Reactive Access to MongoDB from Java 8

Author: Hermann Hueck

Last Update: September 22nd, 2016

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 1/3

This talk explores different 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

- Three preliminary notes on 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 immutable.

- Immutable base types are thread-safe.
- Immutable object references make the references thread-safe, but not the objects themselves. Therefore ...

Note no. 2a/3

Make the properties of your Domain Objects immutable (with **final**).

If all properties are **final**, you can make them public. You no longer need getters and setters.

Better solution: https://immutables.github.io/

Note no. 2b/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. 3a/3

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

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

Optional can shield you from many NPEs.

For more on avoiding null, see:
https://github.com/google/guava/wiki/UsingAndAvoidingNullExplained

Note no. 3b/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 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

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

nonblockingIOWithCallbacks_GetDataFromDB(callback);
```

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

What does “reactive“ mean?

Reactive systems are …

- Responsive
- Resilient
- Elastic
- Message Driven

See: http://www.reactivemanifesto.org/

In the context of our application:

reactive = asynchronous + non-blocking

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 own orders from the DB. Then the eCommerce statistics is computed from all orders of this user 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 MongoCollection interface that complies with a new cross-driver CRUD specification.“ (quote from the doc)
- See:

  https://mongodb.github.io/mongo-java-driver/3.3/
- SBT:

```
libraryDependencies += "org.mongodb" % "mongodb-driver" % "3.3.0"
// or
libraryDependencies += "org.mongodb" % "mongo-java-driver" % "3.3.0"
```

  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 {

    private Optional<User> _findUserByName(final String name) {
        final Document doc = usersCollection
                .find(eq("_id", name))
                .first();
        return Optional.ofNullable(doc).map(User::new);
    }

    private 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());
    }

    Optional<User> findUserByName(final String name) {
        return _findUserByName(name);
    }

    List<Order> findOrdersByUsername(final String username) {
        return _findOrdersByUsername(username);
    }
}
```

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 {

    private Optional<User> _findUserByName(final String name) {
        // blocking access to usersCollection as before
    }

    private List<Order> _findOrdersByUsername(final String username) {
        // blocking access to ordersCollection as before
    }

    Future<Optional<User>> findUserByName(final String name) {
        Callable<Optional<User>> callable = () -> _findUserByName(name);
        return executor.submit(callable);
    }

    Future<List<Order>> findOrdersByUsername(final String username) {
        Callable<List<Order>> callable = () -> _findOrdersByUsername(username);
        return executor.submit(callable);
    }
}
```

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 until the result is available.
- *get(timeout)* polls the *Future* but wastes resources. (busy wait)

Future (Java 5) with *CompletionService*

- To mitigate the pain of Java 5 *Future*s you can use a *java.util.concurrrent.CompletionService*.
- The *CompletionService* manages a Queue of *Future*s and allows to wait for the *Future*s as they complete.
- With the take() method you **block only one Thread** to wait for many *Future*s to complete.
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionService.html

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

```
class DAO {

    private Optional<User> _findUserByName(final String name) {
        // blocking access to usersCollection as before
    }

    private List<Order> _findOrdersByUsername(final String username) {
        // blocking access to ordersCollection as before
    }

    CompletionStage<Optional<User>> findUserByName(final String name) {
        Supplier<Optional<User>> supplier = () -> _findUserByName(name);
        return CompletableFuture.supplyAsync(supplier);
    }

    CompletionStage<List<Order>> findOrdersByUsername(final String username) {
        Supplier<List<Order>> supplier = () -> _findOrdersByUsername(username);
        return CompletableFuture.supplyAsync(supplier);
    }
}
```

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 after 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 {

    private List<Order> _findOrdersByUsername(final String username) {
        // blocking access to ordersCollection as before
    }

    CompletionStage<List<Order>> findOrdersByUsername(final String username) {

        final CompletableFuture<List<Order>> future = new CompletableFuture<>();

        final Runnable runnable = () -> {
            try {
                future.complete(_findOrdersByUsername(username));
            } 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 after 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/3.3/driver-async/
- SBT:

```
libraryDependencies += "org.mongodb" % "mongodb-driver-async" % "3.3.0"
```

- 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 {

    private void _findOrdersByUsername(final String username,
                           final SingleResultCallback<List<Order>> callback) {
        ordersCollection
                .find(eq("username", username))
                .map(doc -> new Order(doc))
                .into(new ArrayList<>(), callback);
    }

    void findOrdersByUsername(final String username,
                           final SingleResultCallback<List<Order>> callback) {
        _findOrdersByUsername(username, 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 {

    private void _findOrdersByUsername(final String username,
                          final SingleResultCallback<List<Order>> callback) {
        // non-blocking access to ordersCollection as before
    }

    CompletionStage<List<Order>> findOrdersByUsername(final String username) {

        final CompletableFuture<List<Order>> future =
                                                new CompletableFuture<>();

        findOrdersByUsername(username, (orders, t) -> {
            if (t == null) {
                future.complete(orders);
            } else {
                future.completeExceptionally(t);
            }
        });

        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/3.3/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 {

    private Observable<Order> _findOrdersByUsername(final String username) {
        return ordersCollection
                .find(eq("username", username))
                .toObservable()
                .map(doc -> new Order(doc));
    }

    Observable<Order> findOrdersByUsername(final String username) {
        _findOrdersByUsername(username);
    }
}

   . . .


    Observable<Result> obsResult =
        dao.findOrdersByUsername(username)
            .map(orders -> . . .);
            . . .
            .map(... -> new Result(username, ...));

    obsResult.subscribe(new Observer() { . . . });
```

Example 7: Mongo RxJava driver + async db access with *Observable*s

- 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 {

    private Observable<Order> _findOrdersByUsername(final String username) {
        // RX access to ordersCollection as before
    }

    Publisher<List<Order>> findOrdersByUsername(final String username) {
        return rx.RxReactiveStreams.toPublisher(
                        _findOrdersByUsername(username).toList());
    }
}

   . . .


    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 {

    private Observable<Order> _findOrdersByUsername(final String username) {
        // RX access to ordersCollection as before
    }

    Publisher<List<Order>> findOrdersByUsername(final String username) {
        return rx.RxReactiveStreams.toPublisher(
                        _findOrdersByUsername(username).toList());
    }
}

   . . .


    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 {

    private Observable<Order> _findOrdersByUsername(final String username) {
        // RX access to ordersCollection as before
    }

    Source<List<Order>, NotUsed> findOrdersByUsername(final String username) {
        Observable<List<Order>> observable =
                        _findOrdersByUsername(username).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 Characteristics

- Powerful set of stream operators
- Simple API
- Simple thread pool configuration with *subscribeOn()* and *observeOn()*
- RxJava available for Java 6
- 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 RxJava 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-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/
- Sync driver: https://mongodb.github.io/mongo-java-driver/3.3/
- Async driver: https://mongodb.github.io/mongo-java-driver/3.3/driver-async/
- How the RxJava driver and the Reactive Streams driver wrap the async driver: https://mongodb.github.io/mongo-java-driver/3.3/driver-async/reference/observables/
- RxJava driver: https://mongodb.github.io/mongo-java-driver-rx/1.2/
- Reactive Streams driver: https://mongodb.github.io/mongo-java-driver-reactivestreams/1.2/
- Morphia Driver: https://mongodb.github.io/morphia/1.1/

Resources – *Immutables* + *Avoiding null*

- Immutables:
  https://immutables.github.io/
- Using and avoiding null:
  https://github.com/google/guava/wiki/UsingAndAvoidingNullExplained

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 (2.4.8) Documentation for Java 8:
  http://doc.akka.io/docs/akka/2.4.8/java.html
- The Reactive Manifesto:
  http://www.reactivemanifesto.org/

Thanks for your attention!

Q & A