

Reactive Access to MongoDB from Java 8

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-java-8>

Who am I?

Hermann Hueck

Software Developer – Scala, Java 8, Akka, Play

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

Abstract / Part 1

Beginning with a discussion of ...

What is asynchronous and what is reactive?

I will show small code samples using different Mongo drivers for Java:

- Synchronous Java Driver
- Asynchronous Java Driver (using Callbacks)
- Asynchronous Java Driver (using RxJava)
- Asynchronous Java Driver (implementing Reactive Streams Interface)

Abstract / Part 2

I will use these drivers in combination with different asynchronous access strategies:

- Future (from Java 5)
- CompletableFuture / CompletionStage (from Java 8)
- Callbacks
- RxJava Observables
- Reactive Streams Interface + RxJava Observables
- Reactive Streams Interface + Akka Streams

Abstract / Part 3

The combination of these drivers and access strategies result in a great variety of possible solutions. Thus with the examples we can discuss their advantages and drawbacks.

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

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

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

Overview 1/2

- Three preliminary notes on my programming style
- What is the difference between asynchronous and reactive?
- The Database Collections
- The Example Use Case
- Mongo sync driver
 - Example 1: sync driver + sync data access
 - Example 2: sync driver + async (?) data access with (Java 5) Future
 - Example 3: sync driver + async data access with (Java 8) CompletableFuture
 - Example 4: sync driver + async data access using the CompletableFuture's result provider interface

Overview 2/2

- Mongo async driver
 - Example 5: async driver + async data access with callbacks
 - Example 6: async driver + async data access with callbacks and `CompletableFuture`
- Mongo RxJava and Reactive Streams drivers
 - Example 7: async RxJava driver + async data access with Observables
 - Example 8: async Reactive Streams driver + async data access with Observables
 - Example 9: async RxJava 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

Note no. 1/3

final is the default.

final makes your variables „invariable“, i.e. immutable. What is immutable is also thread-safe.

Note no. 2/3

Make your Domain Objects
immutable (with **final**).

If all properties are **final**, you can make them public. You don't need getters and setters.

Note no. 2/3

```
public class User {  
    public final String name;  
    public final String password;  
  
    public User(final String name, final String password) {  
        this.name = name;  
        this.password = password;  
    }  
}  
  
public class Order {  
    public final int id;  
    public final String username;  
    public final int amount;  
  
    public Order(final int id, final String username, final int amount) {  
        this.id = id;  
        this.username = username;  
        this.amount = amount;  
    }  
}
```

Note no. 3/3

Use *java.util.Optional*
instead of **null**.

An *Optional* is just a typed container which contains either nothing or a something of a certain type.

Optional can shield you from many NPEs.

Note no. 3/3

```
Optional<User> findUserByName(final String name) {  
    final Document doc = usersCollection  
        .find(eq("_id", name)).first();  
    // either: return doc == null ?  
    //           Optional.empty() : Optional.of(new User(doc)); ... or better  
    return Optional.ofNullable(doc).map(User::new);  
}
```

. . .

```
if (!optUser.isPresent()) {      // replaces:    if (user != null) {  
    // user found  
    User user = optUser.get();  
    . . .  
} else {  
    // user not found  
    . . .  
}
```

A look at synchronous code

```
List<String> simpsons = blockingIOGetDataFromDB();  
simpsons.forEach(simpson -> System.out.println(simpson));
```

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

Asynchronous Code

```
Runnable r = () -> {  
    List<String> simpsons = blockingIOGetDataFromDB();  
    simpsons.forEach(simpson -> System.out.println(simpson));  
};  
  
new Thread(r).start();
```

- Asynchronous: The main thread does not 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

```
SingleResultCallback<List<String>> callback = (simpsons, t) -> {  
    if (t != null) {  
        t.printStackTrace();  
    } else {  
        simpsons.forEach(simpson -> System.out.println(simpson));  
    }  
};  
  
nonblockingIOWithCallbacks_GetDataFromDB(callback);
```

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

Reactive Code with Streams

```
Observable<String> obsSimpsons = nonblockingIOWithStreams_GetDataFromDB();
```

```
obsSimpsons
    .filter(. . .)
    .map(. . .)
    .flatMap(. . .)
    .subscribe(new Observer<String>() {
        public void onNext(String simpson) {
            System.out.println(simpson);
        }
        public void onCompleted() {
            System.out.println("----- DONE -----");
        }
        public void onError(Throwable t) {
            t.printStackTrace();
        }
    });
```

- Streams (here with RxJava): provide the most convenient reactive solution strategy

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 statistice only from his own orders. Therefore she must first log in.

