Casbah (MongoDB + Scala Toolkit) Documentation

Release 2.6.0

10gen, Inc.

April 23, 2013

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

INSTALLATION

Casbah is released to the Sonatype repository, the latest Casbah build as is 2.6.0 and supports the follwoing scala versions: 2.9.1, 2.9.2 and 2.10.0.

The easiest way to install the latest Casbah driver (2.6.0) is by using sbt - the Scala Build Tool.

1.1 Setting up via sbt

Once you have your sbt project setup - see the sbt setup guide for help there.

Add Casbah to sbt to your *./build.sbt* file:

```
libraryDependencies += "org.mongodb" %% "casbah" % "2.6.0"
```

Note: 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.

To test your installation load the sbt console and try importing casbah:

```
$ sbt console
scala> import com.mongodb.casbah.Imports._
import com.mongodb.casbah.Imports._
```

Once you have installed correctly got to the tutorial or casbah modules

1.2 Problem solving

If sbt can't find casbah then you may have an older version of sbt and will need to add the sonatype resolvers to your *./build.sbt* file:

```
// For stable releases
resolvers += "Sonatype releases" at "https://oss.sonatype.org/content/repositories/releases"
// For SNAPSHOT releases
resolvers += "Sonatype snapshots" at "https://oss.sonatype.org/content/repositories/snapshots"
```

1.3 Alternative installation methods

You can install Casbah with maven, ivy or from source - see the alternative install documentation.

CHAPTER

TWO

ALTERNATIVE INSTALLATION METHODS FOR CASBAH

Note: There is no single jar as Casbah is split into multiple modules. See the casbah modules documentation for more information.

2.1 Using Dependency/Build Managers

First, you should add the package repository to your Dependency/Build Manager. Our releases & snapshots are currently hosted at Sonatype; they should eventually sync to the Central Maven repository.:

```
https://oss.sonatype.org/content/repositories/releases/  /* For Releases */
https://oss.sonatype.org/content/repositories/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.

2.2 Setting Up Maven

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

Please substitute $SCALA_VERSION$ with your Scala version (we support 2.9.x+)

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>casbah_$SCALA_VERSION$</artifactId>
    <version> 2.6.0 </version>
    <type>pom</type>
<dependency>
```

2.3 Setting Up Ivy (w/ Ant)

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

Please substitute $SCALA_VERSION$ with your Scala version (we support 2.9.x+)

```
<dependency org="org.mongodb" name="casbah_$SCALA_VERSION$" rev="2.6.0"/>
```

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

The builds are published to the Sonatype.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
```

The *master* branch is once again the leading branch suitable for snapshots and releases and should be considered (and kept) stable.

CHAPTER

THREE

TUTORIAL: USING CASBAH

3.1 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`, `MongoClient` 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.

3.2 Briefly: Automatic Type Conversions

As we mentioned, as soon as you construct a `MongoClient` 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.common.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.common.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.common.conversions.scala._
DeregisterConversionHelpers()
RegisterConversionHelpers()
```

3.2.1 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.

3.2.2 Connecting to MongoDB

The core Connection class as you may have noted above is `com.mongodb.casbah.MongoClient`. There are two ways to create an instance of it. First, you can invoke `.asScala` from a MongoDB builtin Connection (`com.mongodb.MongoClient`). 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
scala> val mongoClient =  MongoClient()
mongoClient: com.mongodb.casbah.MongoClient ...

// connect to "mongodb01" host, default port
scala> val mongoClient =  MongoClient("mongodb01")
mongoClient: com.mongodb.casbah.MongoClient ...

// connect to "mongodb02" host, port 42017
scala> val mongoClient =  MongoClient("mongodb02", 42017)
mongoClient: com.mongodb.casbah.MongoClient ...

// connect using mongdb's challenge response authentication
scala> val uri = new MongoClientURI("mongodb://username:pwd@localhost/?authMechanism=MONGODB-CR")
scala> val mongoClient =  MongoClient(uri)
mongoClient: com.mongodb.casbah.MongoClient ...

