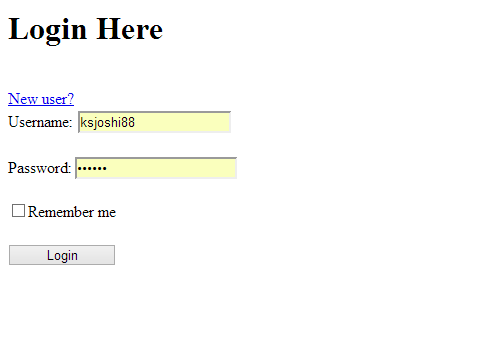
Following are some of the methods to handle session.

* In JSP whenever a request arises the server generates a unique Session ID which is stored in the client machine.
* Cookies store the information in the client browser
* URL rewriting the session information is appended to the end of the URL
* Hidden form fields the sessionID is embedded to GET and POST command.

Now that we know how to use basic session handling, lets create a user session in our blogging application but before we do that, lets also add the authentication functionality to our system.

## Adding Login Screen

Lets first create a new JSP page **login.jsp**, it looks like this:



Now let’s look at the java logic in this **login.jsp** page



### Code Explanation:

Now if you see, instead of repeating the connectivity code all over again, we have used the dconnect singleton class, to not only establish the database connection but also select the appropriate collection in the database.

**dconnect mymongo=dconnect.createInstance();**

**DBCollection collection=mymongo.getCollection("users");**

Next we take the username and password from the form, using request.getParameter( ), in here we are submitting the login form to the same page.

**String uname=request.getParameter("username");**

**String pwd=request.getParameter("password");**

Now in the following code we check whether there exists a user in the “users” collection with the values of given username and password.

BasicDBObject query=**new** BasicDBObject();

query.put("\_id",uname);

query.put("pwd",pwd);

BasicDBObject where=**new** BasicDBObject();

where.put("$and",query);

DBObject doc = collection.findOne(query);

**if**(doc!=**null**)

{

session.setAttribute("id",request.getParameter("username"));

response.sendRedirect("createBlog.jsp");

}

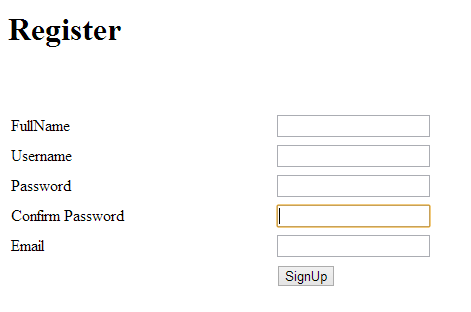
**else**

{

out.println("\nInvalid Credentials");

}

If the user is found we will redirect him to **createBlog.jsp** else we will print the message “ Invalid Credentials”. Ofcousre while giving the authentication facility you also have to add a way for new users to be created, so will will create a new page **signup.jsp**, it looks like this:



Remember you can download the source code for this for free on bbarters.com.

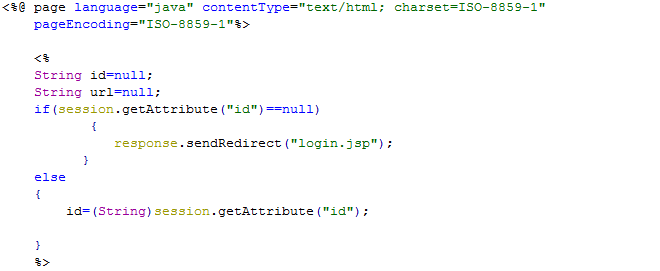
If we look at the logic of creating a new user in **signup.jsp** :



Here the code is pretty much self-explanatory, we are taking form values and using them forming a BasicDBObject instance, which we will insert as document inside a mongoDB collection “users” in database “javablogs”. Please download the entire source code for this book for free on bbarters.com to enhance it and create your own web application using jsp and Mongodb.

## Maintaining user session

Now that we have everything in place its finally time to put session data in action, unless the user has logged in he won’t be allowed to visit any page, and just in case he/she does they will be redirected to Login screen. The following code at the top of all JSP pages takes care of that:



The code again is very straightforward we are carrying the userid/username as session data in between pages and if that data is not found, the user is redirected to the login page.

# Performing Simple Authorization:

In this section we will create a JSP page which is only meant to be visited by a user with admin privileges. While creating a user if his/her username is admin that user will be admin user.

Of course you don’t have to implement it this way, you can always have your own custom logic to make sure that one/more users will have admin privileges. So we create this special page **adminBoard.jsp**, and there is code on this page which makes sure that only the authenticated user whose username is “admin” can view this page.



Again there is hardly any need to explain this code, if the user is authenticated i.e the session data for username is found then we check if the username is “admin” , if yes then we let user view this page else we redirect him/her to **createBlog.jsp** and he/she cant view **adminBoard.jsp**



Let’s look at the code for pagination:

<%! **double** get\_PAGES\_COUNT(String id)

{

dconnect mongo1=**null**;

**try**

{

mongo1=dconnect.createInstance();

}

**catch**(UnknownHostException e)

{

e.printStackTrace();

}

DBCollection collection=mongo1.getCollection("articles");

BasicDBObject query = **new** BasicDBObject("Author",id);

DBCursor cursor = collection.find(query);

**int** nrecords=cursor.count();

**return**(Math.ceil((**double**)nrecords/5));

}

%>

<%

**double** pages = get\_PAGES\_COUNT(id);

dconnect mymongo=dconnect.createInstance();

DBCollection collection=mymongo.getCollection("articles");

BasicDBObject query = **new** BasicDBObject("Author",id);

**int** skip=0;

**if**(request.getParameter("p")!=**null**)

{

String c = request.getParameter("p");

skip = Integer.parseInt(c);

}

DBCursor cursor=**null**;

**if**(skip==0)

cursor = collection.find(query).limit(3);

**else**

{

cursor = collection.find(query).skip((skip-1)\*3).limit(3);

}

**try** {

**while**(cursor.hasNext())

{

DBObject str= cursor.next();

%>

<h2><%out.println(str.get("Title"));%></h2>

<h3><% Date cdate=(Date)str.get("CreatedAt");

out.println("Last updated on: "+cdate.toString());%></h3>

<% String data=(String)str.get("Content");

out.println(data.substring(0,10)+".....");

%>

<a href=*"readBlog.jsp?id=*<%out.println(str.get("\_id"));%>*"*>View</a>

<a href=*"editBlog.jsp?id=*<%out.println(str.get("\_id"));%>*"*>Edit</a>

<a href=*"deleteBlog.jsp?id=*<%out.println(str.get("\_id"));%>*"* onclick="return confirmDelete()">Delete</a>

<%

}

}

**finally**

{

cursor.close();

}

%>

<div>

</br>

</br>

<% **for**(**double** i=0;i<pages;i++)

{

**int** j=(**int**)i;

%>

<a href=*"viewBlogs.jsp?p=*<%out.print(j+1);%>*"*><%out.print(j+1);%></a>

&nbsp;&nbsp;

<%} %>

</div>

</body>

## Code Explanation:

<%! **double get\_PAGES\_COUNT(String id)**

**{**

**dconnect mongo1=null;**

**try**

**{**

**mongo1=dconnect.createInstance();**

**}**

**catch(UnknownHostException e)**

**{**

**e.printStackTrace();**

**}**

**DBCollection collection=mongo1.getCollection("articles");**

**BasicDBObject query = new BasicDBObject("Author",id);**

**DBCursor cursor = collection.find(query);**

**int nrecords=cursor.count();**

**return(Math.ceil((double)nrecords/5));**

**}**

%>

Here a function double get\_PAGES\_COUNT(String id) is used with id as a parameter to it. The DBCursor cursor is iterated and we get the number of pages to be shown by returning –

**Math.ceil((double)nrecords/5));**

**double pages = get\_PAGES\_COUNT(id);**

Here we call the function and store the value in pages variable.

