Casbah (MongoDB + Scala Toolkit) Documentation

Release 2.0b3p1

Brendan W. McAdams & 10gen, Inc.

December 09, 2010

CONTENTS

Welcome to the Casbah Documentation. Casbah is a Scala toolkit for MongoDB—We use the term "toolkit" rather than "driver", as Casbah integrates a layer on top of the official mongo-java-driver for better integration with Scala. This is as opposed to a native implementation of the MongoDB wire protocol, which the Java driver does exceptionally well. Rather than a complete rewrite, Casbah uses implicits, and *Pimp My Library* code to enhance the existing Java code.

Casbah's approach is intended to add fluid, Scala-friendly syntax on top of MongoDB and handle conversions of common types. If you try to save a Scala List or Seq to MongoDB, we automatically convert it to a type the Java driver can serialize. If you read a Java type, we convert it to a comparable Scala type before it hits your code. All of this is intended to let you focus on writing the best possible Scala code using Scala idioms. A great deal of effort is put into providing you the functional and implicit conversion tools you've come to expect from Scala, with the power and flexibility of MongoDB.

The ScalaDocs for Casbah along with SXR cross referenced source are available at the MongoDB API site.

You may also download this tutorial in other formats.

- ePub
- PDF

CHAPTER

ONE

GETTING STARTED

1.1 Why Casbah?

Casbah grew (originally named *mongo-scala-wrappers*) from the frustration of dealing with a very structured, objects & imperative only approach which was forced by the Mongo Java drivers. These drivers are a fine, well built tool - but best suited to the pure Java programmer. In addition, the primary developer (Brendan McAdams) came from a background of working with MongoDB primarily with Python. He missed the fluid syntax and similarity to the JS Shell which Python provided (and let's face it, pymongo is awesome), and the flexibility of syntax. As he was learning Scala he saw the ability to provide much of this functionality.

During the course of building out application infrastructure with MongoDB + Scala, with the ideas of (Python|Mongo)esque syntax with a functional bent, Casbah slowly emerged. Casbah provides improved interfaces to GridFS, Map/Reduce and the core Mongo APIs. It also provides a fluid query syntax which emulates an internal DSL and allows you to write code which looks like what you might write in the JS Shell. There is also support for easily adding new serialization/deserialization mechanisms for common data types (including Joda Time, if you so choose; with some caveats - See the GridFS Section).

With version 2.0, Casbah has become an official MongoDB project and will continue to improve the interaction of Scala + MongoDB. Casbah aims to remain fully compatible with the existing Java driver—it does not talk to MongoDB directly, preferring to wrap the Java code. This means you shouldn't see any wildly unexpected behavior from the underlying Mongo interfaces when a data bug is fixed.

1.2 Installing & Setting up Casbah

You should have MongoDB setup and running on your machine (these docs assume you are running on *localhost* on the default port of *27017*) before proceeding. If you need help setting up MongoDB please see the MongoDB quickstart install documentation.

To start with, you need to either download the latest Casbah driver and place it in your classpath, or set it up in the dependency manager/build tool of your choice (The authors highly recommend the Scala simple-build-tool - it makes Scala development easy).

1.2.1 Setting up without a Dependency/Build Manager (Source + Binary)

The latest build as of December 09, 2010 is 2.0b3p1, cross-built for both Scala 2.8.0 (final) and Scala 2.8.1 (final).

The builds are published to the Scala-tools.org Maven repositories and should be easily available to add to an existing Scala project.

You can always get the latest source for Casbah from the github repository:

```
$ git clone git://github.com/mongodb/casbah
```

***PLEASE NOTE*: As of the 2.0 release, Casbah has been broken into** several modules which can be used to strip down which features you need. For example, you can use the Query DSL independent of the GridFS implementation if you wish; please see *casbah-modules*. The following dependency manager information uses the master artifact which downloads and uses *all* of Casbah's modules by default.

1.2.2 Using Dependency/Build Managers

First, you should add the package repository to your Dependency/Build Manager. Our releases & snapshots are currently hosted at:

```
http://scala-tools.org/repo-releases/ /* For Releases */
http://scala-tools.org/repo-snapshots/ /* For snapshots */
```

Set both of these repositories up in the appropriate manner - they contain Casbah as well as any specific dependencies you may require. (SBT users note that Scala-Tools is builtin to SBT as most Scala projects publish there)

1.2.3 Setting Up Maven

You can add Casbah to Maven with the following dependency block.