// connect using mongdb's GSSAPI authentication
scala> val uri = new MongoClientURI("mongodb://username%40domain@kdc.example.com/?authMechanism=M
scala> val mongoClient =  MongoClient(uri)
mongoClient: com.mongodb.casbah.MongoClient ...
```

If you imported `Imports._`, you already have `MongoClient` in scope and won't require additional importing. These all return an instance of the `MongoClient` 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()`:

```
scala> val mongoDB = mongoClient("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:

```
scala> val mongoColl = mongoClient("casbah_test")("test_data")
mongoColl: com.mongodb.casbah.MongoCollection = MongoCollection()
```

Note: `MongoClient` was added to the Java driver in 2.10 as the default connection class for MongoDB. Older Casbah code may use `MongoConnection` which should be updated to use `MongoClient`.

3.2.3 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. Be-

ginning iteration on the `MongoCollection` instance is fundamentally equivalent to invoking `find` on the `MongoCollection` (without a query). We'll return to this after we discuss working with `MongoDBObjects` and inserting data...

3.2.4 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 for 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

```
scala> val newObj = MongoDBObject("foo" -> "bar",
     |                            "x" -> "y",
     |                            "pie" -> 3.14,
     |                            "spam" -> "eggs")

newObj: com.mongodb.casbah.commons.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

```
scala> val builder = MongoDBObject.newBuilder
scala> builder += "foo" -> "bar"
scala> builder += "x" -> "y"
scala> builder += ("pie" -> 3.14)
scala> builder += ("spam" -> "eggs", "mmm" -> "bacon")
builder.type = com.mongodb.casbah.commons.MongoDBObjectBuilder@...

scala> val newObj = builder.result
newObj: com.mongodb.casbah.commons.Imports.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 +=

```
scala> newObj += "OMG" -> "Ponies!"

com.mongodb.casbah.commons.MongoDBObject =
  { "foo" : "bar" , "x" : "y" , "pie" : 3.14 , "spam" : "eggs" ,
    "mmm" : "bacon" , "OMG" : "Ponies!"}

scala> newObj += "x" -> "z"

com.mongodb.casbah.commons.MongoDBObject =
  { "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
scala> val x = newObj("OMG")
x: AnyRef = Ponies!

// Can't put AnyRef in a String
scala> val y: String = newObj("OMG")
<console>:12: error: type mismatch;
 found   : AnyRef
 required: String
       val y: String = newObj("OMG")

// Scala can cast for you if type is valid
scala> val xStr = newObj.as[String]("OMG")
xStr: String = Ponies!
```

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:

```
scala> val foo = newObj.getAs[String]("foo")
foo: Option[String] = Some(bar)
scala> val omgWtf = newObj.getAs[String]("OMGWTF")
omgWtf: Option[String] = None
scala> 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
scala> 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

```
scala> val obj2 = MongoDBObject("n" -> "212")

obj2: com.mongodb.casbah.commons.Imports.DBObject = { "n" : "212"}

scala> val z = newObj ++ obj2

z: com.mongodb.casbah.commons.Imports.DBObject =
  { "foo" : "bar" , "x" : "z" , "pie" : 3.14 , "spam" : "eggs" ,
    "mmm" : "bacon" , "OMG" : "Ponies!" , "n" : "212"}

scala> val zCast: DBObject = newObj ++ obj2

zCast: com.mongodb.casbah.Imports.DBObject =
  { "foo" : "bar" , "x" : "z" , "pie" : 3.14 , "spam" : "eggs" ,
    "mmm" : "bacon" , "OMG" : "Ponies!" , "n" : "212"}
```

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 ex-

plicitly 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

```
scala> z.asDBObject

com.mongodb.casbah.commons.Imports.DBObject =
  { "foo" : "bar" , "x" : "z" , "pie" : 3.14 , "spam" : "eggs" ,
    "mmm" : "bacon" , "OMG" : "Ponies!" , "n" : "212"}

val zDBObj: DBObject = z

zDBObj: com.mongodb.casbah.Imports.DBObject =
  { "foo" : "bar" , "x" : "z" , "pie" : 3.14 , "spam" : "eggs" ,
    "mmm" : "bacon" , "OMG" : "Ponies!" , "n" : "212"}
```

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!

3.2.5 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

```
scala> val builder = MongoDBList.newBuilder
scala> builder += "foo"
scala> builder += "bar"
scala> builder += "x"
scala> builder += "y"
builder.type = com.mongodb.casbah.commons.MongoDBListBuilder@...

