Reactive Access to MongoDB from Scala

Author: Hermann Hueck

Source code and Slides available at:

https://github.com/hermannhueck/reactive-mongo-access

http://de.slideshare.net/hermannhueck/reactive-access-to-mongodb-from-scala

Who am I?

Hermann Hueck

Software Developer – Scala, Java 8, Akka, Play

https://www.xing.com/profile/Hermann_Hueck

Abstract / Part 1/3

This talk explores different Scala/Java Drivers for MongoDB and different (mostly asynchronous) strategies to access the database. The combination of Mongo drivers and programming strategies result in a great variety of possible solutions each shown by a small code sample. With the samples we can discuss the advantages and drawbacks of these stragegies.

Beginning with a discussion of …

What is asynchronous and what is reactive?

The code samples are using the following Mongo drivers for Java:

- Sychronous Java Driver
- Asychronous Java Driver (using Callbacks)
- Asychronous Java Driver (using RxJava)
- Asychronous Java Driver (implementing Reactive Streams Interface)

Abstract / Part 2/3

The code samples use the drivers in combination with the following access strategies:

- Synchronous DB access
- Async DB access with Future (from Java 5)
- Async DB access with CompletableFuture (from Java 8)
- Async DB access with Callbacks
- Reactive DB access with Callbacks and CompletableFuture (from Java 8)
- Reactive DB access with RxJava Observables
- Reactive DB access with Reactive Streams Interface + RxJava Observables
- Reactive DB access with Reactive Streams Interface + Akka Streams

Abstract / Part 3/3

All code samles are written in Java 8 using Java 8 features like Lambdas, *java.util.Optional* and *java.util.concurrent.CompletableFuture*.

A great introduction to CompletableFutures by Angelika Langner can be viewed here:

https://www.youtube.com/watch?v=Q_0_1mKTlnY

Overview 1/2

- What is the difference between asynchronous and reactive?
- The Database Collections
- The Example Use Case
- Mongo sync driver
  - Example 1: Casbah sync driver + sync data access
  - Example 2: Casbah sync driver + async data access with *Future*
  - Example 3: Casbah sync driver + async data access with *Future* and *Promise*

Overview 2/2

- Mongo async Java driver
  - Example 4: async Java driver + async data access with callbacks
  - Example 5: async Java driver + async data access with callbacks and CompletableFuture
- Mongo async Scala driver providing *MongoObservables* (similar to RX Observables)
  - Example 6: async Scala driver + async data access with *MongoObservables*
  - Example 8: async Scala driver + async data access with *rx.Observables*
  - Example 9: async Scala driver + conversion of Observable to Publisher + RxJava application code
  - Example 10: async RxJava driver + conversion of Observable to Publisher + Akka Streams application code
  - Example 11: async RxJava driver + conversion of Observable to Akka Stream Source
- Q & A

A look at synchronous code

```
val simpsons: Seq[User] = blockingIO_GetDataFromDB
simpsons.foreach(println)
```

- Synchronous: The main thread is blocked until IO completes. This is not acceptable for a server.

Asynchronous Code

```
val r: Runnable = new Runnable {
  def run(): Unit = {
    val simpsons: Seq[String] = blockingIO_GetDataFromDB
    simpsons.foreach(println)
  }
}

new Thread(r).start()
```

- Asynchronous: The main thread doesn‘t wait for the background thread. But the background thread is blocked until IO completes. This is the approach of traditional application servers, which spawn one thread per request. But this approach can be a great waste of thread resources. It doesn‘t scale.

Reactive Code with Callbacks

```
val callback = new SingleResultCallback[JList[String]]() {
  def onResult(simpsons: JList[String], t: Throwable): Unit = {
    if (t != null) {
      t.printStackTrace()
    } else {
       jListToSeq(simpsons).foreach(println)
    }
  }
}

nonblockingIOWithCallbacks_GetDataFromDB(callback)
```

- Callbacks: They do not block on I/O, but they can bring you to the “Callback Hell“.