Scala 2.8.0 users:

```
<dependency>
    <groupId>com.novus<groupId>
    <artifactId>casbah_2.8.0<artifactId>
    <version>2.0b3p1<version>
<dependency>
```

Scala 2.8.1 users:

```
<dependency>
    <groupId>com.novus<groupId>
    <artifactId>casbah_2.8.1<artifactId>
    <version>2.0b3p1<version>
<dependency>
```

1.2.4 Setting Up Ivy (w/ Ant)

You can add Casbah to Ivy with the following dependency block.

Scala 2.8.0 users:

```
<dependency org="com.novus" name="casbah_2.8.0" rev="2.0b3p1"/>
```

Scala 2.8.1 users:

```
<dependency org="com.novus" name="casbah_2.8.1" rev="2.0b3p1"/>
```

1.2.5 Setting up SBT

Finally, you can add Casbah to SBT by adding the following to your project file:

```
val casbah = "com.novus" %% "casbah" % "2.0b3p1"
```

The double percentages (%%) is not a typo—it tells SBT that the library is crossbuilt and to find the appropriate version for your project's Scala version. If you prefer to be explicit you can use this instead:

```
// Scala 2.8.0
val casbah = "com.novus" % "casbah_2.8.0" % "2.0b3p1"
// Scala 2.8.1
val casbah = "com.novus" % "casbah_2.8.1" % "2.0b3p1"
```

Don't forget to reload the project and run `sbt update` afterwards to download the dependencies (SBT doesn't check every build like Maven).

CHAPTER

TWO

TUTORIAL: USING CASBAH

2.1 Import the Driver

Now that you've added Casbah to your project, it should be available. As of this writing, it lives in the package namespace `com.mongodb.casbah`. Casbah uses a few tricks to act as self contained as possible - it provides an `Imports` object which automatically imports everything you need including Implicits, and type aliases to a few common MongoDB types. This means you should only need to use our `Imports` package for the majority of your work. Let's start out bringing it into your code. At the appropriate place (Be it inside a class/def/object or at the top of your file), add our import:

```
import com.mongodb.casbah.Imports._
```

That's it. Most of what you need to work with Casbah is now at hand. .. If you want to know what's going on inside the `Imports._` take a look at Implicits.scala which defines it.

2.2 Briefly: Automatic Type Conversions

The other important thing to note is that as soon as you construct a `MongoConnection` object, a few type conversions will be loaded automatically for you - Scala's builtin regular expressions (e.g. `"\\d{4}-\\d{2}-\\d{2}".r` will now serialize to MongoDB automatically with no work from you), as well as a few other things. You can also explicitly enable serialization and deserialization of Joda time (w/ full support for the Scala-Time wrappers) by an explicit call:

```
import com.mongodb.casbah.conversions.scala._
RegisterJodaTimeConversionHelpers()
```

Once these are loaded, Joda Time (and Scala Time wrappers) will be saved to MongoDB as proper BSON Dates, and on retrieval/deserialization all BSON Dates will be returned as Joda `DateTime` instead of a JDK Date (aka *java.util.Date*). Because this can cause problems in some instances, you can explicitly unload the Joda Time helpers:

```
import com.mongodb.casbah.conversions.scala._
DeregisterJodaTimeConversionHelpers()
```

And reload them later as needed. If you find you need to unload the other helpers as well, you can load and unload them just as easily:

```
import com.mongodb.casbah.conversions.scala._
DeregisterConversionHelpers()
RegisterConversionHelpers()
```

I haven't managed to nail down every possible Scala -> Java conversion yet but it does a pretty good job most of the time. If you get any weird errors let me know and we'll create registrations for it!

2.2.1 Wrappers

Casbah provides a series of wrapper classes (and in some cases, companion objects) which proxy the "core" Java driver classes to provide scala functionality.

In general, we've provided a "Scala-esque" wrapper to the MongoDB Java objects whereever possible. These make sure to make iterable things `Iterable`, Cursors implement `Iterator`, DBObjects act like Scala Maps, etc.

