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 `Promise`s and `Callback`s are 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 and circuit `Breakers`.

Replay promises are the most like their JS cousins but implemented with a multithreaded 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*.

Note: Working from home and not JavaOne?

You can still grab the source code and follow along.

```
$ mkdir j1-reakt
$ cd j1-reakt

$ mkdir lab; cd lab
$ git clone -b https://github.com/advantageous/j1-talks-2016.git

$ cd labs/lab2-todo-cassandra

//# $ git clone -b https://github.com/advantageous/j1-talks-2016.git solution
```

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 stopTestDocker
# 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.invokablePromise;
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 invokablePromise(promise ->
                ifConnected("Adding todo", promise, () -> doAddTodo(promise, todo))
        );
    }
```

The method `Promise.invokablePromise` 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 promise 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 `ifEmpty`, `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 of 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

```
$ gradle startTestDocker
```

Then use the IDE to run the unit test.

Note: Make sure before you run the REST interface open that you modified the ***TodoServiceImpl*** and call the ***todoRepo.addTodo*** method.

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 a `Todo` 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`. `Promise`s, async `Result`s and `Callback`s. QBit 2 and

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).

For more details about grafana https://github.com/advantageous/docker-grafana-statsd.

Step 8 Add a circuit breaker

You will add a circuit breaker to managed the health of your Cassandra session.

Circuit Breakers Background

A `Breaker` is short for *Circuit Breaker*. The idea behind the breaker is to wrap access to a service so that errors can be tracked and the *circuit breaker* can open if errors are exceeded. Like all things in *Reakt* there is an interface for `Breaker` that defines a contract but other implementations can get creative on how they detect the `Breaker` has been thrown.

A *Circuit Breaker* in *Reakt* is not just for remote calls per se but any sort of out of process access (databases, remote services, message queue, remote pipe or stream).

Since a remote call or message can fail, you want to be able to detect it without blocking, and attempt to fix it. If the remote service cannot be fixed and its usage it not optional, then you want to mark your service as broken for alerting and so your services can be taken out of upstream discovery.

The *reactor* allows us to specify timeouts for the downstream services to return to us. This is to help deal with unresponsive supplier, and to not be an unresponsive supplier. Timeouts, async programming and the *circuit breaker* prevents cascading failures for upstream clients and services that use the `TodoService`.

Let's walk through an example. First we use `Breaker.opened` to create a circuit breaker for a Cassandra `session` that is open (open and broken mean the same thing with `Breaker`).

Then we use Reakt's `reactor` to run `circuitBreakerTest` after 60 seconds (`runTaskAfter`) for every 30 seconds (`addRepeatingTask`).

Creating a periodic health check for the TodoRepo called circuitBreakerTest

```java
public class TodoRepoImpl implements TodoRepo {
    ...
    // Breaker to hold the session is initially open.
    private Breaker<Session> sessionBreaker = Breaker.opened();
    ...

    // Check the session breaker health after 60 seconds, check every 30 seconds.
    @PostConstruct
    private void start() {
        reactor.runTaskAfter(Duration.ofSeconds(60), () -> {
            logger.info("Registering health check and recovery for repo");
            reactor.addRepeatingTask(Duration.ofSeconds(30),
            this::circuitBreakerTest);
        });
    }
```

Breaker has methods like `ifBroken`, and `cleanup` to check if a the *circuit breaker* is open and to do cleanup on the circuit breaker.

circuitBreakerTest: Clean up the session if the breaker is broken

```java
   private void circuitBreakerTest() {
        sessionBreaker.ifBroken(() -> {
            //Alert monitoring system.
            serviceMgmt.increment("repo.breaker.broken");

            //Clean up the old session.
            sessionBreaker.cleanup(session -> {
                try {
                    if (!session.isClosed()) {
                        session.close();
                    }
                } catch (Exception ex) {
                    logger.warn("unable to clean up old session", ex);
                }
            });
        });
        ...

    }
```

Notice the following tracks the number of times that we retry to `connect` with `notConnectedCount`. Also notice that if we hit a certain limit of connection retries as an example we call `serviceMgmt.setFailingWithError` which will mark the entire microservice as failed which could mean *for example* that `Mesos` will remove the container and attempt a redeploy or if you are using `Consul`, this could mean that `Consul` takes this microservice out of its list of healthy nodes for upstream discovery. (Note `serviceMgmt` is not part of *Reakt*, it is used as an example for alerting and monitoring).

After we cleanup the old session, we connect to a new one.

