

# Hands on Lab JavaOne 2016

## Reactive Java - Promises and Streams with Reakt in Practice

### Overview of Reakt for the lab

[Reakt](#) is reactive interfaces for Java which includes:

- [Promises](#),
- [Streams](#),
- [Callbacks](#),
- [Async Results](#) with [Expected](#)
- [Circuit Breakers](#)

The emphasis is on defining interfaces that enable lambda expressions, and fluent APIs for asynchronous programming for Java.

Note: This mostly just provides the interfaces not the implementations. There are some starter implementations for Reakt but the idea is that anyone can implement this. It is all about interfaces. There are be adapters for [Vertx](#), [Guava](#), [Cassandra](#), etc. [Elekt](#) uses Reakt for its reactive leadership election. [Lokate](#) uses Reakt for client side service discovery for DNS-A, DNS-SRV, Consul and Mesos/Marathon. [QBit uses Reakt] for its reactor implementations and supports Reakt `Promises` and `Callbacks` as first class citizens.

You can use Reakt from gradle or maven.

### Using from maven

Reakt is published in the [maven public repo](#).

```
<dependency>
  <groupId>io.advantageous.reakt</groupId>
  <artifactId>reakt</artifactId>
  <version>2.6.0.RELEASE</version>
</dependency>
```

## Using from gradle

```
compile 'io.advantageous.reakt:reakt:2.6.0.RELEASE'
```

Reakt provides a fluent API for handling async calls.

## Fluent Promise API

```
Promise<Employee> promise = promise()
    .then(e -> saveEmployee(e))
    .catchError(error -> logger.error("Unable to lookup employee", error));

employeeService.lookupEmployee(33, promise);
```

Or you can handle it in one line by using an invokeable promise.

## Fluent Promise API example using an invokeable promise

```
employeeService.lookupEmployee(33,
    promise().then(e -> saveEmployee(e))
    .catchError(error ->
        logger.error("Unable to lookup ", error))
    );
```

## Promise concepts

This has been adapted from this [article on ES6 promises](#). A promise can be:

- resolved The callback/action relating to the promise succeeded
- rejected The callback/action relating to the promise failed

- pending The callback/action has not been resolved or rejected yet
- completed The callback/action has been resolved or rejected

Java is not single threaded, meaning that two bits of code can run at the same time, so the design of *this promise and streaming library* takes that into account.

There are three types of promises:

- Callback promises
- Blocking promises (for testing and legacy integration)
- Replay promises (allow promises to be handled on the same thread as caller)

This lab will cover all three as well as Promise coordination.

*Replay promises* are the most like their JS cousins but implemented with a MT world in mind. *Replay promises* are usually managed by the Reakt `Reactor` and supports environments like *Vert.x* and *QBit*. We will cover some examples of Replay promises.

Let's transition into some actual code examples and lab work.

## VirtualBox Credentials

- Box name: JavaOneReakt
- Computer name: java1-reakt
- Username: *dev*
- Password: *j1reakt!*

## Building and running the example

To do a complete build and run all of the tests navigate to the project folder and use gradle.

**build and run (don't run this yet)**

```
$ pwd
~/.../j1-talks-2016/labs/lab2-todo-cassandra

$ ./gradlew clean dockerTest build
```

This will run the docker containers and then run the tests.

This example works with *Cassandra*, *InfluxDB*, *Grafana*, and *StatsD*.

The `dockerTest` task is from a gradle plugin that starts up docker instances for testing. You can annotate your unit tests so that they depend on docker containers like InfluxDB, StatsD, Cassandra etc. You can read more about this [gradle docker plugin here](#).

The docker containers are specified in the build file.

## build.gradle

```
testDockerContainers {
    a_grafana {
        containerName "grafana_j1"
        image "advantageous/grafana:v1"
        portMapping(container: 9000, host: 3003)
        portMapping(container: 8086, host: 8086)
        portMapping(container: 8083, host: 3004)
        portMapping(container: "8125/udp", host: 8125)
    }
    b_elk {
        containerName "elk_j1"
        image "advantageous/elk:0.1"
        portMapping(container: 9200, host: 9200)
        portMapping(container: 5044, host: 5044)
        portMapping(container: 5000, host: 5000)
        portMapping(container: 5601, host: 5601)
        portMapping(container: "5001/udp", host: 5001)
        runArgs " /usr/local/bin/start.sh "
    }
}
```

```

    c_cassandra {
        publishAll true
        containerName "cassandra_j1"
        portMapping container: 9042, host: 39042
        image "cassandra:2.2.5"
        waitAfterRun 30
    }
}

```

## To run in the IDE run you first need to run downstream docker dependencies

```

$ ./gradlew startTestDocker
# then run things in IDE

```

## To stop docker container dependencies use this

```

$ ./gradlew startTestDocker
# then run things in IDE

```

Docker is setup on the machine so you can stop containers with `docker stop`, and remove them with `docker rm`. You may also need to get a list of containers with `docker ps` or `docker ps -a`. All of the docker containers are named (elk, grafana, and cassandra). (The command `docker stop elk` would stop the elk stack.)