**dconnect mymongo=dconnect.createInstance();**

**DBCollection collection=mymongo.getCollection("articles");**

**BasicDBObject query = new BasicDBObject("Author",id);**

We connect to the Mongo driver with required database and collection respectively.

**int skip=0;**

**if(request.getParameter("p")!=null)**

**{**

**String c = request.getParameter("p");**

**skip = Integer.parseInt(c);**

**}**Then initialise a variable skip, and take the value from the passed parameter which will determine the number of records to be skipped in each iteration.

**DBCursor cursor=null;**

**if(skip==0)**

**{**

**cursor = collection.find(query).limit(3);**

**}**

**else**

**{**

**cursor = collection.find(query).skip((skip-1)\*3).limit(3);**

**}**Now we fire the query find on the collection with the skip and limit parameters which here is – limit(3), which will fetch only 3 records.  
 - skip((skip-1)\*3), skip will skip the no of records specified before fetching the records.  
Note:

if skip=1 ie the first page then skip((1-1)\*3) will skip 0 records.  
 if skip=2 ie the second page then skip((2-1)\*3) will skip 3 records.  
 if skip=3 ie the second page then skip((3-1)\*3) will skip 6 records

# Aggregation Queries in MongoDB using Java

Aggregation queries are different from the queries we have done so far. These queries perform certain computations/calculations on the documents and the result of the computation is sent back to the user. For instance, grouping the documents on the value of a certain field and counting their values is a kind of aggregation operation. If you have experience with SQL, aggregation queries are the ones that use COUNT/SUM/AVG/GROUP BY statements.

MongoDB uses MapReduce, a functional programming paradigm to perform aggregation and batch processing of data. In this chapter, we are going to familiarize ourselves with the concepts of MapReduce. We will learn how to perform MapReduce queries in JSP. We will also learn to use utility functions (group(), distinct(), count(), and so on) in MongoDB that are used for aggregation.

## Java Driver and Aggregation Framework

Release 2.2.0 of MongoDB introduces the aggregation framework. Designed to be both performant and easy to use, the aggregation framework calculates aggregate values, (such as counts, totals and averages), without the need for complex map-reduce operations. The aggregation framework is both multithreaded and written in C++, thus it executes natively across nodes.

Aggregation tasks are built around the concept of the aggregation pipeline. Just as UNIX-like shells use the pipe operator | to connect a series of command-line operations together, the aggregation framework passes documents through a pipeline of operations which transform these objects as they go. Version 2.9.0 of the Java driver provides a new helper method,DBCollection.aggregate() which can be used to create aggregation tasks.

Let’s use a simple example to demonstrate how the aggregation helper works. Suppose I am using MongoDB to store my employee’s travel expenses. I’ve created a collection named expenses, which store individual expenses by employee and by department. Here’s a sample document:

{ "\_id" : ObjectId("503d5024ff9038cdbfcc9da4"),

"empId" : 155,

"department" : "TO",

"amount" : 9800,

"type" : "rail ticket"

}

I am auditing three departments: TO, HR and Retail. I want to calculate each department’s average spend on airfare. I’d like to use the Aggregation Framework for the audit, so I think of the operation in terms of a pipeline:

1. Operation: Match documents where type = "rail ticket"; then pipe into
2. Operation: Pass only the department and the amount fields through the pipeline; then pipe into
3. Operation: Average the expense amount, grouped by department.

I will use the aggregation operators $match, $project and $group to perform each operation. Individual aggregation operations can be expressed as JSON objects, so I can think of my pipeline in JSON as:

First operation:

$match: {type: "rail ticket"}

Piped into:

$project: {department: 1, amount: 1 }

Piped into:

$group: { \_id: "$department”, average: { $avg: "$amount" } }

I use the Java Driver’s aggregation helper to build out this pipeline in my application. Let’s take a look at the aggregate()method signature.

**public** AggregationOutput aggregate( DBObject firstOp, DBObject ... additionalOps)

The aggregate() method uses Java varargs and accepts arbitrary number of DBObjects as parameters. These DBObjectsrepresent aggregation operations, which will be chained together by the helper method to form the aggregation pipeline. Callers of the aggregate() method must pass at least one aggregation operation. Here’s the Java code we’ll use to perform the aggregation task:

*// create our pipeline operations, first with the $match*

DBObject match = **new** BasicDBObject("$match", **new** BasicDBObject("type", "rail ticket") );

*// build the $projection operation*

DBObject fields = **new** BasicDBObject("department", 1);

fields.put("amount", 1);

fields.put("\_id", 0);

DBObject project = **new** BasicDBObject("$project", fields );

*// Now the $group operation*

DBObject groupFields = **new** BasicDBObject( "\_id", "$department");

groupFields.put("average", **new** BasicDBObject( "$avg", "$amount"));

DBObject group = **new** BasicDBObject("$group", groupFields);

*// run aggregation*

AggregationOutput output = collection.aggregate( match, project, group );

Aggregations are executed as database commands in MongoDB. These commands embed the results of the aggregation task in an object that also contains additional information about how the command was executed. The return value of aggregate()is an instance of the AggregationOutput class, which provides assessors to this information. How to get the desired result from the AggregationOutput class is described later when we build a JSP page.

## MapReduce in MongoDB using Java

MapReduce is a software framework introduced by Google in 2004 to support distributed computing on large data sets on clusters of computers. MapReduce is nothing but a design pattern for data processing. A large task is broken down into smaller subtasks. Each subtask is performed independently. The results of all these subtasks are then combined to produce the final output. MapReduce has two crucial phases:

• **The map phase**: Breakdown the task into smaller subtasks and execute them to produce intermediate results

• **The reduce phase**: Combine the intermediate results and produce the final output

If you have done functional programming in the past, the idea should not be new to you. In

the paradigm of functional programming, map() takes an array as an input and performs

an operation on each element on the array. reduce()takes the result array of map() as

its input and combines all the elements in that array into a single element by performing

some operation.