Reactive Code with Streams

```
val obsSimpsons: MongoObservable[String] =
    nonblockingIOWithStreams_GetDataFromDB()

obsSimpsons.subscribe(new Observer[String] {
  override def onNext(simpson: String): Unit = println(simpson)
  override def onComplete(): Unit = println("----- DONE -----")
  override def onError(t: Throwable): Unit = t.printStackTrace()
})
```

- Streams (here with the RxJava-like Observable implementation of the Mongo Driver for Scala): Streams provide the most convenient reactive solution strategy.

What does “reactive“ mean?

Reactive …

- … in our context: asynchronous + non-blocking
- … generally it is much more than that. See:

http://www.reactivemanifesto.org/

The Database Collections

```
> use shop
switched to db shop
> db.users.find()
{ "_id" : "homer", "password" : "password" }
{ "_id" : "marge", "password" : "password" }
{ "_id" : "bart", "password" : "password" }
{ "_id" : "lisa", "password" : "password" }
{ "_id" : "maggie", "password" : "password" }
> db.orders.find()
{ "_id" : 1, "username" : "marge", "amount" : 71125 }
{ "_id" : 2, "username" : "marge", "amount" : 11528 }
{ "_id" : 13, "username" : "lisa", "amount" : 24440 }
{ "_id" : 14, "username" : "lisa", "amount" : 13532 }
{ "_id" : 19, "username" : "maggie", "amount" : 65818 }
```

The Example Use Case

A user can generate the eCommerce statistics only from his own orders. Therefore she first must log in to retrieve her orders and calculate the statistics from them.

The Login queries the user by her username and checks the password throwing an exception if the user with that name does not exist or if the password is incorrect.

Having logged in she queries her orders from the DB. Then the eCommerce statistics is computed from all the users orders and printed on the screen.

The Example Use Case Impl

```
def logIn(credentials: Credentials): String = {
  val optUser: Option[User] = dao.findUserByName(credentials.username)
  val user: User = checkUserLoggedIn(optUser, credentials)
  user.name
}

private def processOrdersOf(username: String): Result = {
  Result(username, dao.findOrdersByUsername(username))
}

def eCommerceStatistics(credentials: Credentials): Unit = {
  try {
    val username: String = logIn(credentials)
    val result: Result = processOrdersOf(username)
    result.display()
  }
  catch {
    case t: Throwable =>
      Console.err.println(t.toString)
  }
}
```

Synchronous MongoDB Java Driver

- “The MongoDB Driver is the updated synchronous Java driver that includes the legacy API as well as a new generic MongoCollection interface that complies with a new cross-driver CRUD specification.“ (quote from the doc)
- See:

  https://mongodb.github.io/mongo-java-driver/3.2/
- SBT:

```
libraryDependencies += "org.mongodb" % "mongodb-driver" % "3.2.2"
// or
libraryDependencies += "org.mongodb" % "mongo-java-driver" % "3.2.2"
```

  Imports:

```
import com.mongodb.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
```

Synchronous Casbah Driver for Scala

- “Casbah is the legacy Scala driver for MongoDB. It provides wrappers and extensions to the Java Driver meant to allow a more Scala-friendly interface to MongoDB. It supports serialization/deserialization of common Scala types (including collections and regex), Scala collection versions of DBObject and DBList and a fluid query DSL.“ (quote from the doc)
- See:

  https://mongodb.github.io/casbah/
- SBT:

```
libraryDependencies += "org.mongodb" % "casbah" % "3.1.1"
```

  Imports:

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

Example 1: Blocking Casbah driver + sync db access

```
object dao {

  private def _findUserByName(name: String): Option[User] = {
    usersCollection
      .findOne(MongoDBObject("_id" -> name))
      .map(User(_))
  }

  private def _findOrdersByUsername(username: String): Seq[Order] = {
    ordersCollection
      .find(MongoDBObject("username" -> username))
      .toSeq
      .map(Order(_))
  }

  def findUserByName(name: String): Option[User] = {
    _findUserByName(name)
  }

  def findOrdersByUsername(username: String): Seq[Order] = {
    _findOrdersByUsername(username)
  }
}
```

Example 1: Blocking Casbah driver + sync db access

- We use the synchronous Casbah driver.
- We use sync DB access code.
- A completely blocking solution.