2.2.2 Connecting to MongoDB

The core Connection class as you may have noted above is `com.mongodb.casbah.MongoConnection`. There are two ways to create an instance of it. First, you can invoke `.asScala` from a MongoDB builtin Connection (`com.mongodb.Mongo`). This method is provided via implicits. The pure Scala way to do it is to invoke one of the `apply` methods on the companion object:

```
// Connect to default - localhost, 27017
val mongoConn = MongoConnection()
// mongoConn: com.mongodb.casbah.MongoConnection

// connect to "mongodb01" host, default port
val mongoConn = MongoConnection("mongodb01")
// mongoConn: com.mongodb.casbah.MongoConnection

// connect to "mongodb02" host, port 42001
val mongoConn = MongoConnection("mongodb02", 42001)
// mongoConn: com.mongodb.casbah.MongoConnection
```

If you imported `Imports._`, you already have `MongoConnection` in scope and don't require additional importing. These all return an instance of the `MongoConnection` class. This class provides all the methods as the Java `Mongo` class it proxies (which is available from the `underlying` attribute, incidentally) with the addition of having an apply method for getting a DB instead of calling `getDB()`:

```
val mongoDB = mongoConn("casbah_test")
// mongoDB: com.mongodb.casbah.MongoDB = casbah_test
```

This should allow a more fluid Syntax to working with Mongo. The DB object provides an `apply()` as well for getting Collections so you can freely chain them:

```
val mongoColl = mongoConn("casbah_test")("test_data")
// mongoColl: com.mongodb.casbah.MongoCollection = MongoCollection()
```

2.2.3 Working with Collections

Feel free to explore Casbah's `MongoDB` object on your own; for now let's focus on `MongoCollection`.

It should be noted that Casbah's *MongoCollection* object implements Scala's Iterable[A] interface (specifically `Iterable[DBObject]`), which provides a full monadic interface to your MongoDB collection. Beginning iteration on the `MongoCollection` instance is fundamentally equivalent to invoking `find` on the `MongoCollection`. We'll return to this after we discuss working with `MongoDBObjects` and inserting data...

2.2.4 MongoDBObject - A Scala-ble DBObject Implementation

As a Scala developer, I find it important to be given the opportunity to work consistently with my data and objects - and in proper Scala fashion. To that end, I've tried where possible to ensure Casbah provides Scala-ble (my phrasing fr the Scala equivalent of "Pythonic") interfaces to MongoDB without disabling or hiding the Java equivalents. A big part of this is extending and enhancing Mongo's `DBObject` and related classes to work in a Scala-ble fashion.

That is to say - `DBObject`, `BasicDBOBject`, `BasicDBObjectBuilder`, etc are still available - but there's a better way. *MongoDBObject* and its companion trait (tacked in a few places implicitly via Pimp-My-Library) provide a series of ways to work with Mongo's DBObjects which closely match the Collection interface Scala 2.8 provides. Further, `MongoDBObject` can be implicitly converted to a `DBObject` - so any existing Mongo Java code will accept it without complaint. There are two easy ways to create a new `MongoDBObject`. In an additive manner:

```
val newObj: DBObject = MongoDBObject("foo" -> "bar",
                                     "x" -> "y",
                                     "pie" -> 3.14,
                                     "spam" -> "eggs")
// newObj: com.mongodb.casbah.Imports.DBObject =
// { "foo" : "bar" , "x" : "y" , "pie" : 3.14 , "spam" : "eggs"}
```

You should note the use of the **->** there. You may recall that `"foo" -> "bar"` is the equivalent of `("foo", "bar")`; however, the **->** is a clear syntactic indicator to the reader that you're working with `Map`-like objects. The explicit type annotation is there merely to demonstrate that it will happily return itself as a `DBObject`, should you so desire. (You should also be able to call the `asDBObject` method on it). However, in most cases this shouldn't be necessary - the Casbah wrappers use View boundaries to allow you to implicitly recast as a proper `DBObject`. You could also use a Scala 2.8 style builder to create your object instead:

```
val builder = MongoDBObject.newBuilder
builder += "foo" -> "bar"
builder += "x" -> "y"
builder += ("pie" -> 3.14)
builder += ("spam" -> "eggs", "mmm" -> "bacon")
// You must explicitly cast the result to a DBObject
// to receive it as such; you can also request it 'asDBObject'
val newObj = builder.result.asDBObject
// newObj: com.mongodb.DBObject =
/* { "foo" : "bar" , "x" : "y" , "pie" : 3.14 ,
     "spam" : "eggs" , "mmm" : "bacon"} */
```