scala> val newLst = builder.result

newLst: com.mongodb.BasicDBList = [ "foo" , "bar" , "x" , "y"]
```

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

3.2.6 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

```
scala> val mongoColl = MongoClient()("casbah_test")("test_data")
scala> val user1 = MongoDBObject("user" -> "bwmcadams",
     |                           "email" -> "~~brendan~~<AT>10genDOTcom")
scala> val user2 = MongoDBObject("user" -> "someOtherUser")
scala> mongoColl += user1
scala> mongoColl += user2
scala> mongoColl.find()

mongoColl.CursorType = non-empty iterator

scala> 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

```
scala> val q = MongoDBObject("user" -> "someOtherUser")
scala> val cursor = mongoColl.find(q)

cursor: mongoColl.CursorType = non-empty iterator

scala> val user = mongoColl.findOne(q)

Option[mongoColl.T] = Some(
  { "_id" : { "$oid" : "50cb1dc50cf24a7d3562412c"} ,
    "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

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

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

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

{ "_id" : { "$oid" : "50cb1dc50cf24a7d3562412b"} , "user" : "bwmcadams"}
{ "_id" : { "$oid" : "50cb1dc50cf24a7d3562412c"} , "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`

```
scala> val q = "email" $exists true

q: com.mongodb.casbah.query.Imports.DBObject
  with com.mongodb.casbah.query.dsl.QueryExpressionObject =
  { "email" : { "$exists" : true}}

scala> val users = for (x <- mongoColl.find(q)) yield x
scala>     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 without comment. Casbah provides a powerful *fluid query syntax* to allow you to operate with MongoDB much like you'd expect to work in the JavaScript shell. We drop much of the excess nested object syntax to simplify your code. I find that the use of these expression operators lets me rapidly put queries together that closely match how I'd work with MongoDB in Javascript or Python. Most of the MongoDB Query Expression Operators are supported (The exceptions being new ones I haven't added support yet through indolence). There are two "Essential" types of Query Operators from the standpoint of Casbah:

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

These are defined in `query/BarewordOperators.scala` and `query/CoreOperators.scala`, respectively. A Barev

- $set

```
scala> $set ("foo" -> 5, "bar" -> 28)

com.mongodb.casbah.query.Imports.DBObject =
  { "$set" : { "foo" : 5 , "bar" : 28}}
```

- $unset

```
scala> $unset ("foo", "bar")

com.mongodb.casbah.query.Imports.DBObject =
  { "$unset" : { "foo" : 1 , "bar" : 1}}
```

- $inc

```
scala> $inc ("foo" -> 5.0, "bar" -> 1.6)

com.mongodb.casbah.query.Imports.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*

```
scala> mongoColl.findOne("user" $ne "bwmcadams")

Option[mongoColl.T] = Some(
  { "_id" : { "$oid" : "50cb1dc50cf24a7d3562412c"} ,
    "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

```
scala> val rangeColl = mongoClient("casbah_test")("rangeTests")
scala> rangeColl += MongoDBObject("foo" -> 5)
scala> rangeColl += MongoDBObject("foo" -> 30)
scala> rangeColl += MongoDBObject("foo" -> 35)
scala> rangeColl += MongoDBObject("foo" -> 50)
scala> rangeColl += MongoDBObject("foo" -> 60)
```

```
scala> rangeColl += MongoDBObject("foo" -> 75)
scala> rangeColl.find("foo" $lt 50 $gt 5)

rangeColl.CursorType = non-empty iterator

scala> for (x <- rangeColl.find("foo" $lt 50 $gt 5) ) println(x)

{ "_id" : { "$oid" : "50cb28760cf24a7d3562412e"} , "foo" : 30}
{ "_id" : { "$oid" : "50cb28760cf24a7d3562412f"} , "foo" : 35}

scala> for (x <- rangeColl.find("foo" $lte 50 $gt 5) ) println(x)

{ "_id" : { "$oid" : "50cb28760cf24a7d3562412e"} , "foo" : 30}
{ "_id" : { "$oid" : "50cb28760cf24a7d3562412f"} , "foo" : 35}
{ "_id" : { "$oid" : "50cb28760cf24a7d35624130"} , "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.