```
...
public class TodoRepoImpl implements TodoRepo {
...
    private void circuitBreakerTest() {
          ...
            //Clean up the old session.
          ...
            //Connect to repo.
            connect().catchError(error -> {
                notConnectedCount++; //Limit retry attempts.
                logger.error("Not connected to repo " + notConnectedCount, error);
                ... // send error stats
                if (notConnectedCount > 10) {
                    logger.error("Attempts to reconnect to Cassandra have failed.
Marking repo as failed.");
                    serviceMgmt.increment("repo.connect.error.fatal");
                    serviceMgmt.setFailingWithError(error);
                }
            }).thenSafe(connected -> {
                //If the TodoRepo service is failing, recover on connect.
                if (serviceMgmt.isFailing()) {
                    serviceMgmt.increment("repo.connect.recover");
                    serviceMgmt.recover();
                }
                notConnectedCount = 0;
            }).invokeWithReactor(reactor);
        });
    }
```

We have an async call to `discoveryService.lookupService` whose results we use to make an async call to `Builder.connectAsync`, then we send back the results to the invokablePromise of the `TodoRepoImpl.connect` method. The most important bit is that we have another async call to an internal method (connect()->lookupService()->connectAsync()/futureToPromise()->buildDBIfNeeded()) whose `then` handler creates the session *circuit breaker*.

The connect method, good example of call coordination.

```
public class TodoRepoImpl implements TodoRepo {
  ...
    @Override
    public Promise<Boolean> connect() {
        return invokablePromise(promise -> {
            serviceMgmt.increment("connect.called");

            discoveryService.lookupService(cassandraURI).thenSafe(cassandraUris -> {
                serviceMgmt.increment("discovery.service.success");

                final Builder builder = builder();
                cassandraUris.forEach(cassandraURI1 ->
builder.withPort(cassandraURI1.getPort())
                        .addContactPoints(cassandraURI1.getHost()).build());
                futureToPromise(builder.build().connectAsync()) //Cassandra / Guava
Reakt bridge.
                        .catchError(error -> promise.reject("Unable to load initial
session", error))
                        .then(sessionToInitialize ->
                                buildDBIfNeeded(sessionToInitialize)
                                        .thenSafe(session -> {
                                            cassandraErrors.set(0);
                                            sessionBreaker =
Breaker.operational(session, 10, theSession->
                                                !theSession.isClosed() &&
cassandraErrors.incrementAndGet() > 25
                                            );
                                            promise.resolve(true);
                                        })
                                        .catchError(error ->
                                                promise.reject(
                                                        "Unable to create or initialize
session", error)
                                        ).invokeWithReactor(reactor)
                        ).invokeWithReactor(reactor);

            }).catchError(error ->
serviceMgmt.increment("discovery.service.fail")).invokeWithReactor(reactor);

        });
    }
```

The connect method just as an example uses Lokate's `discoveryService` (which works with DNS A, DNS SRV, Consul, and Mesos). This is a good example of *Reakt* call coordination as we call the async `discoveryService` to lookup the nodes of the Cassandra service `then` we use the Cassandra driver's `Builder` to build a Cassandra using Cassandra's async API which relies on Guava, and we use the *Reakt Guava Bridge* `futureToPromise(future)` to convert the non-lambda friendly Guava `future` returned from `connectAsync()` into a *Reakt* `invokablePromise`.

The `then` handler for `buildDBIfNeeded` creates the session *circuit breaker*.

connect()->lookupService()->connectAsync()/futureToPromise()->buildDBIfNeeded

```
cassandraErrors.set(0);
sessionBreaker = Breaker.operational(session, 10,
    theSession->
      !theSession.isClosed() && cassandraErrors.incrementAndGet() > 25);
```

The `Breaker.operational` method creates a closed `Breaker` (ok, operational and closed are synonyms.) We pass 10, which means the `session` can throw ten errors before we consider it broken (this is an optional parameter). We also pass it a session predicate so we can customize the `isBroken` behavior with an additional check. In this case, our additional check checks to see if the session is closed or if we received more than 25 async errors from Cassandra.

The `Breaker` can be used in the `addTodo` method and the `loadTodos` method of the `TodoRepoImpl`.

Using Breaker to fail fast from interface methods