To elaborate the idea, consider the array of integers [1, 2, 3, 4, 5]. We have

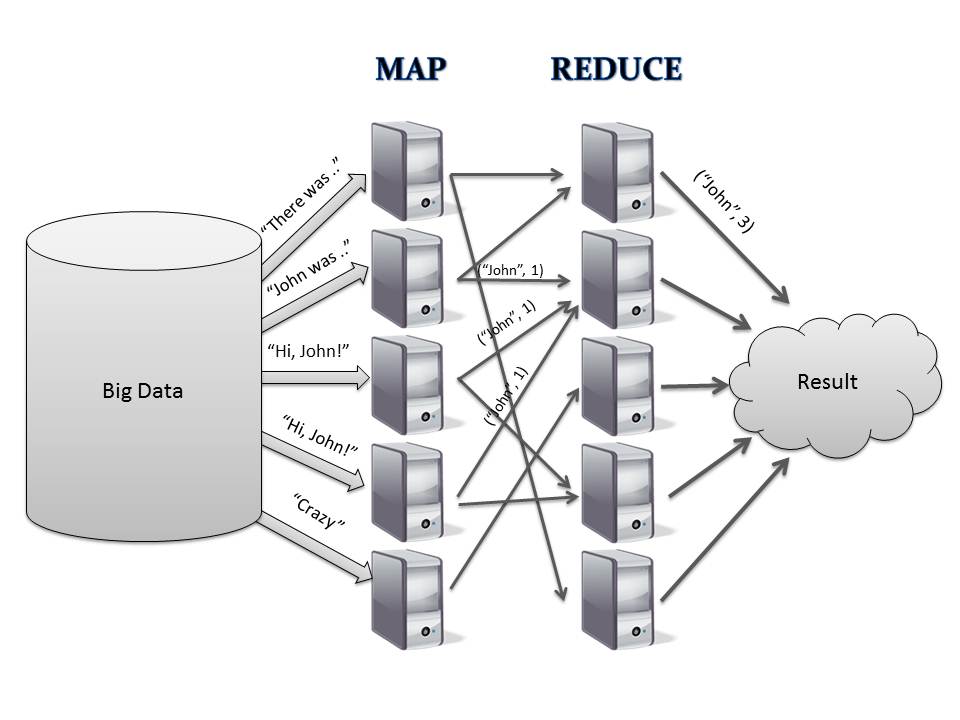
to find the sum of the cubes of all these numbers. map() takes this array and applies the

function f(x) = x3 on each integer in the array and produces the output [1, 8, 27, 64, 125].

Now, reduce() takes this output array, sums all the numbers in it, and outputs the number

225 (1+8+27+64+125 = 225).

Another example could be to find the word “John” in a blog, so the corresponding mapreduce operation can be explained as follows:



## Performing MapReduce using Java:

Let’s process this library collection in a way that, we need to find the number of books having pages less 250 pages and greater than that.

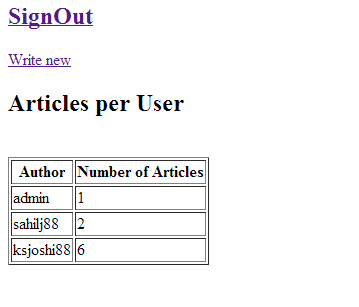
|  |  |
| --- | --- |
| import com.mongodb.BasicDBObject;  import com.mongodb.DB;  import com.mongodb.DBCollection;  import com.mongodb.DBObject;  import com.mongodb.MapReduceCommand;  import com.mongodb.MapReduceOutput;  import com.mongodb.Mongo;  public class MongoClient {    public static void main(String[] args) {    Mongo mongo;    try {  mongo = new Mongo("localhost", 27017);  DB db = mongo.getDB("library");    DBCollection books = db.getCollection("books");    BasicDBObject book = new BasicDBObject();  book.put("name", "Understanding JAVA");  book.put("pages", 100);  books.insert(book);    book = new BasicDBObject();  book.put("name", "Understanding JSON");  book.put("pages", 200);  books.insert(book);    book = new BasicDBObject();  book.put("name", "Understanding XML");  book.put("pages", 300);  books.insert(book);    book = new BasicDBObject();  book.put("name", "Understanding Web Services");  book.put("pages", 400);  books.insert(book);    book = new BasicDBObject();  book.put("name", "Understanding Axis2");  book.put("pages", 150);  books.insert(book);    String map = "function() { "+  "var category; " +  "if ( this.pages >= 250 ) "+  "category = 'Big Books'; " +  "else " +  "category = 'Small Books'; "+  "emit(category, {name: this.name});}";    String reduce = "function(key, values) { " +  "var sum = 0; " +  "values.forEach(function(doc) { " +  "sum += 1; "+  "}); " +  "return {books: sum};} ";    MapReduceCommand cmd = new MapReduceCommand(books, map, reduce,  null, MapReduceCommand.OutputType.INLINE, null);    MapReduceOutput out = books.mapReduce(cmd);    for (DBObject o : out.results()) {  System.out.println(o.toString());  }  } catch (Exception e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  } |  |

As you can see we will write the map and reduce functions in JavaScript string and pass those strings as parameters to MapReduceCommand Object, which in turn is passed as an argument to mapReduce function of mongodb collection.

## Performing Aggregation in Blogging Application:

Now that we have learned both the techniques used for performing aggregation queries in mongodb its time for us to use this techniques in our very own blogging application, we will perform these by using both aggregation framework and mapreduce framework.

So we create a new JSP page adminDashboard.jsp



Download the source code of this book for free at bbarters.com.

Now let’s have a look at the code that we have written to get these articles per user:



As first we get the collection we wanted to work with, using the dconnect singleton class. While we perform the aggregation query there are three operations which are must:

The where clause (Query), the projection (deciding which fields to show) and the grouping of data ( the field based on which we want to group documents)

Here, our where clause is blank (no query condition):

**DBObject match=new BasicDBObject("$match",new BasicDBObject());**

Now we move towards the project operation, here we want to project Author and Title fields and hide the \_id field:

**DBObject fields=new BasicDBObject("Author",1);**

**fields.put("\_id",0);**

**fields.put("Title",1);**

//We are deciding which fields to project

**DBObject project=new BasicDBObject("$project",fields);**

Now we move on to the grouping part of the query:

//Now the grouping operation

**DBObject groupFields=new BasicDBObject("\_id","$Author");**

**groupFields.put("Number of Articles",new BasicDBObject("$sum",1));**

**DBObject group=new BasicDBObject("$group",groupFields);**

We here are trying to group the articles by field “Author” and then perform the count operation on the number of articles found for particular author, so we use $sum operator with every finding of an article we add 1 to the count.

And then finally we fire the aggregation query using aggregate method of DBCollection object, the result of this method is AggregationOutput, which is iterable set of documents received as a part of aggregation output. Now the last task is to iterate through the aggregation results and present them in tabular format:

**for(DBObject a:output.results())**

**{ %>**

**<tr>**

**<td><%out.println(a.get("\_id"));%></td>**

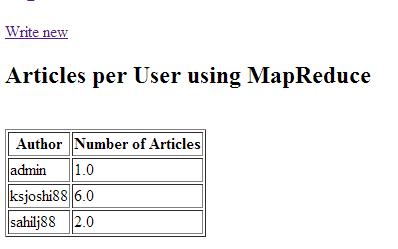
**<td><%out.println(a.get("Number of Articles"));%></td>**

**</tr>**

**<% } %>**

## Implementing MapReduce in java

We will create a new page adminMapReduce.jsp, let’s see how it looks like:



Lets see how this code works out:



The important thing to notice here is the two JavaScript functions we have written as strings:

**String map="function(){emit(this.Author,{count:1});}";** And

**String reduce="function(key,values){ total=0; for(var i in values){total+=values[i].count;} return {count:total}}";**

First we separate out the documents based on Author and increase the count by 1 every time we find one, and then in the reduce function for every different value of author we calculate the sum of count values.