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

```
private String logIn(final Credentials credentials) {  
    final Optional<User> optUser = dao.findUserByName(credentials.username);  
    final User user = checkUserLoggedIn(optUser, credentials);  
    return user.name;  
}  
  
private Result processOrdersOf(final String username) {  
    final List<Order> orders = dao.findOrdersByUsername(username);  
    return new Result(username, orders);  
}  
  
private void eCommerceStatistics(final Credentials credentials) {  
    try {  
        final String username = logIn(credentials);  
        final Result result = processOrdersOf(username);  
        result.display();  
    } catch (Exception e) {  
        System.err.println(e.toString());  
    }  
}
```

Synchronous MongoDB Driver

- “The MongoDB Driver is the updated synchronous Java driver that includes the legacy API as well as a new generic MongoClient 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;
```

Example 1: Blocking Mongo driver + sync db access

```
class DAO {  
    Optional<User> findUserByName(final String name) {  
        final Document doc = usersCollection  
            .find(eq("_id", name))  
            .first();  
        return Optional.ofNullable(doc).map(User::new);  
    }  
  
    List<Order> findOrdersByUsername(final String username) {  
        final List<Document> docs = ordersCollection  
            .find(eq("username", username))  
            .into(new ArrayList<>());  
        return docs.stream()  
            .map(doc -> new Order(doc))  
            .collect(toList());  
    }  
}
```

Example 1: Blocking Mongo driver + sync db access

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

Example 2: Blocking Mongo driver + async db access with Future (Java 5)

```
class DAO {  
  
    Future<Optional<User>> findUserByName(final String name) {  
        return executor.submit(() -> {  
            final Document doc = usersCollection  
                .find(eq("_id", name))  
                .first();  
            return Optional.ofNullable(doc).map(User::new);  
        });  
    }  
  
    Future<List<Order>> findOrdersByUsername(final String username) {  
        return executor.submit(() -> {  
            final List<Document> docs = ordersCollection  
                .find(eq("username", username))  
                .into(new ArrayList<>());  
            return docs.stream()  
                .map(doc -> new Order(doc))  
                .collect(toList());  
        });  
    }  
}
```

Example 2: Blocking Mongo driver + async db access with Future (Java 5)

- We still use the sync Mongo driver.
- To access and process the result of the Future you must use the Future's *get()* or *get(timeout)* method.
- *get()* blocks the invoker.
- *get(timeout)* polls the Future but wastes resources. (busy wait)

Example 3: Blocking Mongo driver + async db access with CompletableFuture (Java 8)

```
class DAO {  
  
    CompletionStage<Optional<User>> findUserByName(final String name) {  
        return CompletableFuture.supplyAsync(() -> {  
            Document doc = usersCollection  
                .find(eq("_id", name))  
                .first();  
            return Optional.ofNullable(doc).map(User::new);  
        });  
    }  
  
    CompletionStage<List<Order>> findOrdersByUsername(final String username) {  
        return CompletableFuture.supplyAsync(() -> {  
            List<Document> docs = ordersCollection  
                .find(eq("username", username))  
                .into(new ArrayList<>());  
            return docs.stream()  
                .map(doc -> new Order(doc))  
                .collect(toList());  
        });  
    }  
}
```

Example 3: Blocking Mongo driver + async db access with `CompletableFuture` (Java 8)

- We still use the sync Mongo driver.
- To access and process the result of the *CompletableFuture* you can specify the processing action before the future completes using *thenApply()*, *thenCompose()* etc.
- No callbacks are necessary.
- The *CompletableFuture* is reactive but the Mongo driver is not.

Example 4: Blocking Mongo driver + async db access with `CompletableFuture.complete()`

```
class DAO {  
    CompletionStage<List<Order>> findOrdersByUsername(final String username) {  
        final CompletableFuture<List<Order>> future =  
            new CompletableFuture<>();  
        final Runnable runnable = () -> {  
            try {  
                List<Document> docs = ordersCollection.  
                    find(eq("username", username))  
                        .into(new ArrayList<>());  
                List<Order> orders = docs.stream()  
                    .map(doc -> new Order(doc))  
                    .collect(toList());  
                future.complete(orders);  
            } catch (Exception e) {  
                future.completeExceptionally(e);  
            }  
        };  
        executor.execute(runnable);  
        return future;  
    }  
}
```

Example 4: Blocking Mongo driver + async db access with `CompletableFuture.complete()`

- We still use the sync Mongo driver.
- To access and process the result of the *CompletableFuture* you can specify the processing action before the future completes using *thenApply()*, *thenCompose()* etc.
- No callbacks are necessary.
- The *CompletableFuture* is reactive but the Mongo driver is not.

Asynchronous MongoDB 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 5: Async Mongo driver + async db access with Callbacks