When in doubt, reset the docker containers as follows:

## Reset the docker containers

```

$ docker stop grafana_j1 cassandra_j1 elk_j1
$ docker rm grafana_j1 cassandra_j1 elk_j1

```

Let's get started with writing code.

## Step 1 implement the add operation in TodoRepo

Add the `addTodo` operation in the `TodoRepo` class.

## ACTION Edit the file `./src/main/java/io/advantageous/j1/reakt/ToDoRepo` and finish `addTodo` method

```
package io.advantageous.j1.reakt;
...
import io.advantageous.reakt.promise.Promise;

//Used to map Guava futures used by Cassandra driver to Reakt promises
import static io.advantageous.reakt.guava.Guava.registerCallback;

//Used to return an invokeable Promise
import static io.advantageous.reakt.promise.Promises.invokeablePromise;
import static io.advantageous.reakt.promise.Promises.promise;
...

public class ToDoRepo {

    private final List<URI> cassandraUris;
    private final int replicationFactor;
    private final AtomicReference<Session> sessionRef = new AtomicReference<>();
    private final Logger logger = LoggerFactory.getLogger(ToDoRepo.class);

    public Promise<Boolean> addTodo(final Todo todo) {
        //Add invokeable promise
        return invokeablePromise(promise ->
            ifConnected("Adding todo", promise, () -> doAddTodo(promise, todo))
        );
    }
}
```

The method `Promise.invokeablePromise` returns an *invokeable promise*, which is a handle on an async operation call. The client code can register error handlers and async return handlers (callbacks) for the async operation and then `invoke` the async operation.

When you return a promise, client code can call your method as follows:

## INFO Calling this REPO from a service

```

    /** Send KPI addTodo called every time the addTodo method gets called. */
    mgmt.increment("addTodo.called");
    todoRep.addTodo(todo)
        .then(result -> {
            logger.error("Added todo to repo");
            promise.resolve(result);
        })
        .catchError(error -> {
            logger.error("Unable to add todo to repo", error);
            promise.reject("Unable to add todo to repo");
        })
        .invoke();

```

Notice you have different handlers for handling the successful outcome (`then`) versus the unsuccessful outcome (`catchError`).

## Background on promise handlers

Here are the different types of promises handlers.

- `then` - use this to handle async calls (success path)
- `catchError` - use this to handle async calls (error path)
- `thenExpected` - use this to handle async calls whose result could be null
- `thenSafe` - use this to report errors with async call and your handler
- `thenSafeExpected` - same as `thenSafe` but used where the result could be null
- `thenMap` - converts one type of promise into another type of promise

The handlers `thenExpect` and `thenSafeExpect` return a Reakt `Expected` instance. `Expected` is like `Option` in Java 8, it has methods like `map`, `filter`, etc. and adds methods `isEmpty`, `isEmpty`. This gives a nice fluent API when you don't know if a successful return is null or not.

The methods `then` and `thenSafe` async return the result that is not wrapped in an `Expected` object, i.e., the raw result. Use `then` and `thenSafe` when you know the async return will

not be null. Use `thenExpect` and `thenSafeExpect` if the value could be null or if you want to map or filter the result.

Use `thenMap` when a promise returns for example a `List<Employee>`, but you only want the first `Employee`. See [Promise.thenMap](#) for more details.

Note unless you are using a reactor, custom Promises or blocking promises, the `then*` handlers will typically run in a foreign thread and if they throw an exception depending on the library, they could get logged in an odd way. If you think your handler could throw an exception (not the service you are calling but your handlers), then you might want to use `thenSafe` or `thenSafeExpect`. These will wrap your async `then*` handler code in a `try/catch` and pass the thrown exception to a `ThenHandlerException` to `catchError`. If your code ever hangs when making an async call, try using a `thenSafe` or `thenSafeExpect`. They ensure that any exceptions thrown in your handler don't get dropped by the system you are using, which could indicate a lack of understanding of the async lib you are using or that you are using it wrong. If it hangs, try `thenSafe` or `thenSafeExpect`. They help you debug async problems.

