Casbah (MongoDB + Scala Toolkit) Documentation

Release 2.0rc1

Brendan W. McAdams & 10gen, Inc.

December 27, 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.

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.

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

You may also download this documentation in other formats.

- ePub
- PDF

CHAPTER

ONE

GETTING STARTED

1.1 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).

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

The latest build as of December 27, 2010 is 2.0rc1, 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. The following dependency manager information uses the master artifact which downloads and uses *all* of Casbah's modules by default.

1.1.2 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)

1.1.3 Setting Up Maven

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

Scala 2.8.0 users:

```
<dependency>
    <groupId>com.mongodb.casbah<groupId>
    <artifactId>casbah_2.8.0<artifactId>
    <version>2.0rc1<version>
<dependency>
```

Scala 2.8.1 users:

```
<dependency>
    <groupId>com.mongodb.casbah<groupId>
    <artifactId>casbah_2.8.1<artifactId>
    <version>2.0rc1<version>
<dependency>
```

1.1.4 Setting Up Ivy (w/ Ant)

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

Scala 2.8.0 users:

```
<dependency org="com.mongodb.casbah" name="casbah_2.8.0" rev="2.0rc1"/>
```

Scala 2.8.1 users:

```
<dependency org="com.mongodb.casbah" name="casbah_2.8.1" rev="2.0rc1"/>
```

1.1.5 Setting up SBT

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

```
val casbah = "com.mongodb.casbah" %% "casbah" % "2.0rc1"
```

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.mongodb.casbah" % "casbah_2.8.0" % "2.0rc1"
// Scala 2.8.1
val casbah = "com.mongodb.casbah" % "casbah_2.8.1" % "2.0rc1"
```

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

1.2 Migrating to Casbah 2.0 from Casbah 1.x

If you used Casbah before, and are looking to migrate from Casbah 1.x to Casbah 2.x there are some things which have changed and you should be aware of to effectively update your code.

1.2.1 Base Package Name

For starters, the base package has changed. The now abandoned 1.1.x branch which became 2.0 was already doing a package change, and with 2.0 Casbah has become a supported MongoDB project. As a result, Casbah's package has changed for the 2.0 release and you will need to update your code accordingly:

Casbah 1.0.x	Casbah 1.1.x (never released)	Casbah 2.0
com.novus.casbah.mongodb	com.novus.casbah	com.mongodb.casbah

1.2.2 Removed Features

A number of features existed in Casbah as artifacts of early prototyping. They were buggy, poorly tested and because of their nature often introduced weird problems for users.

As a result, they have been removed now that replacement versions of their functionality exist.

Removal of Implicit Tuple -> DBObject Conversions

Previously, it was possible with Casbah to cast Tuples to DBObject:

```
val x: DBObject = ("foo" -> "bar", "x" -> 5, "y" -> 238.1)
```

This feature was provided by implicit conversions which attempt to target *Product* which is the base class of all Tuples. Unfortunately, this functionality was often unreliable and targeted the wrong things for conversion (Such as instances of *Option[_]*). After a lot of evaluation and attempts to create a better approach a decision was made to remove this feature. Casbah 2.0 includes wrappers for DBObject which follow Scala 2.8's Collection interfaces including Scala compatible builders and constructors. As such, the same previous syntax is possible by passing the Tuple pairs to *MongoDBObject.apply*:

```
val x: DBObject = MongoDBObject("foo" -> "bar", "x" -> 5, "y" -> 238.1)
/* x: com.mongodb.casbah.Imports.DBObject = { "foo" : "bar" , "x" : 5 , "y" : 238.1} */
val y = MongoDBObject("foo" -> "bar", "x" -> 5, "y" -> 238.1)
/* y: com.mongodb.casbah.commons.Imports.DBObject = { "foo" : "bar" , "x" : 5 , "y" : 238.1} */
```

We also provide a builder pattern which follows Scala 2.8's Map Builder:

```
val b = MongoDBObject.newBuilder
/* b: com.mongodb.casbah.commons.MongoDBObjectBuilder = com.mongodb.casbah.commons.MongoDBObjectBuil
b += "x" -> 5
b += "y" -> 238.1
b += "foo" -> "bar"
val x: DBObject = b.result
/* x: com.mongodb.casbah.commons.Imports.DBObject = { "x" : 5 , "y" : 238.1 , "foo" : "bar"} */
```

Finally, any Scala map can still be cast to a DBObject without issue:

```
val x: DBObject = Map("foo" -> "bar", "x" -> 5, "y" -> 238.1)
/* x: com.mongodb.casbah.Imports.DBObject = { "foo" : "bar" , "x" : 5 , "y" : 238.1} */
```

It is *still* possible to use Tuples in the *Query DSL* however, as there is less need for broad implicit conversions to accomplish that functionality.

batchSafely Removed

Casbah 1.1.x introduced a *batchSafely* command which used the Java Driver's *requestStart()*, *requestDone()* and *getPrevErrors()* functions. MongoDB is deprecating the use of *getPrevErrors()* and as such, Casbah has removed the

functionality in anticipation of that feature going away in a near future release.

1.2.3 New Features

Query DSL Operators

Casbah previously lagged behind the official MongoDB server in supported *Query DSL $ Operators*. As of 2.0, all *$ Operators* currently documented as supported in MongoDB are provided. A list of some of the new operators added in 2.0 include:

- $slice
- $or
- $not
- $each (*special operator only supported nested inside :dochub:'$addToSet'*)
- $type (*Uses type arguments and class manifests to allow a nice fluid Scala syntax*)
- $elemMatch
- Array Operators
- All GeoSpatial Operators including $near and $within

Further, the DSL system has been completely overhauled. As part of adding test coverage a number of edge cases were discovered with the DSL that caused inconsistent behavior. The majority of the Query DSL should continue to work the same, but we have started moving to the use of Type Classes and Context Boundaries to limit what a valid input to any given operator is (2.1 will include expanded use of these merged with custom serializers).

New syntax for $not

In order to fix a number of bugs and readability issues with the $not operator, it has been modified.

Previously, the correct syntax for using $not was:

```
"foo".$not $gte 15 $lt 35.2 $ne 16
```

With Casbah 2.0, this syntax has been modified to be more clear to both the developer *and* the compiler:

```
"foo" $not { _ $gte 15 $lt 35.2 $ne 16 }
```

The same syntax is supported for the special version of $pull which allows for nested operator tests.

1.2.4 General Code Cleanup

There has been a lot of general code cleanup in this release and while many features appear the same externally they may have been refactored.

1.2.5 Casbah Modules

While Casbah has a large stable of features, some users (such as those using a framework like Lift which already provides MongoDB wrappers) wanted access to certain parts of Casbah without importing the whole system. As a result, Casbah has been broken out into several modules which make it easier to pick and choose the features you want.

If you use the individual modules you'll need to use the import statement from each of these. If you use the import statement from the *casbah-core* module, everything except GridFS will be imported (not everyone uses GridFS so we don't load it into memory & scope unless it is needed). The module names can be used to select which dependencies you want from maven/ivy/sbt, as we publish individual artifacts. If you import just *casbah*, this is a master pom which includes the whole system and can be used just like 1.1.x was (that is to say, you can pretend the module system doesn't exist more or less).

This is the breakdown of dependencies and packages for the new system:

Module	Package	Dependencies
casbah-commons ("Commons") **NOTES** Provides Scala-friendly :dochub:DBObject & :dochub:DBList implementations as well as Implicit conversions for Scala types	com.mongodb.casbah.commons	mongo-java-driver, scalaj-collection, scalaj-time, JodaTime, slf4j-api
casbah-query ("Query DSL") **NOTES** Provides a Scala syntax enhancement mode for creating MongoDB query objects using an Internal DSL supporting Mongo *$ Operators*	com.mongodb.casbah.query	*casbah-commons* along with its dependencies transitively
casbah-core ("Core") **NOTES** Provides Scala-friendly wrappers to the Java Driver for connections, collections and MapReduce jobs	com.mongodb.casbah	*casbah-commons* and *casbah-query* along with their dependencies transitively
casbah-gridfs ("GridFS") **NOTES** Provides Scala enhanced wrappers to MongoDB's GridFS filesystem	com.mongodb.casbah.gridfs	*casbah-core* and *casbah-commons* along with their dependencies transitively

We cover the import of each module in their appropriate tutorials, but each module has its own *Imports* object which loads all of its necessary code. By way of example both of these statements would import the Query DSL:

```
// Imports core, which grabs everything including Query DSL
import com.mongodb.casbah.Imports._
// Imports just the Query DSL along with Commons and its dependencies
import com.mongodb.casbah.query.Imports._
```

CHAPTER

TWO

TUTORIAL: USING CASBAH

2.1 Import the Driver

Now that you've added Casbah to your project, it should be available for import. For this tutorial, we're going to import the *Core* module which brings in all of Casbah's functionality (except *GridFS*) As of this writing, *Core* 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 Implicit conversions 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. The `Imports` call will make common types such as `DBObject`, `MongoConnection` and `MongoCollection` available. *Core*'s `Imports` also run the imports from *Commons* and the *Query DSL*. 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

As we mentioned, 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. The general idea is that common Java types (such as ArrayList) will be returned as the equivalent Scala type.

As many Scala developers tend to prefer Joda time over JDK Dates, you can also explicitly enable serialization and deserialization of them (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()
```

It is also possible to create your own custom type serializers and deserializers. See *Custom Serializers and Deserializers*.