```
   @Override
    public Promise<Boolean> addTodo(final Todo todo) {
        logger.info("Add Todo called");
        return invokablePromise(promise -> sessionBreaker
                .ifBroken(() -> {
                    final String message = "Not connected to cassandra while adding
todo";
                    promise.reject(message);
                    logger.error(message);
                    serviceMgmt.increment("cassandra.breaker.broken");
                })
                .ifOperational(session ->
                        futureToPromise(session.executeAsync(insertInto("Todo")
                                .value("id", todo.getId())
                                .value("createTime", todo.getCreateTime())
                                .value("name", todo.getName())
                                .value("description", todo.getDescription()))
                        ).catchError(error -> {
                            serviceMgmt.increment("add.todo.fail");
                            serviceMgmt.increment("add.todo.fail." +
                                    error.getClass().getName().toLowerCase());
                            recordCassandraError();
                            promise.reject("unable to add todo", error);
                        }).then(resultSet -> {
                            if (resultSet.wasApplied()) {
                                promise.resolve(true);
                                serviceMgmt.increment("add.todo.success");
                            } else {
                                promise.resolve(false);
                                serviceMgmt.increment("add.todo.fail.not.added");
                            }
                        }).invokeWithReactor(reactor, Duration.ofSeconds(10)))
        );
    }

    private void recordCassandraError() {
        cassandraErrors.incrementAndGet();
        serviceMgmt.increment("cassandra.error");
    }

    @Override
    public Promise<List<Todo>> loadTodos() {
        return invokablePromise(promise -> sessionBreaker
                .ifBroken(() -> {
                    final String message = "Not connected to Cassandra while loading
todo";
                    promise.reject(message);
                    logger.error(message);
                })//ifBroken
                .ifOperational(session ->
                        futureToPromise(
session.executeAsync(select().all().from("Todo").where().limit(1000))
                        ).catchError(error -> {
                            recordCassandraError();
                            promise.reject("Problem loading Todos", error);
                        }).thenSafe(resultSet ->
                                promise.resolve(

resultSet.all().stream().map(this::mapTodoFromRow)
                                                .collect(Collectors.toList())
                                )
                        ).invokeWithReactor(reactor)
                )//ifOperational
        );
    }
```

The `reactor` has default timeouts for promise construction, but you can override the timeouts when you create the promise or use `invokeWithReactor` (`invokeWithReactor(reactor, Duration.ofSeconds(10))`).

ACTION pull down the labs and the solutions into two separate directories.

```
$ mkdir breaker
$ cd breaker
$ git clone -b breaker-lab https://github.com/advantageous/j1-talks-2016.git


# SOLUTION $ git clone -b circuit-breaker-connection-cleanup
https://github.com/advantageous/j1-talks-2016.git solution
```

Modify /lab/lab2-todo-cassandra/src/main/java/io.advantageous.j1.reakt.repo/TodoRepoImpl

Modify the file `TodoRepoImpl` and follow the instructions in the comments that say TODO.

Validate using curl commands

Run the app with gradle run and use the curl commands from earlier to test the application. Also run the unit test.

Step 9 Call coordination using all and nested promises

Let's say Todo items can be edited and created by many users in a collaborative fashion at once, and Todo items can be pushed into this system through legacy integration. Don't put too much thought in the actual use case because this is all a contrived anyway.

In this contrived example we want to update two tables during the `addTodo` operation, namely, `Todo` and`TodoLookup`.

Update two tables

```
CREATE KEYSPACE IF NOT EXISTS  todoKeyspace2
with REPLICATION =  { 'class' : 'SimpleStrategy', 'replication_factor' : 1 }

USE todoKeyspace2
CREATE TABLE IF NOT EXISTS Todo (
                        id text,
                        name text,
                        version bigint,
                        description text,
                        updatedTime timestamp,
                        createdTime timestamp,
                        primary key (id, updatedTime)
                    )
                    WITH CLUSTERING ORDER BY ( updatedTime desc )

CREATE TABLE IF NOT EXISTS TodoLookup (
                        id text,
                        updatedTime timestamp,
                        primary key (id, updatedTime)
                    )
                    WITH CLUSTERING ORDER BY ( updatedTime asc )
```

`TodoLookup` stores the data by updatedTime time ascending so you can quickly look up a Todo item by its earliest date. The `Todo` table stores by descending updatedTime so you can quickly look up the last version of the Todo item. (Do not take this in any way shape or form as a best practices recommendation for using Cassandra.)

Now we want to update both tables when we save a Todo item and we only want to `resolve` the `promise` to the `addTodo` when both table saves succeed.

This is where an `all` promise comes in.

Let's break the addTodo into two methods to make it easier to follow:

addTodo that just delegates to doAddTodo