```
scala> val q: DBObject = ("foo" $lt 50 $gt 5) ++ ("bar" $gte 9)

q: com.mongodb.casbah.Imports.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*:

```
scala> val qMix = ("baz" -> 5) ++ ("foo" $gte 5) ++ ("x" -> "y")
<console>:10: 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`)

```
scala> val qMix = ("foo" $gte 5) ++ ("baz" -> 5) ++ ("x" -> "y")

qMix: com.mongodb.casbah.commons.Imports.DBObject =
  { "foo" : { "$gte" : 5} , "baz" : 5 , "x" : "y"}

scala> 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 , "x" : "y" , "n" : "r"}
```

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

3.2.7 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(mongoClient) // 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!

CHAPTER

FOUR

CASBAH MODULES

While Casbah has many 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 will install all its dependencies, as such there is no single jar file for Casbah.

This is the breakdown of dependencies and packages:

Module	Package	Dependencies
Casbah 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 Commons **NOTES** Provides Scala-friendly DBObject & DBList implementations as well as Implicit conversions for Scala types	com.mongodb.casbah.commons	• mongo-java-driver, • nscala-time, • slf4j-api, • slf4j-jcl
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 their dependencies transitively
Gridfs **NOTES** Provides Scala enhanced wrappers to MongoDB's GridFS filesystem	com.mongodb.casbah.gridfs	casbah-commons and casbah-query 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

FIVE

CHANGELOG

5.1 Changes in Version 2.6.0

- Added suport for GSSAPI SASL mechanism and MongoDB Challenge Response protocol
- Updated support for latest Java driver 2.11.1

5.2 Changes in Version 2.5.1

- Added 2.10.1 support
- Removed reference to scala-tools (SCALA-78)
- Added 2.9.3 support (SCALA-94)
- Removed Specs2 and Scalaz dependencies outside test (SCALA-93)
- Fixed 2.10 support, no need for -Yeta-expand-keeps-star compile flag (SCALA-89)
- Fixed distinct regression (SCALA-92)
- Fixed test data import - now in tests :)

5.3 Changes in Version 2.5.0

- Added support for Scala 2.10.0
- Dropped support for Scala 2.9.0
- Dropped support for Scala 2.8.X
- Updated support for latest Java driver 2.10.1
- Added support for the new MongoClient connection class
- Removed scalaj.collections dependency
- Updated to nscala-time
- Updated the build file
- Added unidoc and updated documentation
- Migrated documentation theme
- Updated MongoDBList to handle immutable params
- Maven Documentation fix (SCALA-71)
- MongoOpLog - uses new MongoClient and defaults to replciaSet oplog database

5.4 Changes in Version 2.4.1

- Fixed QueryDSL imports for "default" (com.mongodb.casbah.Imports) import so that bareword ops like $set and $inc are available.

5.5 Changes in Version 2.4.0

- Hide BasicDBList; now, getAs and As and related will always return a MongoDBList which is a Seq[_]. Enjoy!
- This is an API breakage - you should *never* get back a BasicDBList from Casbah anymore, and asking for one will cause a ClassCastException. This brings us more in line with sane Scala APIs

5.6 Changes in Version 2.3.0

BT/Maven Package change. Casbah is now available in: "org.mongodb" %% "casbah" % "2.3.0"

- Update mongo-java-driver to 2.8.0 release
- Updated to Mongo Java Driver 2.8.0-RC1
- Changed some tests to run sequentially to avoid shared variable races.
- JodaGridFS wasn't properly checked in before.
- Updated MongoOptions to sync up with options provided in Java Driver.
- Pre-Beta milestone (linked against unreleased Java Driver release)
- Dropped Scala 2.8.0 support...
  - 2.1.5-1 is the final Casbah release for 2.8.0; please migrate to Scala 2.8.1 or higher
- SCALA-62: Simple solution - hack the date type on the base class.
  - There is now a JodaGridFS implementation which works cleanly with Joda DateTime and will return them to you
- Backport casbah-gridfs from 3.0
  - Fixes SCALA-45: Allow filename and contentType to be nullable
    - Retrieving filename or contentType on a GridFS File now returns Option[String] when fetched
    - To facilitate sane usage, the loan-pattern/execute-around-resource methods now return the _id of the created file as Option[AnyRef]
- Backports to casbah-core from 3.0