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 won't require additional importing. These all return an instance of the `MongoConnection` class, which 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 also provides an `apply()` 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 (without a uqery). 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 = 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")
val newObj = builder.result
// 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. As 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:

```
// apply returns AnyRef
val x = newObj("OMG")
/* x: AnyRef = Ponies  */

// Can't put AnyRef in a String
val y: String = newObj("OMG")
/* error: type mismatch;
   found   : AnyRef
   required: String
*/
```

/Casbah provides two methods to help automatically infer a type from you however — 'as[A]' which is the typed equivalent of 'apply', and 'getAs[A]' which is the typed equivalent of 'get' returns 'Option[A]'/ These functions are available on ANY `DBObject` — not just ones you created through the `MongoDBObject` function (There is an implicit conversion loaded that can Pimp any `DBObject` as `MongoDBObject`. 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 Exception("OMG! WTF? BBQ!"))
/* java.lang.Exception: OMG! WTF? BBQ! */
// Or you can use the chain ops available on Option
val omgWtfFailChain = newObj.getAs[String]("OMGWTF") orElse ( throw new Exception("Chain Fail.") )
/* java.lang.Exception: Chain Fail. */
```

Combining Multiple DBObjects

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)) */
val zCast: DBObject = newObj ++ obj2
/* zCast: com.mongodb.casbah.Imports.DBObject =
  { "foo" : "bar" , "mmm" : "bacon" , "spam" : "eggs" , "pie" : 3.14 , "n" : "212" , "OMG" : "Ponies
```

Due to some corners in Scala's Map traits some base methods return Map instead of the more appropriate `this.type`, and you'll need to cast to DBObject explicitly. 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"}
 */
```

This 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 MongoDBList - Mongo-friendly List implementation

While Scala's builtin list and sequence types can be serialized to MongoDB, in some cases (especially with Casbah's DSL) it is easier to work with `MongoDBList`, which is built for creating valid Mongo lists. `MongoDBList`, like `MongoDBObject`, follows the Scala 2.8 collections pattern. It provides an object constructor as well as a builder:

```
val builder = MongoDBList.newBuilder
builder += "foo"
builder += "bar"
builder += "x"
builder += "y"
val newLst = builder.result
/* newLst: com.mongodb.BasicDBList = [ "foo" , "bar" , "x" , "y"] */
```

Apart from that it's a pretty standard Scala list.

2.2.6 Querying with Casbah

I'm not going to wax lengthily and philosophically on the insertion of data; if you need a bit more guidance you should take a look at the MongoDB Tutorial. 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 = 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. For example, 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)
/* { "_id" : { "$oid" : "4d190356b9d8ba42efa80898"} , "user" : "bwmcadams"}
   { "_id" : { "$oid" : "4d190356b9d8ba42f0a80898"} , "user" : "someOtherUser"} */
```

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.

Fluid Querying with Casbah's DSL

There's one last big feature you should be familiar with to get the most out 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. What if I want fluidity on multiple fields? In that case, use the ++ additivity operator to combine multiple blocks.:

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

Just remember that when you call ++ with *DBObjects* you get a *Map* instance back and you'll need to cast it.

If you really feel the need to use ++ with a mix of DSL and bare matches, we provide additive support for `<key> -> <value>` Tuple pairs. You should make the query operator calls *first*:

```
val qMix = ("baz" -> 5) ++ ("foo" $gte 5) ++ ("x" -> "y")
/* error: value ++ is not a member of (java.lang.String, Int)
   val qMix = ("baz" -> 5) ++ ("foo" $gte 5) ++ ("x" -> "y") */
```

The operator is chained against the result of DSL operators (which incidentally properly return a `DBObject`):

```
val qMix = ("foo" $gte 5) ++ ("baz" -> 5) ++ ("x" -> "y")
/* qMix: com.mongodb.casbah.commons.Imports.DBObject =
   { "foo" : { "$gte" : 5} , "baz" : 5 , "x" : "y"} */

val qMix2 = ("foo" $gte 5 $lte 10) ++ ("baz" -> 5) ++ ("x" -> "y") ++ ("n" -> "r")
/* qMix2: com.mongodb.casbah.commons.Imports.DBObject =
   { "foo" : { "$gte" : 5 , "$lte" : 10} , "baz" : 5 , "n" : "r" , "x" : "y"} */
```

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

2.2.7 GridFS with Casbah

Casbah contains 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".

Because many projects don't use GridFS at all, we don't import it by default. If you want to use GridFS you'll need to import our GridFS objects:

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

Then create your new GridFS handle:

```
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 we explicitly unload the Joda Time deserializers and reload them when we're 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. Be careful — generally we don't recommend mixing Joda Time and GridFS in the same JVM at the moment.

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!