The class MapReduceComman, groups the arguments for map/reduce operation and can build the underlying command object; its constructor requires 6 arguments:

* 1. The input collection
  2. Map JavaScript (String)
  3. Reduce JavaScript( String)
  4. Output collection ( if you want to put the mapreduce output into a collection)
  5. Outputype, keep it inline if you want to get the output as iterative array
  6. Query (if you want to perform any query on mapReduce output

**MapReduceCommand cmd=new MapReduceCommand(collection, map,reduce,null,MapReduceCommand.OutputType.INLINE,null);**

And finally we fire the mapReduce operation:

**MapReduceOutput mapout=collection.mapReduce(cmd);**

The mapout instance is pretty much same as aggregationOuput we saw in the last section, all you need to do is to iterate through it:  
**<%**

**for(DBObject a:mapout.results())**

**{ %> <tr>**

**<td><%out.println(a.get("\_id"));%></td>**

**<td><%DBObject val=(DBObject)a.get("value");**

**out.print(" " + val.get("count"));%></td>**

**</tr>**

**<% }%>**

# Web Analytics using Capped Collections

MongoDB is good for storing and manipulating large datasets in general. Web analytics is one example of such a large data problem. Some features which make mongodb a better choice for storing web analytics data:

• MongoDB is well suited to handle large volumes of data. The scalability features of MongoDB (replication, sharding, replica sets, and so on) help to optimize the performnace as the size of data and number of operations on it continue to grow. A high traffic website can use it to store all the user activity on the site, so they can be processed and analysed in the background.

• MongoDB supports asynchronous inserts. This means that your application code, whether it is PHP, C, or the JavaScript interface of the mongo shell, asks MongoDB to insert a document and moves on to the next instruction without waiting for the server to respond. This makes it an excellent tool for logging.

• Running MapReduce on your analytics data and generate reports that may give clues on how to better optimize the website or how to improve its usability. The Flexible schema feature of MongoDB makes it an excellent choice for web analytics storage. It is difficult to define a data structure for analytics data beforehand, because often we do not know what are the most important pieces of information we would have to store.

Also, if we decide to store any additional information (or decide not to store any existing information going forward) the flexibility of the schema makes it very easy to introduce the required changes to the data structure.

## Capped collections

A capped collection is just like any other collection in MongoDB, except that if we specify the size of the collection in bytes, it will maintain this size by itself. That means when this collection grows larger than the specified size, it replaces the oldest documents (the documents that were inserted first) automatically with new ones.

A capped collection is created explicitly by calling createCollection(), unlike regular collections which are created implicitly. A second parameter has to be passed to this method specifying that this is a capped collection and the size of the collection in bytes.

**//create a capped collection of 1000000 bytes named mylog**

**>db.createCollection('mylog', {capped:true, size: 10000})**

This command will create the collection and pre-allocate the specified size on the disk. We can also specify the maximum number of objects to be stored in a capped collection

**//cap the number of documents to 1000**

**>db.createCollection('mylog',{capped:true, size:10000, max: 1000})**

Capped collections implement natural ordering. Natural ordering is the database's native approach of ordering documents in a collection. When we query a collection, without specifying to sort on a certain field, we will get the documents in the order they were inserted. In a regular collection, this is not guaranteed to happen because as we update the documents, their sizes change and they are moved around to fit into the collection. A capped collection on the other hand guarantees that the documents are returned in the order of their insertion:

**//this will return the oldest documents first**

**db.mylog.find()**

You can also reverse the order by sending in the {$natural : -1} parameter to sort():

**//this will return the newest documents first**

**db.mylog.find().sort({$natural : -1})**

This natural ordering behaviour, coupled with the fixed size property makes capped collection an ideal logging backend (the order of items is preserved and the log is not allowed to grow beyond a specific size).

### Updating and deleting documents in a capped collection

We can update documents in a capped collection the same way we update documents for a regular collection. But there is a catch; the document being updated is not allowed to grow in size (otherwise capped collection could not guarantee natural ordering). Also, we cannot delete documents from a capped collection. We can however use drop() to delete the collection entirely.

## How to create a capped collection in mongodb using Java

The following lines of codes illustrate how to create capped collections in mongoDB using java:

int cap = **1000000**; // A capped collection with size 1000000 bytes

int max = 1000; // A capped collection with max documents 1000

DBObject options = BasicDBObjectBuilder.start().add("capped", true).add("size", cap).add("max",max ).get( );

db\_col = db.createCollection(collections\_new, options); //create a capped collection

## Web Analytics using Capped Collections:

Capped collections make a very good case for storing web analytics data, as you don’t want to log data for all users of your system for the system lifetime. By using capped collection you make sure, that user logs are maintained only up to certain period of time.

It’s time for us to add this web analytics using capped collection facility to our blogging application, for that we modify the logic of our **readBlog.jsp** page. After the use reads a particular blog, we log the user data in a collection called as userlog. This is the code we would use for that purpose:



If you have difficulties in reading the code from above image, please download the source code from bbarters.com.

In first few lines of the code, we create a capped collection with size 1000000 bytes and max documents 1000. After that, using the session attribute we get the important browsing data such as userid, the browser used, the time of visit etc. There is also a provision for history where you can track which pages he visited, for that you will have to write some code on every JSP page we have created so far and make sure you append the page name in history field of the userlog collection. Finally we put all the data in the userlog collection, which we will read from to show the web analysis data on the **webAnalytics.jsp** which looks like this:



Let’s look at the code that we have used to extract the web analytics information from the capped collection:



It’s a very simple code, again where we are trying to show the browser usage distribution for this blogging application using the userlog data from our capped collection. I won’t try and explain this code , but you can very well do it in some hurry. Please download the source code for all these examples for free on bbarters.com.

# Storing Large Files Using GridFS

[GridFS](http://docs.mongodb.org/manual/reference/glossary/#term-gridfs) is a specification for storing and retrieving files that exceed the [BSON](http://docs.mongodb.org/manual/reference/glossary/#term-bson)-document [size limit](http://docs.mongodb.org/manual/reference/limits/#limit-bson-document-size) of 16MB. Instead of storing a file in a single document, GridFS divides a file into parts, or chunks, [[1]](http://docs.mongodb.org/manual/core/gridfs/#chunk-disambiguation) and stores each of those chunks as a separate document. By default GridFS limits chunk size to 256k. GridFS uses two collections to store files. One collection stores the file chunks, and the other stores file metadata.