Example 2: Blocking Casbah driver + async db access with *Future*

```
object dao {

  private def _findUserByName(name: String): Option[User] = {
    // blocking access to usersCollection as before
  }

  private def _findOrdersByUsername(username: String): Seq[Order] = {
    // blocking access to ordersCollection as before
  }

  def findUserByName(name: String): Future[Option[User]] = {
    Future {
      _findUserByName(name)
    }
  }

  def findOrdersByUsername(username: String): Future[Seq[Order]] = {
    Future {
      _findOrdersByUsername(username)
    }
  }
}
```

Example 2: Blocking Casbah driver + async db access with *Future*

- We still use the sync Casbah driver.
- To access and process the result of the Future you must implement the *onComplete()* handler
- The app code does not block.

Example 3: Blocking Casbah driver + async db access with *Future* and *Promise*

```
object dao {

  private def _findUserByName(name: String): Option[User] = {
    // blocking access to usersCollection as before
  }

  private def _findOrdersByUsername(username: String): Seq[Order] = {
    // blocking access to ordersCollection as before
  }

  def findUserByName(name: String): Future[Option[User]] = {
    val p = Promise[Option[User]]
    p.complete(Try {
      _findUserByName(name)
    })
    p.future
  }

  def findOrdersByUsername(username: String): Future[Seq[Order]] = {
    val p = Promise[Seq[Order]]
    p.complete(Try {
      _findOrdersByUsername(username)
    })
    p.future
  }
}
```

Example 3: Blocking Casbah driver + async db access with *Future* and *Promise*

- We still use the sync Casbah driver.
- To access and process the result of the *Future* you must implement the *onComplete()* handler
- The app code does not block.

Asynchronous MongoDB Java Driver

- “The new asynchronous API that can leverage either Netty or Java 7’s *AsynchronousSocketChannel* for fast and non-blocking IO.“ (quote from the doc)
- See:

  https://mongodb.github.io/mongo-java-driver/3.2/driver-async/
- SBT:

```
libraryDependencies += "org.mongodb" % "mongodb-driver-async" % "3.2.2"
```

- Imports:

```
import com.mongodb.async.SingleResultCallback;
import com.mongodb.async.client.MongoClient;
import com.mongodb.async.client.MongoClients;
import com.mongodb.async.client.MongoCollection;
import com.mongodb.async.client.MongoDatabase;
```

Example 4: Async Mongo Java driver + async db access with Callbacks

```
object dao {

  private def _findOrdersByUsername(username: String,
                                    callback: SingleResultCallback[JList[Order]]): Unit = {
    ordersCollection
      .find(Filters.eq("username", username))
      .map(scalaMapper2MongoMapper { doc => Order(doc) })
      .into(new JArrayList[Order], callback)
  }

  def findOrdersByUsername(username: String,
                           callback: SingleResultCallback[JList[Order]]): Unit =
    _findOrdersByUsername(username, callback)
}
```

```
    . . .
    val callback = new SingleResultCallback[JList[Order]]() {
      override def onResult(orders: JList[Order], t: Throwable): Unit = {
        if (t != null) {
           // handle exception
        } else {
          // process orders
        }
      }
    }

    findOrdersByUsername("homer", callback);
```

Example 4: Async Mongo Java driver + async db access with Callbacks

- We use the asynchronous callback based Mongo Java driver.
- The callback function must be passed into the invocation of the Mongo driver.
- There is no blocking code neither in the driver nor in the application code.
- The solution is completely reactive.
- Callback based programming may easily lead to unmanagable code … the so called “Callback Hell“.