## Step 2 finish the `ifConnected` method

Next we need to finish up the `ifConnected` operation

**ACTION** Edit the file `./src/main/java/io/advantageous/j1/reakt/ToDoRepo` and finish `ifConnected` method

```
private boolean isConnected() {
    return sessionRef.get() != null && !sessionRef.get().isClosed();
}

private void ifConnected(final String operation,
                        final Promise<> promise, final Runnable runnable) {
    // If we are not connected, try to connect, but fail this request.
    if (!isConnected()) {
        forceConnect();
    }
}
```



```

        //Promise rejected because we were not connected.
        promise.reject("Not connected to cassandra for operation " + operation);
    } else {
        // Try running the operation
        try {
            runnable.run();
        } catch (Exception ex) {
            //Operation failed, exit
            promise.reject("Error running " + operation, ex);
        }
    }
}

```

Notice that we catch the `Exception` and then call `promise.reject` to send the exception back to the handler. We also implement a fail fast operation if we are not yet connected or lost our connection (outage?). The fail fast operation attempts a reconnect.

## Step 3 Finish the `doAddTodo` method

**ACTION** Edit the file `./src/main/java/io/advantageous/j1/reakt/ToDoRepo` and finish `doAddTodo` method

```

    private void doAddTodo(final Promise<Boolean> promise, final Todo todo) {

        final Insert insert = QueryBuilder.insertInto("Todo")
            .value("id", todo.getId())
            .value("createTime", todo.getCreateTime())
            .value("name", todo.getName())
            .value("description", todo.getDescription());

        registerCallback(sessionRef.get().executeAsync(insert),
            promise(ResultSet.class)
                .catchError(promise::reject)
                .then(resultSet -> promise.resolve(resultSet.wasApplied())))
    }

```

```
    );  
}
```

The `registerCallback` method is from the [Guava integration with Reakt](#). Cassandra uses [Guava](#) as do many other libs for their async lib operations.

## Step 4 Finish the `addTodo` method in the service impl

**ACTION** Edit the file `./src/main/java/io/advantageous/j1/reakt/ToDoServiceImpl` and finish `addTodo` method

```
...  
@RequestMapping("/todo-service")  
public class ToDoServiceImpl implements ToDoService {  
...  
    @Override  
    @POST(value = "/todo")  
    public Promise<Boolean> addTodo(final Todo todo) {  
        logger.debug("Add Todo to list {}", todo);  
        return invokablePromise(promise -> {  
            /** Send KPI addTodo called every time the addTodo method gets called. */  
            mgmt.increment("addTodo.called");  
            todoRep.addTodo(todo)  
                .then(result -> {  
                    logger.error("Added todo to repo");  
                    promise.resolve(result);  
                })  
                .catchError(error -> {  
                    logger.error("Unable to add todo to repo", error);  
                    promise.reject("Unable to add todo to repo");  
                });  
        });  
    }  
}
```

A promise has to be resolved (`promise.resolve`) or rejected (`promise.reject`).

Once you are done editing the files, you can test them. There is a `ToDoRepoTest`.

## Step 5 Run the test

### src/test/java/i.a.j.r.TODORepoTest

```
@Category(DockerTest.class)
public class TODORepoTest {

    TODORepo todoRepo;

    @Before
    public void before() throws Exception {
        todoRepo = new TODORepo(1,
ConfigUtils.getConfig("todo").getConfig("cassandra").getUriList("uris"));
        todoRepo.connect().invokeAsBlockingPromise().get();
    }

    @Test
    public void addTODO() throws Exception {
        final Promise<Boolean> promise = todoRepo.addTODO(new TODO("Rick", "Rick",
System.currentTimeMillis()))
                .invokeAsBlockingPromise();
        assertTrue(promise.success());
        assertTrue(promise.get());
    }
    ...
}
```

Notice the above uses a `BlockingPromise`. A `BlockingPromise` is very much like a Java `Future`. It is blocking. This is useful for unit testing and for legacy integration.

The method `invokeAsBlockingPromise` has a version that takes a timeout duration so your tests do not hang forever if there is an error. The `invokeAsBlockingPromise` greatly simplifies testing of async software which can be a bit difficult.

To run the test, the docker containers have to be running.

You can control the docker containers from gradle.

- dockerTest - Run docker integration tests (works with tests that have `@Category(DockerTest.class)`)
- showDockerContainers
- stopTestDocker - Stop docker containers used in tests
- startTestDocker - Start docker containers used in tests

If you want to run the examples in the IDE, just run this once

```
$ docker startTestDocker
```

Then use the IDE to run the unit test.

## Step 6 Test addTodo using REST interface

### ACTION Run the app

```
$ gradle clean build run
```

The above should run the application and bind the service port to 8081 and the admin port to 9090.

### ACTION Add a TODO

```
$ curl -X POST http://localhost:8081/v1/todo-service/todo \
-d '{"name":"todo", "description":"hi", "id":"abc", "createTime":1234}' -H
"Content-type: application/json" | jq .
```

The above use curl to POST JSON Todo item to our example.

### ACTION Read Todos

```
$ curl http://localhost:8081/v1/todo-service/todo/ | jq .
```

You should be able to see the Todo item that you posted.