When you query a GridFS store for a file, the driver or client will reassemble the chunks as needed. You can perform range queries on files stored through GridFS. You also can access information from arbitrary sections of files, which allows you to “skip” into the middle of a video or audio file. GridFS is useful not only for storing files that exceed 16MB but also for storing any files for which you want access without having to load the entire file into memory.

## When should I use GridFS?

For documents in a MongoDB collection, you should always use [GridFS](http://docs.mongodb.org/manual/reference/glossary/#term-gridfs) for storing files larger than 16 MB. In some situations, storing large files may be more efficient in a MongoDB database than on a system-level filesystem.

* If your filesystem limits the number of files in a directory, you can use GridFS to store as many files as needed.
* When you want to keep your files and metadata automatically synced and deployed across a number of systems and facilities. When using [geographically distributed replica sets](http://docs.mongodb.org/manual/core/replica-set-architecture-geographically-distributed/#replica-set-geographical-distribution) MongoDB can distribute files and their metadata automatically to a number of [mongod](http://docs.mongodb.org/manual/reference/program/mongod/#bin.mongod) instances and facilities.
* When you want to access information from portions of large files without having to load whole files into memory, you can use GridFS to recall sections of files without reading the entire file into memory.

Do not use GridFS if you need to update the content of the entire file atomically. As an alternative you can store multiple versions of each file and specify the current version of the file in the metadata. You can update the metadata field that indicates “latest” status in an atomic update after uploading the new version of the file, and later remove previous versions if needed.

Furthermore, if your files are all smaller the 16 MB [BSON Document Size](http://docs.mongodb.org/manual/reference/limits/#BSON Document Size) limit, consider storing the file manually within a single document.

## GridFS Collections

[GridFS](http://docs.mongodb.org/manual/reference/glossary/#term-gridfs) stores files in two collections:

* **chunks** stores the binary chunks.
* **files** stores the file’s metadata.

GridFS places the collections in a common bucket by prefixing each with the bucket name. By default, GridFS uses two collections with names prefixed by fs bucket:

* fs.files
* fs.chunks

You can choose a different bucket name than fs, and create multiple buckets in a single database. Each document in the chunks collection represents a distinct chunk of a file as represented in the GridFS store. Each chunk is identified by its unique [ObjectID](http://docs.mongodb.org/manual/reference/glossary/#term-objectid) stored in its \_id field. For descriptions of all fields in the chunks and files collections, see [GridFS Reference](http://docs.mongodb.org/manual/reference/gridfs/).

## GridFS Index

[GridFS](http://docs.mongodb.org/manual/reference/glossary/#term-gridfs) uses a [unique](http://docs.mongodb.org/manual/reference/glossary/#term-unique-index), [compound](http://docs.mongodb.org/manual/reference/glossary/#term-compound-index) index on the chunks collection for the files\_id and n fields. The files\_id field contains the \_id of the chunk’s “parent” document. The n field contains the sequence number of the chunk. GridFS numbers all chunks, starting with 0.

The GridFS index allows efficient retrieval of chunks using the files\_id and n values, as shown in the following example:

cursor = db.fs.chunks.find({files\_id: myFileID}).sort({n:1});

## Example Interface

The following is an example of the GridFS interface in Java. The example is for demonstration purposes only. By default, the interface must support the default GridFS bucket, named fs, as in the following:

*// returns default GridFS bucket (i.e. "fs" collection)*

GridFS myFS = **new** GridFS(myDatabase);

*// saves the file to "fs" GridFS bucket*

myFS.createFile(**new** File("/tmp/largething.mpg"));

Optionally, interfaces may support other additional GridFS buckets as in the following example:

*// returns GridFS bucket named "contracts"*

GridFS myContracts = **new** GridFS(myDatabase, "contracts");

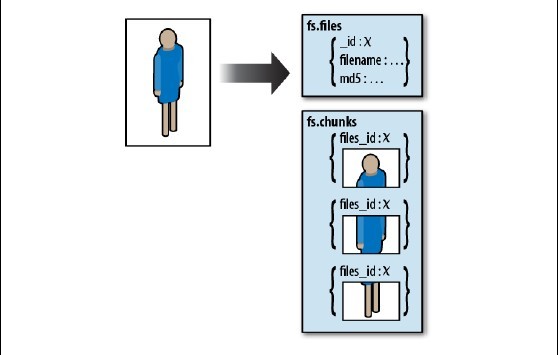
*// retrieve GridFS object "smithco"*

GridFSDBFile file = myContracts.findOne("smithco");

*// saves the GridFS file to the file system*

file.writeTo(**new** File("/tmp/smithco.pdf"));

Let's learn about the GridFS concept briefly. GridFS stores a file in two separate collections: **files** and **chunks**. The basic idea is for every file to be stored in GridFS, files will have exactly one document that will contain the filename, size, time of upload, and any other metadata set by the user. The contents of the file will be stored in one or more documents in chunks. GridFS concept can be pictorially expressed as:



## Java Example:

Mongo mongo = **new** Mongo("localhost", 27017);

DB db = mongo.getDB("test");

DBCollection collection = db.getCollection("mygrid\_meta");

File file = **new** File("/Users/picture.avi ");

*// Store the file to MongoDB using GRIDFS*

GridFS gridfs = **new** GridFS(db, "mygrid");

GridFSInputFile gfsFile = gridfs.createFile(file);

gfsFile.setFilename("picture");

gfsFile.save();

*// Let's create a new JSON document with some "metadata" information on the download*

BasicDBObject info = **new** BasicDBObject();

info.put("name", "MongoDB");

info.put("fileName", "Picture");

info.put("rawName", "mypicture");

info.put("rawPath", "/Users/");

*// Let's store our document to MongoDB*

collection.insert(info, WriteConcern.SAFE);

Hopefully the example is pretty straightforward and good to understand on its own. Basically there are two things happening there. The blob is stored to *MongoDB* using GRIDFS. Afterwards a document is created to *MongoDB* (to a different collection) using the Java-version of the commands we have seen in the course of this blog series. For the sake of keeping the example short absolutely no error handling is done, but any exception will be simple thrown to the command line. Let’s take a look what we have in *MongoDB* after running the Java program. Therefore I display the collections that are available inside my test-db in the Mongo-shell.

> show collections

users

mygrid.chunks

mygrid.files

mygrid\_meta

system.indexes

**mygrid.chunks** – *MongoDB* has created this one to store smaller chunks of the big files. This way it can stay inside the limits of a maximum size of 16 MB it has for one document.

**mygrid.files** – This collection contains the document of the actual file, so that we do not have to fiddle around with individual chunks on our own.

**mygrid \_meta** – This one has been created automatically when we saved the metadata document inside the Java program.

Checking the files-document a lot of – what I would say is – *MongoDB* internal attributes can be seen, but also the filename that we have given for this file in the Java program. I am not going to look into individual chunks, but let’s have a count to see how many have been created for this document, which by the way had a size of roughly 174,4 MB.