  - SCALA-70: Removed type alias to com.mongodb.WriteConcern and made method args for it explicit, as it was causing a fun post-compile (aka "library compiles, user code doesn't") implosion.
  - added socketKeepAlive option
  - Fixes SCALA-45: Allow filename and contentType to be nullable
  - Retrieving filename or contentType on a GridFS File now returns Option[String] when fetched
  - To facilitate sane usage, the loan-pattern/execute-around-resource methods now return the _id of the created file as Option[AnyRef]
- Backports for QueryDSL
  - Major cleanups and bugfixes to the DSL, it's heavily and fully tested now and much faster/cleaner
  - Added support for $and bareword operator

  - SCALA-30, SCALA-59 - $or is not properly accepting nested values esp. from other DSL constructors
    * Introduced proper type class filter base to fix $or, will implement across other operators next.
  - SCALA-59 - Fix Bareword Query Operators to better target accepted values; should only accept KV Tuple Pairs or DBObjects returned from Core Operators
    * Complete test suites for $and and $nor although they need to be updated to more appropriate contextual examples rather than just "compiles properly"
    * New code logic, fixed $or, $and and $nor for proper nested list operations
    * New :: list cons operator on MongoDBObject to create MongoDBLists on th fly (esp. for DSL)
    * Typesafety kungfu from @jteigen
      · enforce at compile time that type parameters used for casting are not Nothing
      · enforce $pushAll & $pullAll arguments can be converted to Iterable at compile time
      · switched to a type class (AsQueryParam) for queryparams to avoid code duplication
- SCALA-69: Maps saved to DBObject are now eagerly converted to a DBObject, from factory, builder and put methods.
- Always return MongoDBList from Factories/Builders instead of Seq[Any]
- Backports from Casbah 3.0
  - Refactor collections (MongoDBList and MongoDBObject)
  - Use CanBuildFrom properly to compose more appropriate Collection objects
  - As part of above, you should get seq-like objects back from MongoDBList builders & factories instead of the previous BasicDBList; this is part of attempting to "Hide" DBList and let people work with List/Seq
  - SCALA-69: Immediately upon saving any None's will be converted to null inside the DBObject for proper fetching later.
  - Add toString, hashCode and equals methods to DBObject
  - New, refactored tests for DBObject and DBList
    * More typesafety kungfu from @jteigen
      · enforce at *compile time* that type parameters used for casting (as, getAs, getAsOrElse) are not Nothing
- Backport Test Helpers
  - New MongoDB "smart" test helpers for Specs2 and ScalaTest (Thanks Bill Venners for the latter)
- Added SBT Rebel cut, local runner

5.7 Changes in Version 2.1.5.0

- Added support for Scala 2.9.0-1 … As this is a critical fix release against 2.9.0.final, 2.9.0.final is not supported. (Note that SBT, etc requires the artifact specified as 2.9.0-1, not 2.9.0.1)
- Apart from BugFixes this will be the last Casbah release which supports Scala 2.8.0; all future releases will require Scala 2.8.1+ (See 2.8.0 EOL Announcement)
- [2.9.0 only] Adjusted dynamic settings to build against 2.9.0-1 and Casbah 2.1.5.0
- [2.9.0 only] Prototype "Dynamic" module (You must enable Scala's support for Dynamic)
- [2.9.0 only] I seem to have missed project files for SBT and casbah-dynamic
- [2.9.0 only] Tweaks and adjustments to get this building and testing solidly on 2.9.0-1

- Disabled a few tests that weren't passing and known to be 'buggy' in specs1. These are fixed for the upcoming 2.2. release on specs2; they are test bugs rather than Casbah bugs.
- RegEx *not was just flat out wrong - was producing {"foo": {"foo": /<regex>/}} instead of {"foo": {*"not":{//}}
- Added a getAsOrElse method

5.8 Changes in Version 2.1.0

- SCALA-22 Added a dropTarget boolean option to rename collection, which specifies behavior if named target collection already exists, proxies JAVA-238
- Removed resetIndexCache, which has also been removed from the Java Driver
- SCALA-21 Added "set metadata" method to match Java Driver (See Java-261)
- SCALA-20 Updated to Java Driver 2.5
  - See Release Notes: http://groups.google.com/group/mongodb-user/browse_thread/thread/a693ad4fdf9c3731/931f46f7213b6775?show_docid=931f46f7213b6775
- SCALA-21 - Update GridFS to use DBObject views. Holding back full bugfix until we have a 2.5 build to link against