Example 5: Async Mongo Java driver + async db access with *Future* and *Promise*

```
object dao {

  private def _findOrdersByUsername(username: String,
                                    callback: SingleResultCallback[JList[Order]]): Unit = {
    ordersCollection
      .find(Filters.eq("username", username))
      .map(scalaMapper2MongoMapper { doc => Order(doc) })
      .into(new JArrayList[Order], callback)
  }

  def findOrdersByUsername(username: String): Future[Seq[Order]] = {

    val promise = Promise[Seq[Order]]

    _findOrdersByUsername(username): new SingleResultCallback[JList[Order]]() {

      override def onResult(orders: JList[Order], t: Throwable): Unit = {
        if (t == null) {
          promise.success(jListToSeq(orders))
        } else {
          promise.failure(t)
        }
      }
    })

    promise.future
  }
}
```

Example 5: Async Mongo Java driver + async db access with *Future* and *Promise*

- We use the asynchronous callback based Mongo driver.
- The callback function completes the *Promise*. Callback code remains encapsulated inside the DAO and does not pollute application logic. → Thus “Callback Hell“ is avoided.
- There is no blocking code neither in the driver nor in the application code.
- The solution is completely reactive.

MongoDB Scala Driver (async)

“The Scala driver is free of dependencies on any third-party frameworks for asynchronous programming. To achieve this the Scala API makes use of an custom implementation of the Observer pattern which comprises three traits:

- Observable
- Observer
- Subscription

… The MongoDB Scala Driver is built upon the callback-driven MongoDB async driver. The API mirrors the async driver API and any methods that cause network IO return an instance of the Observable[T] where T is the type of response for the operation“. (quote from the doc)

MongoDB Scala Driver (async)

See:

https://mongodb.github.io/mongo-scala-driver/1.1

- SBT:

```
libraryDependencies += "org.mongodb.scala" % "mongo-scala-driver" % "1.1.1"
```

- Imports:

```
import com.mongodb.scala.MongoClient;
import com.mongodb.scala.MongoCollection;
import com.mongodb.scala.MongoDatabase;
```

Example 6: Mongo Scala driver + async db access with *MongoObservables*

```
object dao {

  private def _findOrdersByUsername(
                      username: String): MongoObservable[Seq[Order]] = {
    ordersCollection
      .find(Filters.eq("username", username))
      .map(doc => Order(doc))
      .collect()
  }

  def findOrdersByUsername(username: String): MongoObservable[Seq[Order]] = {
    _findOrdersByUsername(username)
  }
}
```

```
  . . .

  val obsOrders: MongoObservable[Seq[Order]] = dao.findOrdersByUsername("lisa")

  obsOrders.subscribe(new Observer[Seq[Order]] {
    override def onNext(order: Seq[Order]): Unit = println(order)
    override def onComplete(): Unit = println("----- DONE -----")
    override def onError(t: Throwable): Unit = t.printStackTrace()
  })
```

Example 6: Mongo Scala driver + async db access with *MongoObservables*

- We use the async Mongo Scala driver.
- This driver internally uses the Mongo async Java driver.
- The DAO function returns a *MongoObservable* which is a convenient source of a stream processing pipeline.
- There is no blocking code neither in the driver nor in the application code.
- The solution is completely reactive.
- Elegant streaming solution.
- The functionality is a bit reduced compared to the rich set of operators in RxJava or Akka Streams.

Example 7: Mongo Scala driver + async db access with *MongoObservable* + *rx.Observable*

```
object dao {

  private def _findOrdersByUsername(
                          username: String): MongoObservable[Seq[Order]] = {
    ordersCollection
      .find(Filters.eq("username", username))
      .map(doc => Order(doc))
      .collect()
  }

  def findOrdersByUsername(username: String): rx.Observable[Seq[Order]] = {
    RxScalaConversions.observableToRxObservable(_findOrdersByUsername(username))
  }
}
```

```
  . . .

  val obsOrders: rx.Observable[Seq[Order]] = dao.findOrdersByUsername("lisa")

  obsOrders.subscribe(new Observer[Seq[Order]] {
    override def onNext(order: Seq[Order]): Unit = println(order)
    override def onCompleted(): Unit = println("----- DONE -----")
    override def onError(t: Throwable): Unit = t.printStackTrace()
  })
```

Example 7: Mongo Scala driver + async db access with *MongoObservable* + *rx.Observable*

- We use the async Mongo Scala driver.
- This driver internally uses the Mongo async driver.
- The DAO function returns an *rx.Observable* which is a convenient source of a stream processing pipeline.
- There is no blocking code neither in the driver nor in the application code.
- The solution is completely reactive.
- Elegant streaming solution.