```
class DAO {  
  
    void findOrdersByUsername(final String username,  
                             final SingleResultCallback<List<Order>> callback) {  
        ordersCollection  
            .find(eq("username", username))  
            .map(doc -> new Order(doc))  
            .into(new ArrayList<>(), callback);  
    }  
}  
  
    final SingleResultCallback<List<Order>> callback = (orders, t) -> {  
        if (t != null) {  
            // handle exception  
        } else {  
            // process orders  
        }  
    };  
  
    findOrdersByUsername("username", callback);  
}
```

Example 5: Async Mongo driver + async db access with Callbacks

- We use the asynchronous callback based Mongo 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 6: Async Mongo driver + async db access with `CompletableFuture.complete()`

```
class DAO {  
    CompletionStage<List<Order>> findOrdersByUsername(final String username) {  
        final CompletableFuture<List<Order>> future =  
            new CompletableFuture<>();  
  
        final SingleResultCallback<List<Order>> callback = (orders, t) -> {  
            if (t == null) {  
                future.complete(orders);  
            } else {  
                future.completeExceptionally(t);  
            }  
        };  
  
        ordersCollection  
            .find(eq("username", username))  
            .map(doc -> new Order(doc))  
            .into(new ArrayList<>(), callback);  
  
        return future;  
    }  
}
```


Example 6: Async Mongo driver + async db access with `CompletableFuture.complete()`

- We use the asynchronous callback based Mongo driver.
- The callback function completes the *CompletableFuture*. 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 Driver for RxJava

- “An implementation of the MongoDB Driver providing support for ReactiveX (Reactive Extensions) by using the RxJava library”. (quote from the doc)
- See:
<https://mongodb.github.io/mongo-java-driver-rx/1.2/>
<https://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/observables/>

- SBT:

```
libraryDependencies += "org.mongodb" % "mongodb-driver-rx" % "1.2.0"
```

- Imports:

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

Example 7: Mongo RxJava driver + async db access with Observables

```
class DAO {  
  
    Observable<List<Order>> findOrdersByUsername(final String username) {  
  
        return ordersCollection  
            .find(eq("username", username))  
            .toObservable()  
            .map(doc -> new Order(doc))  
            .toList();  
    }  
}  
  
...  
  
Observable<Result> obsResult =  
    dao.findOrdersByUsername(username)  
        .map(orders -> new Result(username, orders));  
  
obsResult.subscribe(new Observer() { . . . });
```

Example 7: Mongo RxJava driver + async db access with Observables

- We use the Mongo RxJava driver.
- This driver internally uses the Mongo async driver.
- The DAO function returns a RxJava *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.

MongoDB Reactive Streams Driver

- The Reactive Streams implementation for asynchronous stream processing with non-blocking back pressure. (quote from the doc)

- See:

<https://mongodb.github.io/mongo-java-driver-reactivestreams/1.2/>

<http://www.reactive-streams.org/announce-1.0.0>

- SBT:

```
libraryDependencies += "org.mongodb" % "mongodb-driver-reactivestreams" % "1.2.0"
```

- Imports:

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

Example 8: Mongo Reactive Streams driver + async db access with Observables

```
class DAO {  
    Publisher<Document> findOrdersByUsername(final String username) {  
        return ordersCollection  
            .find(eq("username", username));  
    }  
}  
  
. . .  
  