## Step 7 Using the reactor to track service actor state

### Overview of Step 7

This example uses a library that has implemented an efficient way to transmit metrics (APM). Let's say when we add `aTodo` that we want to track the number of errors and the number of successes. If you go back to the `addTodo` method (`TodoServiceImpl.addTodo`), you will notice that we do track the number of times `addTodo` has been called (by calling `mgmt.increment("addTodo.called");`).

### Background of Step 7

What you might not have know is that the call to `mgmt.increment` goes to the [Metrik](#) implementation provided by [QBit](#) (which can be queried at runtime for back pressure controls) which sends the messages to a [StatsD daemon](#) which then stores them into [InfluxDB Time series database](#) where you can visualize them with [Grafana](#) which is a metric and analytic dashboards. Once the data is in InfluxDB there are [APM](#) tools which can send notifications or take other actions (based on levels or anomaly detection.)

### Details of the reactor

The library that gathers the stats efficiently is stateful and depends on active object (or rather typed Actors or as I call them Service Actors). This means that the stat collection wants to happen in the same thread as the Service Actor.

The Reactor is a class that enables

- callbacks that execute in caller's thread (thread safe, async callbacks)
- tasks that run in the caller's thread
- repeating tasks that run in a caller's thread

- one shot after time period tasks that run in the caller's thread

The *reakt* Reactor is a lot like the QBit Reactor or the Vert.x context. It allows you to enable tasks that run in actors, service actors or verticles thread.

The *reakt* Reactor creates *replay promises*. Replay promises execute in the same thread as the caller. They are "replayed" in the callers thread.

QBit implements a service actor model (similar to Akka type actors), and Vert.x implements a Reactor model (like Node.js).

QBit, for example, ensures that all method calls are queued and handled by the service/actor thread. You can also use the *Reakt* Reactor to ensure that *callbacks/promises handlers* happen on the same thread as the caller. This allows the callbacks to be thread safe. In this example we are forcing the callback to be replayed in the same thread as the addMethod call (in a non-blocking fashion).

The Reakt Reactor is a drop in replacement for QBit Reactor except that the Reakt Reactor uses Reakt and QBit is moving towards Reakt. Promises, async Results and Callbacks. QBit 2 and

You can use the *Reakt* Reactor is not tied to QBit and you can use it with RxJava, Vert.x, or Spring Reactor and other similar minded projects to manage repeating tasks, tasks, and callbacks on the same thread as the caller (which you do not always need to do).

The Reactor is just an interface so you could replace it with an optimized version.

## Reactor Methods of note

Here is a high level list of Reactor methods.

- `addRepeatingTask(interval, runnable)` add a task that repeats every interval
- `runTaskAfter(afterInterval, runnable)` run a task after an interval expires
- `deferRun(Runnable runnable)` run a task on this thread as soon as you can

- `static reactor(...)` create a reactor
- `all(...)` create a promise that does not async return until all promises async return. (you can pass a timeout)
- `any(...)` create a promise that does not async return until one of the promises async return. (you can pass a timeout)
- `process` process all tasks, callbacks.

A Reactor provides *replay promise*, which are promises whose handlers (callbacks) can be replayed on the callers thread. To replay the handlers on this service actors thread (TodoServiceImpl), we can use the `Promise.invokeWithReactor` method as follows:

### ACTION Edit `src/main/java/io/advantageous/j1/reakt/TodoServiceImpl.java`

```
@Override
@POST(value = "/todo")
public Promise<Boolean> addTodo(final Todo todo) {
    logger.info("Add Todo to list {}", todo);
    return invokablePromise(promise -> {
        /** Send KPI addTodo called every time the addTodo method gets called. */
        mgmt.increment("addTodo.called");
        todoRep.addTodo(todo)
            .then(result -> {
                logger.info("Added todo to repo");
                promise.resolve(result);
                mgmt.increment("addTodo.called.success"); //TRACK SUCCESS
            })
            .catchError(error -> {
                logger.error("Unable to add todo to repo", error);
                promise.reject("Unable to add todo to repo");
                mgmt.increment("addTodo.called.failure"); //TRACK FAILURE
            })
            .invokeWithReactor(mgmt.reactor()); //USE THE Reactor
    });
}
```

Notice that `mgmt.increment` is not a thread safe calls. It keeps a local cache of counts, timings and such. We call it from the same thread as the service actor by using the `reactor (.invokeWithReactor(mgmt.reactor()))`.

## ACTION Run it

```
$ gradle clean build run
```

## ACTION Hit it with rest a few times

```
$ curl -X POST http://localhost:8081/v1/todo-service/todo \
-d '{"name":"todo", "description":"hi", "id":"abc", "createTime":1234}' -H
"Content-type: application/json" | jq .
```

Now go to [grafana](#) and look at the metrics. (Note this is a local link so we are assuming you are running the examples).