Example 8: Mongo Scala driver + async db access with *MongoObservable* + Reactive Streams Interface + *rx.Observable*

```
object dao {

  private def _findOrdersByUsername(
                          username: String): MongoObservable[Seq[Order]] = {
    ordersCollection
      .find(Filters.eq("username", username))
      .map(doc => Order(doc))
      .collect()
  }

  def findOrdersByUsername(username: String): Publisher[Seq[Order]] = {
    RxStreamsConversions.observableToPublisher(_findOrdersByUsername(username))
  }
}
```

```
  . . .

  val pubOrders: Publisher[Seq[Order]] = dao.findOrdersByUsername("lisa")
  val obsOrders: rx.Observable[Seq[Order]] = publisherToObservable(pubOrders)

  obsOrders.subscribe(new Observer[Seq[Order]] {
    override def onNext(order: Seq[Order]): Unit = println(order)
    override def onCompleted(): Unit = println("----- DONE -----")
    override def onError(t: Throwable): Unit = t.printStackTrace()
  })
```

Example 8: Mongo Scala driver + async db access with *MongoObservable* + Reactive Streams Interface + *rx.Observable*

- We use the Mongo Scala driver.
- The DAO function returns an *org.reactivestreams.Publisher* which makes it compatible with any Reactive Streams implementation.
  See: *http://www.reactive-streams.org/announce-1.0.0*
- You cannot do much with a *Publisher* except convert it to the native type of a compatible implementation, e.g. an RxScala *Observable*.
- Having converted the *Publisher* to an *Observable* you can use all the powerful operators of RxScala Observables.
- There is no blocking code neither in the driver nor in the application code.
- The solution is completely reactive.

Example 9: Mongo Scala driver + async db access with *MongoObservable* + Reactive Streams Interface + Akka Streams

```
object dao {

  private def _findOrdersByUsername(
                          username: String): MongoObservable[Seq[Order]] = {
    ordersCollection
      .find(Filters.eq("username", username))
      .map(doc => Order(doc))
      .collect()
  }

  def findOrdersByUsername(username: String): Publisher[Seq[Order]] = {
    RxStreamsConversions.observableToPublisher(_findOrdersByUsername(username))
  }
}
```

```
  . . .

  val pubOrders: Publisher[Seq[Order]] = dao.findOrdersByUsername("lisa")
  val srcOrders: Source[Seq[Order], NotUsed] = Source.fromPublisher(pubOrders)
  val f: Future[Done] = srcOrders).runForeach(order => ...)

  f.onComplete(tryy => tryy match {
    case Success(_) =>  // ...
    case Failure(t) =>  // ...
  })
```

Example 9: Mongo Scala driver + async db access with *MongoObservable* + Reactive Streams Interface + Akka Streams

- We use the Mongo Scala driver.
- The DAO function returns an *org.reactivestreams.Publisher* which makes it compatible with any Reactive Streams implementation.
  See: *http://www.reactive-streams.org/announce-1.0.0*
- You cannot do much with a *Publisher* except convert it to the native type of a compatible implementation, e.g. an Akka Streams *Source*.
- Having converted the *Publisher* to a *Source* you can use all the powerful operators available in Akka Streams.
- There is no blocking code neither in the driver nor in the application code.
- The solution is completely reactive.

Example 10: Mongo Scala driver + async db access with *MongoObservable* + Akka Streams

```
object dao {

  private def _findOrdersByUsername(
                          username: String): MongoObservable[Seq[Order]] = {
    ordersCollection
      .find(Filters.eq("username", username))
      .map(doc => Order(doc))
      .collect()
  }

  def findOrdersByUsername(username: String): Source[Seq[Order], NotUsed] = {
    val pubOrders: Publisher[Seq[Order]] =
      RxStreamsConversions.observableToPublisher(_findOrdersByUsername(username))
    Source.fromPublisher(pubOrders)
  }
}
```

```
  . . .

  val srcOrders: Source[Seq[Order], NotUsed] = dao.findOrdersByUsername("lisa")
  val f: Future[Done] = srcOrders).runForeach(order => ...)

  f.onComplete(tryy => tryy match {
    case Success(_) =>  // ...
    case Failure(t) =>  // ...
  })
```

Example 10: Mongo Scala driver + async db access with *MongoObservable* + Akka Streams

- Technically identical to example no. 9
- But the DAO provides an Akka Streams interface to the app code.