Being a builder - you must call `result` to get a `DBObject`. You cannot pass the builder instance around and treat it like a `DBObject`. I find these to be the most effective, Scala-friendly ways to create new Mongo objects. You'll also find that despite the fact that these are `com.mongodb.DBObject` instances now, they provide a Scala `Map` interface via implicits. For example, one can *put* a value to `newObj` via +=:

```
newObj += "OMG" -> "Ponies!"
// com.mongodb.casbah.MongoDBObject =
// Map((foo,bar), (x,y), (pie,3.14), (spam,eggs), (mmm,bacon), (OMG,Ponies!))
newObj += "x" -> "z"
// com.mongodb.casbah.MongoDBObject =
// Map((foo,bar), (x,z), (pie,3.14), (spam,eggs), (mmm,bacon), (OMG,Ponies!))
```

Note that last - as one would expect with Scala's Mutable `Map`, a *put* on an existing value updates it in place. The first statement adds a new value. We can also speak to the DBObject as if it's a `Map`, for example, to get a value. Note that because MongoDB's `DBObject` always stores `Object` (or, in Scala terms `AnyRef` - you can always force boxing of `AnyVal` primitives with an *5.asInstanceOf[AnyRef]*), you are going to want to cast the retrieved value. Casbah attempts to automatically infer a type from you however, just remember to do it on the REPL or it will guess `scala.Nothing`:

```
val x = newObj("OMG")
// throws java.lang.ClassCastException:
// java.lang.String cannot be cast to scala.runtime.Nothing$
val x: String = newObj("OMG")
// x: String = Ponies!
val x = newObj[String]("OMG")
// x: String = Ponies!
```

These functions are available on ANY `DBObject` - not just ones you created through the `MongoDBObject` function. You can also use the standard nullsafe 'I want an option' functionality. However, due to a conflict in DBObject you need to invoke `getAs` - `get` invokes the base `DBObject` java method. This cannot currently infer type, but requires you to pass it explicitly:

```
val foo = newObj.getAs[String]("foo")
// foo: Option[String] = Some(bar)
val omgWtf = newObj.getAs[String]("OMGWTF")
// omgWtf: Option[String] = None
val omgWtfFail = newObj.getOrElse("OMGWTF",
                                  throw new Error("OMG! WTF? BBQ!"))
// java.lang.Error: OMG! WTF? BBQ!
```

Note that the standard `orElse` get method is available. It's possible additionally to join multiple `DBObjects` together:

```
val obj2 = MongoDBObject("n" -> "212")
val z = newObj ++ obj2
// z: scala.collection.mutable.Map[String,java.lang.Object] =
//  Map((foo,bar), (mmm,bacon), (spam,eggs), (pie,3.14), (n,212), (x,y))
```

However, many of the Map methods don't explicitly do the "OH I'm a DBObject" work for you - in fact, you could put a `DBObject` on one side and a `Map` on the other. But all `Map` instances can be cast as a `DBObject` either explicitly, or with an `asDBObject` call:

```
z.asDBObject
// com.mongodb.DBObject =
/* { "foo" : "bar" , "mmm" : "bacon" ,
     "spam" : "eggs" , "pie" : 3.14 ,
     "n" : "212" , "x" : "y"}
*/

val zDBObj: DBObject = z
// zDBObj: com.mongodb.casbah.Imports.DBObject =
/* { "foo" : "bar" , "mmm" : "bacon" ,
     "spam" : "eggs" , "pie" : 3.14 ,
      "n" : "212" , "x" : "y"}
 */
```

There's actually some magic hiding behind many of the above implicits - it's possible to convert any `Product` (e.g. Tuples) with 'asDBObject'... but be careful, as it can get wonky if you don't have valid pairs inside:

```
val p = ("x" -> 5, "y" -> 15, "z" -> 25).asDBObject
// p: com.mongodb.DBObject = { "x" : 5 , "y" : 15 , "z" : 25}
```

Which pretty much covers working sanely from Scala with Mongo's `DBObject`; from here you should be able to work out the rest yourself... from Scala's side it's just a scala.collection.mutable.Map[String, AnyRef]. Implicits are hard - let's go querying!