- Example adjustments to filter by start time and namespace
- SCALA-10 - And this is why we unit test. Size was returning empty for cursor based results as it wasn't pulling the right value. Fixed, calling cursor.size.
- Added an alternative object construction method for MongoDBObject with a list of pairs, rather than varargs [philwills]
- Making scaladoc for MongoURI more explicit. Note that the wiki markup for lists isn't actually implemented in scaladoc yet. [philwills]
- Refactor Collection and Cursors using Abstract types, explicit 'DBObject' version is always returned from DB, Collection etc now. Those wanting to use typed versions must code the flip around by hand. !!! BREAKING CHANGE, SEE CODE / EXAMPLES
- SCALA-10 Updated MapReduce interfaces to finish 1.8 compatibility
  - Renamed MapReduceError to MapReduceException; MapReduceError is a non exception which represents a failed job
  - Changed MapReduceResult to automatically proxy 'results' in inline result sets
- Added missing methods to GridFSDBFile necessary to access the underlying datastream
- Fixed setter/getter of option on cursor
- For several reasons changed backing trait of DBList PML from Buffer to LinearSeq
- Moved to new MapReduce functionality based on MongoDB 1.7.4+ !!! You must now specify an output mode.
  - See http://blog.evilmonkeylabs.com/2011/01/27/MongoDB-1_8-MapReduce/
- MapReduce failures shouldn't throw Error which can crash the runtime
- New MapReduceSpec updates to include tests against new MongoDB MapReduce logic

5.9 Changes in Version 2.0.2

- Fixed the MongoDBOBject 'as' operator to return the proper type, instead of Any. (philwills)

5.10 Changes in Version 2.0.1

- SCALA-16: Added a few additional validation tests against getAs and as on MongoDBObject
- SCALA-17 - Fixed syntax of $within and its nested operators, unit test passes

5.11 Version 2.0 / 2011-01-03

Notable Changes since Casbah 1.0.8.1:

- Ownership Change: Casbah is now an officially supported MongoDB Driver
  - All bugs should be reported at http://jira.mongodb.org/browse/SCALA
  - Package Change: Casbah is now `com.mongodb.casbah` (See migration guide)
  - Documentation (ScalaDocs, Migration Guide & Tutorial) is available at http://mongodb.github.com/casbah
- Casbah is now broken into several submodules - see http://mongodb.github.com/casbah/migrating.html
- Casbah releases are now published to http://scala-tools.org
- SBT Build now publishes -sources and -javadoc artifacts
- Added heavy test coverage
- ++ additivity operator on MongoDBObject for lists of tuple pairs
- Updates to Java Driver wrappings
  - Casbah now wraps Java Driver 2.4 and fully supports all options & interfaces including Replica Set and Write Concern support
  - added a WriteConcern helper object for Scala users w/ named & default args
  - added findAndModify / findAndRemove
- Stripped out support for implicit Product/Tuple conversions as they're buggy and constantly interfere with other code.
- Migrated Conversions code from core to commons, repackaging as com.mongodb.casbah.commons.conversions
  - Moved loading of ConversionHelpers from Connection creation to instantiation of Commons' Implicits (This means conversions are ALWAYS loaded now for everyone)
- Switched off of configgy to slf4j as akka did
  - Added SLF4J-JCL Bindings as a +test* dependency (so we can print logging while testing without forcing you to use an slf4j implementation yourself)
  - Moved Logger from core to commons
- Massive improvements to Query DSL:
  - Added new implementations of $in, $nin, $all and $mod with tests. $mod now accepts non-Int numerics and aof two differing types.
  - Full test coverage on DSL (and heavy coverage on other modules)
  - Migrated $each to a now functioning internal hook on $addToSet only exposed in certain circumstances
  - Various cleanups to Type constraints in Query DSL
  - Full support for all documented MongoDB query operators
  - Added new $not syntax, along with identical support for nested queries in $pull

  - Valid Date and Numeric Type boundaries introduced and used instead of Numeric (since Char doesn't actually workwith Mongo and you can't double up type bounds)
  - Added full support for geospatial query.
  - Resolved an issue where the $or wasn't being broken into individual documents as expected.