Example 11: Mongo Scala driver + async db access with *MongoObservable* + Future

```
object dao {

  private def _findOrdersByUsername(
                          username: String): MongoObservable[Seq[Order]] = {
    ordersCollection
      .find(Filters.eq("username", username))
      .map(doc => Order(doc))
      .collect()
  }

  def findOrdersByUsername(username: String): Source[Seq[Order], NotUsed] = {
    val pubOrders: Publisher[Seq[Order]] =
      RxStreamsConversions.observableToPublisher(_findOrdersByUsername(username))
    Source.fromPublisher(pubOrders)
  }
}


   . . .

   val srcOrders: Source[Seq[Order], NotUsed] = dao.findOrdersByUsername("lisa")
   val f: Future[Done] = srcOrders).runForeach(order => ...)

   f.onComplete(tryy => tryy match {
     case Success(_) =>  // ...
     case Failure(t) =>  // ...
   })
```

Example 11: Mongo Scala driver + async db access with *MongoObservable* + Future

- We use the async Mongo Scala driver.
- This driver internally uses the Mongo async Java driver.
- The DAO function returns a *Future*.
- There is no blocking code neither in the driver nor in the application code.
- The solution is completely reactive.

RxScala Characteristics

- Powerful set of stream operators
- Simple API
- Simple thread pool configuration with *subscribeOn()* and *observeOn()*
- Rx ports available for many languages:
  RxJava, RxJS, Rx.NET, RxScala, RxClojure, RxSwift, RxCpp, Rx.rb, RxPy, RxGroovy, RxJRuby, RxKotlin, RxPHP
- Rx ports available for specific platforms or frameworks:
  RxNetty, RxAndroid, RxCocoa

Akka Streams Characteristics

- Powerful set of stream operators
- Very flexible operators to split or join streams
- API a bit more complex than RxScala API
- Powerful actor system working under the hood
- Available for Scala and Java 8

Resources – Talk / Author

- Source Code and Slides on Github:
  https://github.com/hermannhueck/reactive-mongo-access
- Slides on SlideShare:
  http://de.slideshare.net/hermannhueck/reactive-access-to-mongodb-from-scala
- Authors XING Profile:
  https://www.xing.com/profile/Hermann_Hueck

Resources – Java Drivers for MongoDB

- Overview of Java drivers for MongoDB: https://docs.mongodb.com/ecosystem/drivers/java/
- Latest sync driver: https://mongodb.github.io/mongo-java-driver/3.2/
- Latest async driver: https://mongodb.github.io/mongo-java-driver/3.2/driver-async/
- How the RxJava driver and the Reactive Streams driver wrap the async driver: https://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/observables/
- Latest RxJava driver: https://mongodb.github.io/mongo-java-driver-rx/1.2/
- Latest Reactive Streams driver: https://mongodb.github.io/mongo-java-driver-reactivestreams/1.2/
- Latest Morphia Driver: https://mongodb.github.io/morphia/1.1/

Resources – RxJava

- ReactiveX: http://reactivex.io/intro.html
- RxJava Wiki: https://github.com/ReactiveX/RxJava/wiki
- Nice Intro to RxJava by Dan Lev: http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/
- RxJava- Understanding observeOn() and subscribeOn() by Thomas Nield: http://tomstechnicalblog.blogspot.de/2016/02/rxjava-understanding-observeon-and.html

Resources – Reactive Streams and Akka Streams

- Reactive Streams:
  http://www.reactive-streams.org/
- Current implementations of Reactive Streams:
  http://www.reactive-streams.org/announce-1.0.0
- RxJava to Reactive Streams conversion library:
  https://github.com/ReactiveX/RxJavaReactiveStreams
- Akka (2.4.6) Documentation for Java 8:
  http://doc.akka.io/docs/akka/2.4.6/java.html
- The Reactive Manifesto:
  http://www.reactivemanifesto.org/

Thanks for your attention!

Q & A