2.2.5 Querying with Casbah

I'm not going to wax lengthily and philosophically on the insertion of data. We'll cover updates and such in a bit, but let's insert a few items just to get started with. It should be pretty straightforward:

```
val mongoColl = MongoConnection()("casbah_test")("test_data")
val user1 = MongoDBObject("user" -> "bwmcadams",
                          "email" -> "~~brendan~~<AT>10genDOTcom")
val user2 = MongoDBObject("user" -> "someOtherUser")
mongoColl += user1
mongoColl += user2
mongoColl.find()
// com.mongodb.casbah.MongoCursor =
// MongoCursor{Iterator[DBObject] with 2 objects.}

for { x <- mongoColl} yield x
/* Iterable[com.mongodb.DBObject] = List(
    { "_id" : { "$oid" : "4c3e2bec521142c87cc10fff"} ,
      "user" : "bwmcadams" ,
      "email" : "~~brendan~~<AT>10genDOTcom"},
     { "_id" : { "$oid" : "4c3e2bec521142c87dc10fff"} ,
      "user" : "someOtherUser"}
 ) */
```

As we mentioned in passing before, you can get a cursor back explicitly via `find`, or treat the `MongoCollection` object just like a monad. For now, you need to use `find` to get a true query, but it returns an `Iterator[DBObject]` - which can also be handled monadically.

If you wanted to go in and find a particular item, it works much as you'd expect from the Java driver:

```
val q: DBObject = MongoDBObject("user" -> "someOtherUser")
val cursor = mongoColl.find(q)
// cursor: com.mongodb.casbah.MongoCursor =
//    MongoCursor{Iterator[DBObject] with 1 objects.}
val user = mongoColl.findOne(q)
// user: Option[com.mongodb.DBObject] =
/* Some({ "_id" : { "$oid" : "4c3e2bec521142c87dc10fff"} ,
        "user" : "someOtherUser"}) */
```

The former case returns a Cursor with 1 item - the latter, being a `findOne`, gives us just the row that matches. We use `Option[_]` for `findOne` for protection from passing `null` around (I hate `null`) - If it *doesn't* find anything, findOne returns `None`. A clever hack might be:

```
mongoColl.findOne(q).foreach { x =>
    // do some work if you found the user...
    println("Found a user! %s".format(x("user")))
}
```

You can also limit the fields returned, etc just like with the Java driver. If we wanted to see all the users, retrieving just the username:

```
val q  = MongoDBObject.empty
val fields = MongoDBObject("user" -> 1)
for (x <- mongoColl.find(q, fields)) println(x)
```

As is standard with MongoDB, you always get back the `_id` field, whether you want it or not. You may also note one other "Scala 2.8" collection feature above - `empty`. `MongoDBObject.empty` will always give you back a... you guessed it... Empty `DBObject`. This tends to be useful working with MongoDB with certain tasks such as an empty

query (all entries) with limited fields. I didn't cast either fields or q back to `DBObject` to demonstrate that find will accept them without remark.

There's one last thing I want to touch on for simple queries, which leads us into the wider world of Casbah: fluid query syntax. Casbah allows you in many cases to construct DBObjects on the fly using MongoDB query operators. If we wanted to find all of the entries which had an email address defined we can use `$exists`:

```
val q = "email" $exists true
// q: (String, com.mongodb.DBObject) =
// (email,{ "$exists" : true})
val users = for (x <- mongoColl.find(q)) yield x
assert(users.size == 1)
```

Unless you messed with the sample data we've been assembling thus far, that assertion should pass. `$exists` is a MongoDB Query Expression Operator designed to let you specify that the field must exist. This is obviously useful in a schemaless setup - we didn't specify an email address for one of our two users.