> db. mygrid.files.find().pretty()

{

"\_id" : ObjectId("509ec4dea0eebce38606d798"),

"chunkSize" : NumberLong(262144),

"length" : NumberLong(178421760),

"md5" : "1522b62e75c973b7b1d780695807047c",

"filename" : "picture",

"contentType" : null,

"uploadDate" : ISODate("2012-11-10T21:19:26.256Z"),

"aliases" : null

}

> db.mygrid.chunks.count()

681

Let’s have a look at the metadata that we have created altogether with storing the file. No surprises here I would say.

> db.mygrid\_meta.find().pretty()

{

"\_id" : ObjectId("509ec4e2a0eebce38606da42"),

"name" : "MongoDB",

"fileName" : "MongoDB-OSX-2-1.2.1",

"rawName" : "mongodb-osx-x86\_64-2-1.2.1.tgz",

"rawPath" : "/Users/thomasjaspers/Downloads/"

}

## Benefits of using GridFS:

• With GridFS we could store millions of files under one (logical) directory. Traditional filesystems will not allow us to do so (even if they do, it will be at the cost of a serious performance decrease).

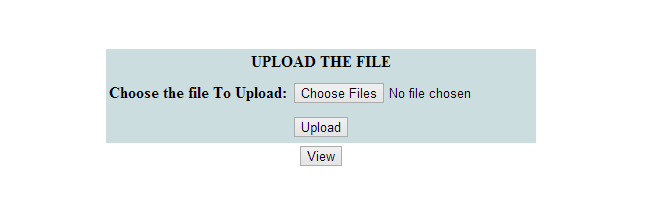
• In a distributed environment, where multiple machines have to access the files, GridFS is a much better choice than having a networked/distributed filesystem. The built-in replication schemes of MongoDB can be used to replicate and sync the files on multiple machines.

• If you have taken measures for backing up your MongoDB data, it will work for backing up the files stored in GridFS as well. You do not have to design a separate system for backing up your filesystem.

• You can store whatever metadata you consider important along with the file itself. For example, if your site allows users to upload photos/videos, you can also store any comments, likes, or ratings along with the file. Since files are stored in chunks, you can access random parts of a large file. Traditional tools that implement this feature on the filesystems are not very good.

## Using GridFS to store Images

Now that we know how to use GridFS lets incorporate this cool thing in our blogging application. We will provide the user with a facility to upload the images and of course to view them. Let’s add page **uploadImage.jsp**, which looks like this:



Now, let’s look at the code which we will be using to put the images into mongodb collection in binary format, the file will be **uploadfile.jsp:**



The first few lines are getting images from the form element in the file **uploadImage.jsp** and putting those images in D:/jsp\_work/Images. After some trivial code of getting the clean path for each image, we create a GridFS object for each photo.

**GridFS gfsPhoto = new GridFS(db, "photo");**

Creates a GridFS instance for the specified prefix for collection names ( files and chunks) in the given database. After that we create a file in collection for every entry in our directory:

GridFSInputFile gfsFile = gfsPhoto.createFile(imageFile);

GridFSInputFile class creates a file entry. After calling this method, you have to call save() on the GridFSInputFile file. We set the fileName before we call the save method, as soon as you call the save method, the method itself calls [save (long)](http://api.mongodb.org/java/current/com/mongodb/gridfs/GridFSInputFile.html#save(long)) with the existing chunk size (128Mb by default). Remember you don’t have to worry about how it internally stores the data in files and chunks at all. Now that we have stored the files in mongodb in binary format, it’s time to retrieve them, we will try and get the binary data using GridFS and put it to show using HTML img tag.

For that we have viewImages.jsp page:



Ok, that’s me all over the place; let’s look at the code very quickly:



First we get the file list using:

**Mongo mongo = new Mongo("localhost", 27017);**

**DB db = mongo.getDB("javablogs");**

**GridFS gfsPhoto = new GridFS(db,"photo");**

**DBCursor cursor = gfsPhoto.getFileList();**

Now that we have the list, we will iterate through the documents and try to show every image using the HTML img tag.

**while(cursor.hasNext())**

**{**

**DBObject fdb=cursor.next();**

**String filename = String.valueOf(fdb.get("filename"));**

**GridFS gfsPhotolist = new GridFS(db,"photo");**

**GridFSDBFile file = gfsPhotolist.findOne(filename);**

**InputStream inputfile = file.getInputStream();**

**BufferedImage image = ImageIO.read(inputfile);**

**ByteArrayOutputStream baos = new ByteArrayOutputStream();**

**ImageIO.write(image, "jpeg", baos);**

**byte[] res =baos.toByteArray();**

**String encodedImage =Base64.encodeBase64String(res); %>**

**<img src="data:image/jpeg;base64,<%=encodedImage%>" width="160" height="200"/>**

**<% }**

The data of image is stored in BSON, it must be converted to base64 format before it can be used to print the image on the webpage. For that purpose we have written all the code. Again, thing to note here is, you don’t have to worry about how gridFS collects data from all the chunks and forms the image as one.

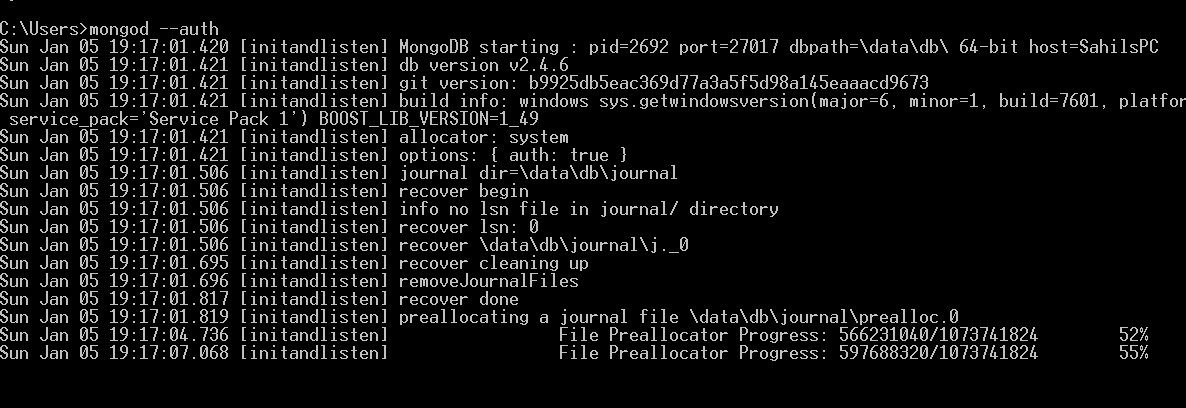
**InputStream inputfile = file.getInputStream();**

This line will get you all the binary data for that particular image from all the chunks related to that file.

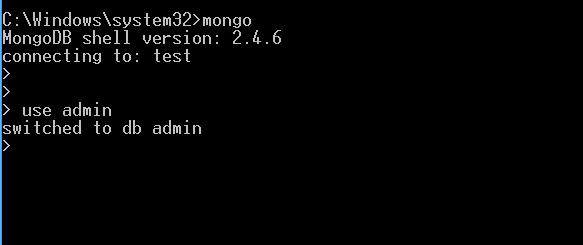
# Security and Performance

MongoDB provides a basic user authentication mechanism for authorizing users accessing the database. In this section, we are going to learn how to add user authentication, adding/removing users to/from databases, and connecting to MongoDB with PHP in authenticated mode.