Observable<Result> obsResult =  
    rx.RxReactiveStreams.toObservable(dao.findOrdersByUsername(username))  
        .map(doc -> new Order(doc))  
        .toList()  
        .map(orders -> new Result(username, orders));  
  
obsResult.subscribe(new Observer() { . . . });
```

Example 8: Mongo Reactive Streams driver + async db access with Observables

- We use the Mongo Reactive Streams driver.
- This driver internally uses the Mongo Rx driver.
- The DAO function returns an *org.reactivestreams.Publisher<org.bson.Document>* 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 RxJava *Observable*.
- Having converted the *Publisher* to an *Observable* you can use all the powerful operators of RxJava Observables.
- There is no blocking code neither in the driver nor in the application code.
- The solution is completely reactive.
- Streaming solution which doesn't allow preprocessing of the db query result inside the DAO.

Example 9: Mongo RxJava driver + conversion of Observable to Publisher + RxJava application code

```
class DAO {  
  
    Publisher<List<Order>> findOrdersByUsername(final String username) {  
        Observable<List<Order>> observable = ordersCollection  
            .find(eq("username", username))  
            .toObservable()  
            .map(doc -> new Order(doc))  
            .toList();  
        return rx.RxReactiveStreams.toPublisher(observable);  
    }  
}  
  
...  
  
Observable<Result> obsResult =  
    rx.RxReactiveStreams.toObservable(dao.findOrdersByUsername(username))  
        .map(orders -> new Result(username, orders));  
  
obsResult.subscribe(new Observer() { . . . });
```


Example 9: Mongo RxJava driver + conversion of Observable to Publisher + RxJava application code

- We use the Mongo RxJava driver.
- We preprocess the data inside the DAO with Observables. Then we convert the *Observable* to a *Publisher* using the RxJava → Reactive Streams Converter library.
- Now the DAO function returns an *org.reactivestreams.Publisher<List<Order>>* which makes it compatible to any Reactive Streams implementation.
See: <http://www.reactive-streams.org/announce-1.0.0>
- In the application code we convert the *Publisher* back to an *Observable* in order to process the result of the DAO.
- There is no blocking code neither in the driver nor in the application code.
- The solution is completely reactive.
- Elegant streaming solution providing a “meaningful” Reactive Streams interface of the DAO.

Example 10: Mongo RxJava driver + conversion of Observable to Publisher + Akka Streams application code

```
class DAO {  
  
    Publisher<List<Order>> findOrdersByUsername(final String username) {  
        Observable<List<Order>> observable = ordersCollection  
            .find(eq("username", username))  
            .toObservable()  
            .map(doc -> new Order(doc))  
            .toList();  
        return rx.RxReactiveStreams.toPublisher(observable);  
    }  
}  
  
. . .  
  
Source<Result, NotUsed> srcResult =  
    Source.fromPublisher(dao.findOrdersByUsername(username))  
        .map(orders -> new Result(username, orders));  
  
srcResult.runForeach(result -> { . . . }, materializer);
```

Example 10: Mongo RxJava driver + conversion of Observable to Publisher + Akka Streams application code

- We use the Mongo RxJava driver (as before).
- We preprocess the data inside the DAO with Observables. Then we convert the *Observable* to a *Publisher* using the RxJava → Reactive Streams Converter library (as before).
- Now the DAO function returns an *org.reactivestreams.Publisher<List<Order>>* which makes it compatible to any other Reactive Streams implementation (as before).
See: <http://www.reactive-streams.org/announce-1.0.0>
- In the application code we convert the *Publisher* to an Akka Stream *Source* in order to process the result of the DAO.
- There is no blocking code neither in the driver nor in the application code.
- The solution is completely reactive.
- Elegant streaming solution providing a “meaningful” Reactive Streams interface of the DAO.

Example 11: Mongo RxJava driver + conversion of Observable to Akka Streams Source

```
class DAO {  
  
    Source<List<Order>, NotUsed> findOrdersByUsername(final String username) {  
        Observable<List<Order>> observable = ordersCollection  
            .find(eq("username", username))  
            .toObservable()  
            .map(doc -> new Order(doc))  
            .toList();  
        Publisher<List<Order>> pub =  
            rx.RxReactiveStreams.toPublisher(observable);  
        return Source.fromPublisher(pub);  
    }  
}  
  
...  
  
Source<Result, NotUsed> srcResult =  
    dao.findOrdersByUsername(username)  
        .map(orders -> new Result(username, orders));  
  
srcResult.runForeach(result -> { . . . }, materializer);
```

Example 11: Mongo RxJava driver + conversion of Observable to Akka Stream Source

- Technically identical to example no. 10
- But the DAO provides an Akka Streams interface to the app code.

RxJava ↔ Akka Streams

- *///////// TBD //////////*

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-java-8>
- 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 – CompletableFuture

- Angelika Langner's talk on Java 8 concurrency and CompletableFuture:
https://www.youtube.com/watch?v=Q_0_1mKTlnY
- CompletableFuture JavaDoc:
<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html>

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>

Thanks for your attention!

Q & A