That said, the use of `"email" $exists true` as bareword code which just "worked" as a Mongo `DBObject` shouldn't go w

- "Bareword" Query Operators
- "Core" Query Operators

These are defined in *query/BarewordOperators.scala*' and *query/CoreOperators.scala*, respectively. A Bareword query operator is

- $set - `$set ("foo" -> 5, "bar" -> 28) // DBObject = { "$set" : { "foo" : 5 , "bar" : 28}}`
- $unset - `$unset ("foo", "bar") // DBObject = { "$unset" : { "foo" : 1 , "bar" : 1}}`
- $inc - `$inc ("foo" -> 5.0, "bar" -> 1.6) // DBObject = { "$inc" : { "foo" : 5.0 , "bar" : 1.6}}` (NOTE: Pick a single numeric type and stick with it or the setup fails.)
- And the so-called Array Operators: *$push*, *pushAll*, *$addToSet*, *$pop*, *$pull*, and *$pullAll*

There is solid ScalaDoc for each operator. All of these can be chained inside a larger query as well. The "Core" operators are the ones you're more likely to encounter regularly (These are doced as well) and all of MongoDB's current operators *with the exception of $or and $type* are supported (and tested). If you wanted to find all of the users whose username is **not** *bwmcadams*:

```
mongoColl.findOne("user" $ne "bwmcadams")
/* Option[com.mongodb.DBObject] =
    Some({ "_id" : { "$oid" : "4c3e2bec521142c87dc10fff"} ,
         "user" : "someOtherUser"})
*/
```

You also can chain operators for an "and" type query... I often find myself looking for ranges of value. This is easily accomplished through chaining:

```
val rangeColl = mongoConn("casbah_test")("rangeTests")
rangeColl += MongoDBObject("foo" -> 5)
rangeColl += MongoDBObject("foo" -> 30)
rangeColl += MongoDBObject("foo" -> 35)
rangeColl += MongoDBObject("foo" -> 50)
rangeColl += MongoDBObject("foo" -> 60)
rangeColl += MongoDBObject("foo" -> 75)
rangeColl.find("foo" $lt 50 $gt 5)
// com.mongodb.casbah.MongoCursor =
```

```
//  MongoCursor{Iterator[DBObject] with 2 objects.}
for (x <- rangeColl.find("foo" $lt 50 $gt 5) ) println(x)
// { "_id" : { "$oid" : "4c42426f30daeca8efe48de8"} , "foo" : 30}
// { "_id" : { "$oid" : "4c42427030daeca8f0e48de8"} , "foo" : 35}
for (x <- rangeColl.find("foo" $lte 50 $gt 5) ) println(x)
// { "_id" : { "$oid" : "4c42426f30daeca8efe48de8"} , "foo" : 30}
// { "_id" : { "$oid" : "4c42427030daeca8f0e48de8"} , "foo" : 35}
// { "_id" : { "$oid" : "4c42427330daeca8f1e48de8"} , "foo" : 50}
```

You can get the idea pretty quickly that with these "core" operators you can do some pretty fantastic stuff. One corner I hit late in the development process (And felt stupid for not finding before) was... What if I want fluidity on multiple fields? In that case, parentheses-wrap your blocks and add them:

```
("foo" $lt 50 $gt 5) ++ ("bar" $gte 9)
/* com.mongodb.DBObject =
    { "foo" : { "$lt" : 50 , "$gt" : 5} ,
      "bar" : { "$gte" : 9}} */
```

You'll probably find it saner to start using either the Builder pattern or Additive MongoDBObject interface for multiples however:

```
MongoDBObject("baz" -> 5, "foo" $gte 5, "x" -> "y", "n" -> "r")
/* java.lang.Object with com.mongodb.casbah.MongoDBObject =
       Map((baz,5), (foo,{ "$gte" : 5}), (x,y), (n,r)) */
// Or with a builder pattern
val bldr = MongoDBObject.newBuilder
bldr += "baz" -> 5
bldr += "foo" $gte 5
bldr += "x" -> "y"
bldr += "n" -> "r"
bldr.result
/* java.lang.Object with com.mongodb.casbah.MongoDBObject =
    Map((baz,5), (foo,{ "$gte" : 5}), (x,y), (n,r)) */
```