  - DSL Operators now return DBObjects rather than Product/Tuple (massive fixes to compatibility and performance result)
  - Added @see linkage to each core operator's doc page
- GridFS Changes:
  - GridFS' 'files' now returned a MongoCursor not a raw Java DBCursor
  - GridFS findOne now returns an Option[_] and detects nulls like Collection
- Added "safely" resource loaning methods on Collection & DB
  - Given an operation, uses write concern / durability on a single connection and throws an exception if anything goes wrong.
- Culled casbah-mapper. Mapper now lives as an independent project at http://github.com/maxaf/casbah-mapper
- Bumped version of scala-time to the 0.2 release
- Added DBList support via MongoDBList, following 2.8 collections
- Adjusted boundaries on getAs and expand; the view-permitting Any was causing ambiguity issues at runtime with non AnyRefs (e.g. AnyVal).
- Fixed an assumption in expand which could cause runtime failure
- Updated MongoDBObject factory & builder to explicitly return a type; some pieces were assuming at runtime that it was a MongoDBObjectBuilder$anon1 which was FUBAR

5.12 Changes in Version 1.0.7.4

- Fixed some issues w/ GridFS libraries attempting to call toMap in iteration, which isn't implemented on the Java side; added custom toString methods on the GridFS files [BWM]
- Cleaned up log spam [BWM / MA]
- Added serialization hook for MongoDBObject to help catch any nested instances [MA]
- Cleaned up some stray references to java.lang.Object, replaced with AnyRef for good Scala coding practices [BWM]

5.13 Changes in Version 1.0.7

- Updated reference to Configgy to have a Scala version attached; this was causing issues on some mixed-version users' systems.
- Corrected massive stupidity from lack of testing on my part and disabled ScalaJDeserializers - in most cases these caused runtime ClassCastExceptions. *SERIALIZERS* still in place - Deserializers were just plain a bad idea.

5.14 Changes in Version 1.0.5

- Due to oddities and ambiguities, stripped the type parameter apply[A] method from MongoDBObject. If you want a cast return, please use MongoDBObject.getAs[A]. This should minimize odd runtime failures.

- Added toplevel detection in MongoDBObject's +=/put methods to try and convert a MongoDBObject value to DBObject for you.
- Added "Product" arguments to $pushAll - this means you can pass a Tuple-style list, where previously it required an Iterable ($pushAll ("foo" -> (5, 10, 23, "spam", eggs") should now work).
- Updated to scalaj-collection 1.0 release, built against 2.8.0 final
- Added a new ScalaJ-Collection based Deserializer and Serializer layer. All base types supported by ScalaJ collection now use asJava / asScala to cleanly ser/deser where possible. This excludes Comparator/Comparable and Map types for sanity reasons. See com.novus.casbah.mongodb.conversions.scala.ScalaConversions for detail. Please report bugs if this breaks your code - it's nascent and a bit naive!
- New Committer - Max Afonov
- Removed the BitBucket Mirror; we're purely on GitHub now. Bug tracker linked from Github page.
- Created a user mailing list - http://groups.google.com/group/mongodb-casbah-users

5.15 Changes in Version 1.0.2

- Changed $in, $notin, $all to always generate an array in Any* mode
- Added default type alias import for com.mongodb.DBRef & Casbah's MongoDB class

5.16 Changes in Version 1.0.1

- Updated externals to link against 2.8.0 final - 1.0 release had some RC/Beta built externals. (scalaj-collection is still linked against Beta)
- Added an Object interface, MongoDBAddress, for static construction of DBAddress instances.
- Added type aliases in MongoTypeImports for all Casbah companion objects - please report any odd behavior this causes.
- Added MapReduceCommand to BaseImports

5.17 Version 1.0

- GridFS enhanced via Loan Pattern
- Full support for MongoDB Query operators via fluid syntax (now with lots of testing to minimize breakage)
- Added support for Scala 2.8-style Map interaction w/ DBObject. Builder pattern, +=, etc.
- Tutorial Available

CHAPTER

SIX

UPGRADE

6.1 Version 2.5.1

6.1.1 Scala 2.10

The *-Yeta-expand-keeps-star* compiler flag is no longer required.

6.2 Version 2.5.0

6.2.1 Scala 2.10

Because of how scala 2.10 handles repeated parameters you may need to build with the *-Yeta-expand-keeps-star* flag to upgrade your codebase.