Launch the monogd process in authentication mode by passing in the --auth flag:

****

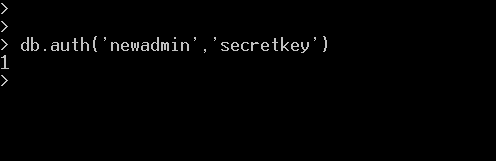
Launch the mongo shell and switch to admin database:

****

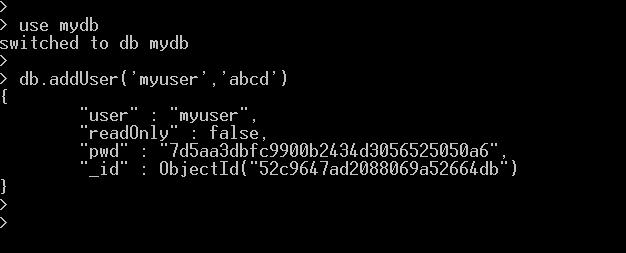
Add a user named newadmin to the admin database and set an arbitrary password:



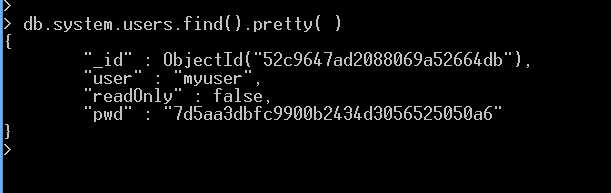
Authenticate yourself as the just created admin user using the following command:

****

Switch to a different database named mydb. Add the user myuser for this database:

****

Issue the next command to list all the users of the current database:

****

With these we have learned to turn authentication on in the MongoDB server and create user accounts with administrative privileges over the whole server and/or a certain database. To turn authentication on, the mongod server process has to be started with the --auth flag: We worked in admin database. This is a special database for doing administrative tasks.

## Creating an admin user

An admin user is one who has administrative privileges over the entire database server. To

Create an admin, we have to be in the admin database and use the addUser() method,

as follows:



This command creates an admin user named newadmin and sets the password as secretkey. Once created, we have to authenticate as the admin user using the auth() method to perform further administrative tasks:

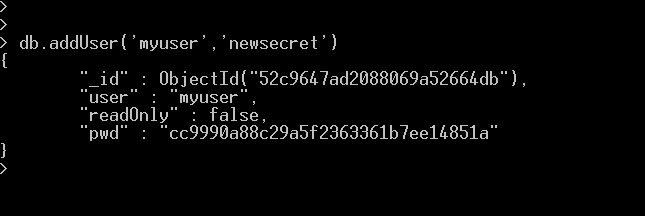
**> db.auth(‘newadmin', ‘secretkey')**

**1**

An admin user can create/remove database-specific users, and has read-write access to all databases on the server.

## Changing passwords and deleting user accounts

To change the password of an existing user, use the following command:

****

Since the myuser account already exists, the command will set its password to newsecret.

Finally, to remove a user account, use the following command:

**> db.removeUser(‘testuser')**

This will delete the myuser account from the namespace.

## User Authentication through Java Driver

The following code can be used to authenticate a database user in java:

Mongo mymongo = new Mongo(localhost,Integer.parseInt(port));

DB db = mymongo.getDB("admin");

CommandResult res = db.authenticateCommand(username, password.toCharArray());

// checking the length of command result

if(res.toString().length()!=0)

{

//do something

}

else

{

//print message ("Invalid User!!!");

}

## Increasing query performance using indexes

In MongoDB, an index is a special data structure (A B-Tree in computer science terms) that holds information about the values of specific fields of the documents in a collection. When we query the collection on these fields, MongoDB looks into this data structure to quickly sort through and order the documents. To understand how indexing benefits query performance, consider a collection of movies:

**> db.movies.find()**

**{ "\_id" : ObjectId("4db439153ec7b6fd1c9093ec"), "name" : “Accepted","genre" : "drama", "year" : 2006 }**

**{ "\_id" : ObjectId("4db439df3ec7b6fd1c9093ed"), "name" : "The Girl Next Door", "genre" : "romance", "year" : 2004 }**

**{ "\_id" : ObjectId("4db439f33ec7b6fd1c9093ee"), "name" : "Paranoia", “genre" : "drama", "year" : 2013 }**

**{ "\_id" : ObjectId("4db439f33ec7b6fd1c9093ef"), "name" : "Prisoners", "genre" : "thriller", "year" : 2013 }**

**{ "\_id" : ObjectId("4db439f43ec7b6fd1c9093f0"), "name" : "Hangover 3", genre" : "comedy", "year" : 2013 }**

Now, let's assume that we need to find all the movies with drama genre. We can do that simply by querying on the genre field.

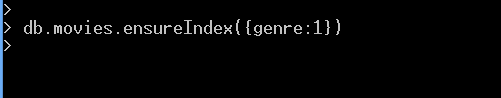
**> db.movies.find({genre: "drama"})**

**{ "\_id" : ObjectId("4db439153ec7b6fd1c9093ec"), "name" : “Accepted","genre" : "drama", "year" : 2006 }**

**{ "\_id" : ObjectId("4db439f33ec7b6fd1c9093ee"), "name" : "Paranoia", “genre" : "drama", "year" : 2013 }**

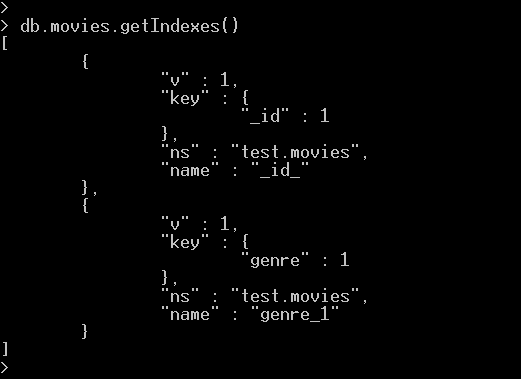
When we queried, MongoDB looked through every document in the collection and checked the value of the genre field. If the collection had a larger number of documents, it would take longer for the query to execute. This is where indexing comes into picture. If the collection has an index on the genre field, MongoDB will check the index first instead of the collection, and randomly access the documents that matched the criteria.

So now, we create an index on genre field as according to our requirement:



So, where we are creating index on genre field which will be stored in database in ascending order.

Now let’s find out how many indexes are there in our collection:

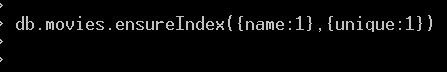


If you look at the output of getIndexes(), you will notice there is an index on the \_id field, even though we did not create this index explicitly.

The \_id index gets created automatically, every time we create a collection, except for capped collections. The values of \_id fields are unique for each document and invariant. Lookups using the \_id field always tend to be fast.