If you really feel the need to use ++, please note that you'll need to group pairs for multiple ++'s - I can't get it to work properly otherwise on my end (if you have any suggestions please let me know). This doesn't work:

```
("baz" -> 5) ++ ("foo" $gte 5) ++ ("x" -> "y")
/* <console>:9: error: overloaded method value ++ with alternatives:
        [SNIP]
 cannot be applied to ((java.lang.String, java.lang.String))
       ("baz" -> 5) ++ ("foo" $gte 5) ++ ("x" -> "y")
                       ^
*/
```

However, this does:

```
("baz" -> 5) ++ (("foo" $gte 5) ++ ("x" -> "y"))
/* com.mongodb.DBObject =
   { "foo" : { "$gte" : 5} , "baz" : 5 , "x" : "y"} */
("baz" -> 5) ++ (("foo" $gte 5) ++ (("x" -> "y") ++ ("n" -> "r")))
/* com.mongodb.DBObject =
  { "foo" : { "$gte" : 5} , "baz" : 5 , "n" : "r" , "x" : "y"} */
```

So Your Mileage May Vary. If you'd like to see all the possible query operators, I recommend you review *query/CoreOperators.scala*.

2.2.6 GridFS with Casbah

Personally, I found some frustration working with the Java GridFS interface - there was a need after creating a new file to explicitly `save`, and it was a bit too un-Scala-ble. To that end, I created a few wrappers to GridFS to make it act more like Scala, and favor a **Loan** style pattern which automatically saves for you once you're done (Given a curried function).

MongoDB's GridFS system allows you to store files within MongoDB - MongoDB chunks the file in a way that allows massive scalability (I've been told the maximum file size is 16 **Exabytes**). Casbah's Scala version of GridFS supports creating files using `Array[Byte]`, `java.io.File` and `java.io.InputStream` (I had some problems with `scala.io.Source` and it's currently disabled). GridFS works in terms of *buckets*. A bucket is a base collection name, and creates two actual collections: *<bucket>.files* and *<bucket>.chunks*. *files* contains the object metadata, while *chunks* contains the actual binary chunks of the files. If you're interested, you can learn more in the GridFS Specification. To work with GridFS you need to provide a connection object, and define the *bucket* name (without *.chunks*/*.files*); however, by default (AKA if you don't specify a bucket) MongoDB uses a bucket called "fs". You also need to import our GridFS objects:

```
import com.mongodb.casbah.gridfs.Imports._
val gridfs = GridFS(mongoConn) // creates a GridFS handle on ``fs``
```

The `gridfs` object is very similar to a `MongoCollection` - it has `find` & `findOne` methods and is Iterable. We're going to pull some sample code from the GridFS unit test.

Creating a new file with the **loan** style is easy:

```
val logo = new FileInputStream("casbah-gridfs/src/test/resources/powered_by_mongo.png")
gridfs(logo) { fh =>
  fh.filename = "powered_by_mongo.png"
  fh.contentType = "image/png"
}
```

We have defined a new file in GridFS from the `FileInputStream`, set it's filename and content type and automatically saved it. The expected function type of the `apply` method is `type FileWriteOp = GridFSInputFile => Unit`. One Note: Due to hardcoding in the Java GridFS driver the Joda Time serialization hooks break **hard** with GridFS. It tries to explicitly cast certain date fields as a `java.util.Date` and fails miserably. To that end, on all find ops I explicitly unload the Joda Time deserializers and reload them when I'm done (if they were loaded before we started). This allows GridFS to always work but *MAY* cause thread safety issues - e.g. if you have another non-GridFS read happening at the same time in another thread at the same time, it may fail to deserialize BSON Dates as Joda DateTime - and blow up. I'm hoping to have this fixed in a future release of the Java driver.

Finally, before I leave you to explore on your own, I'll show you retrieving a file. It should look familiar:

```
val file = gridfs.findOne("powered_by_mongo.png")
```

`find` and `findOne` can take `DBObject` like on `Collection` objects, but you can also pass a filename as a `String`. It is possible to have multiple files with the same filename as far as I know, so findOne would only return the first it found. The returned object is not a DBObject - it is a *GridFSDBFile*. From here, you should be able to explore and have fun on your own - stay out of trouble!

2.2.7 Map/Reduce with Casbah

Coming Soon.