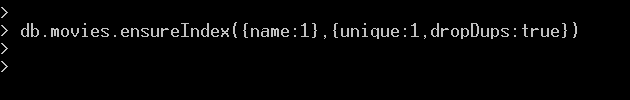
## Creating unique indexes

Unique indexes are like the UNIQUE KEY in MySQL. It ensures that no two documents in a collection will have the same value for the indexed field, i.e no duplicates. The following command creates a unique index on the name field of the movies collection:

****

Now if we try to insert two documents with the same value for the uniquely indexed field, we will get an error. If you insert a document that is missing the value for the uniquely indexed field, MongoDB will insert a null value for that value.

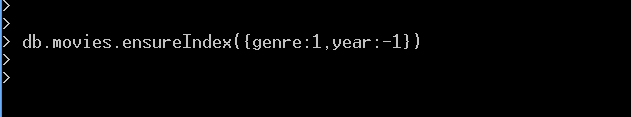
If the field in question has non-unique values across the collection, we will not be able to create a unique index for it. To get around this, we can add the dropDups option. For example, the next command will create a unique index on the name field.

****

If there are multiple documents with the same value for name, only the first such document will be indexed and the rest of them will be dropped. Obviously, we need to be mindful about the possible data loss when applying such an operation.

## Creating compound keys indexes

In MongoDB, you can create an index on multiple fields of a document. This is known as a compound keys index. The next command creates a compound index on genre and year fields of movies:



Direction (1 ascending, -1 descending) is very important in compound indexes when you need to perform sorting and range queries. Sometimes the business logic may require showing recently released movies appear first in the UI, and the movies should be alphabetically ordered by their genres. So, it makes sense to create a compound index on the year field in descending order and on the genre field in ascending order. For single-key indexes or random access retrievals, ordering does not matter much (unless the queries involve reversing the result set order).

Furthermore, when you have a compound index on several fields, you can use it to query on a subset of fields. For example, suppose there is a compound index on fields a, b, and c of the documents in a collection. This index can be used to query on:

• a

• a, b

• a, b, and c

So, if you have a compound index on a, b, and c, you don't need a single key index on a (or a compound keys index on b, c). However, if we do a query on b, c or b only, the compound index may not be used unless we explicitly hint the query planner to do so using the hint () command.

## Indexing embedded document fields

Indexes can be applied on any type of fields, including an embedded document. For example, let's say we have an additional meta field in our movies collection, storing the duration of the movie and the name of the studio:

**> db.movies.insert({name: "Thor", genre: "action", year: 2011, meta:{duration\_minutes: 115, studio: "Paramount"}})**

We can create an index on the meta field as follows:

**> db.movies.ensureIndex({meta: 1})**

We could do equality or range queries on the fields of the embedded document:

**> db.movies.find({meta: {duration\_minutes: 115, studio: "Paramount"}})**

**{ "\_id" : ObjectId("4e58bf4c3eadcfa57f69447b"), "name" : "Thor", "genre": "action", "year" : 2011, "meta" : { "duration\_minutes" : 115,"studio" : "Paramount" } }**

> db.movies.find({meta: {$gte :{duration\_minutes: 115}}})

**{ "\_id" : ObjectId("4e58bf4c3eadcfa57f69447b"), "name" : "Thor", "genre": "action", "year" : 2011, "meta" : { "duration\_minutes" : 115,"studio" : "Paramount" } }**

Alternatively, we could use Dot notation to reach into the embedded documents and create

compound key indexes on their fields.

**> db.movies.ensureIndex({"meta.duration\_minutes": 1, "meta.studio": 1})**

**> db.movies.find({"meta.duration\_minutes": 115})**

There is an important difference between creating an index on a nested document and creating an index on a specific field of the nested document. In the first one, we have to specify the query parameters in the same order as they appear in the embedded document.

//this query will return an object

**> db.movies.find({meta: {duration\_minutes: 115, studio: "Paramount"}})**

//but this query will not

**> db.movies.find({meta: {studio: "Paramount", duration\_minutes: 115}})**

This is not the case when you use the dot notation; you can specify the query parameter in any order:

**> db.movies.find({"meta.studio": "Paramount", "meta.duration\_minutes":115})**

## Indexing array fields

Indexes can also be built on array fields. For example, suppose the documents in the movies collection have an array field named tags:

**> db.movies.insert({name: "Iron Man 2", genre: "action", year: 2010,tags: ['superhero', ‘marvel', ‘comics', ‘scifi']})**

If we build an index on this array field, MongoDB will index each element of the array:

> db.movies.ensureIndex({tags: 1})

> db.movies.find({tags: ‘superhero'})

Guidelines for creating indexes:

Choose keys wisely

Keep an eye on size

Avoid using low selectivity single key indexes

Aware of indexing costs

## Creating Indexes using Java

DB db = m.getDB(selectedDatabase);

DBCollection collection = db.getCollection(coll);

//Creating indexes

collection.ensureIndex(new BasicDBObject(head\_selected, 1));

## Optimizing queries

In this section, we are going to look at some tools provided by MongoDB for analysing individual queries, and learn how to use their output for optimization.

**Explaining queries using explain()**

The explain() method is used to explain a query, giving us useful information about how the query was performed, which we could use to fine-tune the query itself. It is invoked on a cursor, and it returns a document holding pieces of data about the query.

> **db.movies.find({name: ‘Inception'}).explain()**

{

"cursor" : "BtreeCursor genre\_1",

"nscanned" : 3,

"nscannedObjects" : 3,

"n" : 3,

"millis" : 0,

"nYields" : 0,

"nChunkSkips" : 0,

"isMultiKey" : false,

"indexOnly" : false,

"indexBounds" : {

"genre" : [

[

"action",

"action"

]

]

}

}

Let's take a look at some of the important information returned by explain():

• **cursor**: The value for this field could either be a BasicCursor or a BtreeCursor. If it is the second, it means the query has used an index. Since the genre field was indexed, the value is obviously BtreeCursor in this case.

• **nscanned**: It returns the number of items scanned through the collection by the query.

• **n:** It returns the number of documents returned by the query.

• **millis**: It returns the number of milliseconds it took for the database to execute the query.

**Optimization rules**

We can employ the following rules for optimizing queries using the output of explain():

• The number of items scanned (nscanned) should be close to the number of

documents returned (n). If the query is scanning a large number of documents and

returning a small number of them, we should fine-tune the indexing on the fields.

• The number of milliseconds to perform the query (millis) should be very small.

## Using hint()

The hint() method can be used to explicitly direct MongoDB to use a certain index. Let's say we are querying on multiple fields and only some of those fields are indexed. We can supply the indexed field as a JSON argument to hint() and force MongoDB to use the index.

//hint the query planner to use the genre index

**> db.movies.find({name: ‘Inception', genre: ‘sci-fi'}).hint({genre:1})**

In most situations, using hint() will be unnecessary, because the MongoDB query optimizer is quite smart about choosing which index to use. When a query is run for the first time, the optimizer attempts multiple query plans concurrently. It uses the plan that finishes first and suspends the others. This query plan is used in all future queries using the same keys.

# UI tools for mongoDB administration

In this chapter we will talk about two very good Java based mongoDB administration UI tools

MonjaDB

Bmongo