```
@Override
public Promise<Boolean> addTodo(final Todo todo) {
    logger.info("Add Todo called");
    return invokablePromise(promise -> sessionBreaker
            .ifBroken(() -> {
                final String message = "Not connected to cassandra while adding todo";
                promise.reject(message);
                logger.error(message);
                serviceMgmt.increment("cassandra.breaker.broken");
            })
            .ifOperational(session ->
                    doAddTodo(todo, promise, session)
            )
    );
}
```

The `doAddTodo` uses the reactor to make two updates to the database as follows:

Make to async updates to the database and don't resolve the promise until both come back

```
private void doAddTodo(final Todo todo,
                       final Promise<Boolean> returnPromise,
                       final Session session) {



    reactor.all(Duration.ofSeconds(30),
            //Call to save Todo item in two table, don't respond until both calls come
back from Cassandra.
            // First call to cassandra.
            futureToPromise(
                    session.executeAsync(insertInto("Todo")
                            .value("id", todo.getId())
                            .value("updatedTime", todo.getUpdatedTime())
                            .value("createdTime", todo.getCreatedTime())
                            .value("name", todo.getName())
                            .value("description", todo.getDescription()))
            ).catchError(error -> recordCassandraError("add.todo", error))
                    .thenSafe(resultSet -> handleResultFromAdd(resultSet, "add.todo")),
            // Second call to cassandra.
            futureToPromise(
                    session.executeAsync(insertInto("TodoLookup")
                            .value("id", todo.getId())
                            .value("updatedTime", todo.getUpdatedTime()))
            ).catchError(error -> recordCassandraError("add.lookup", error))
                    .thenSafe(resultSet -> handleResultFromAdd(resultSet,
"add.lookup"))
    ).catchError(returnPromise::reject)
            .then(v -> returnPromise.resolve(true))
            .invoke();

}
```

Notice we make two async calls to store Todo data into Cassandra

- `futureToPromise(session.executeAsync(`. The two async calls return promises. The all method takes a list or array of promises (`reactor.all(Duration.ofSeconds(30), promise1, promise2`).

ACTION pull down the labs and the solutions into two separate directories.

```
$ mkdir call-coordination
$ cd call-coordination
$ git clone -b call-coordination-lab https://github.com/advantageous/j1-talks-2016.git
lab
$ git clone -b call-coordination https://github.com/advantageous/j1-talks-2016.git
solution
```

Modify lab/lab2-todo-cassandra/src/main/java/io.advantageous.j1.reakt.repo/TodoRepoImpl

Modify the file `TodoRepoImpl` and follow the instructions in the comments that say TODO.

Validate using unit tests

Run the unit tests from the IDE or use `gradle clean build` from the command line.

Next we want to create a `loadTodo(id)` method that loads the latest `Todo` object, and if it does not have a`createdTime`, it uses another async method to load the first version of this `Todo` item. The trick here is the results of first async call or the second async call may resolve the original call. This is call coordination.

Call coordination find createdTime if not set.

```
/**
 * This always has to load a Todo with a createdTime.
 * The Todo item might not exist so use Reakt Expected (which is like Java Optional).
 * You are expecting this to return a Todo, but it might not.
 */
@Override
public Promise<Expected<Todo>> loadTodo(final String id) {
    logger.info("Load Todo called");
    return invokablePromise(returnPromise -> sessionBreaker
            .ifBroken(() -> {
                final String message = "Not connected to cassandra while loading a todo
item";
                returnPromise.reject(message);
                logger.error(message);
                serviceMgmt.increment("cassandra.breaker.broken");
            })
            .ifOperational(session ->
                    futureToPromise(
                            //Cassandra query.
                            session.executeAsync(select().all().from("Todo")
                                    .where(eq("id", id))
                                    .limit(1))
                    ).catchError(error -> {
                        //Failure.
                        recordCassandraError("load.todo", error);
                        returnPromise.reject("Problem loading Todos", error);
                    }).thenSafe(resultSet -> {
                        final Row row = resultSet.one();
                        //Nothing found so send them an empty result.
                        if (row == null) {
                            returnPromise.resolve(Expected.empty());
                        } else {
                            final Todo todo = mapTodoFromRow(row);
                      // The Todo has a created time, so send it back now.
                            if (todo.getCreatedTime() != 0) {
                                returnPromise.resolve(Expected.of(todo));
                            } else {
                      //We need to find the create time before we send it back.
                      //Next time it is updated, it will have the created time.
                                loadFirstTodoCreateTime(session, id)
                                .thenSafe(createdTime -> {
                                            returnPromise.resolve(
                                            Expected.of(new Todo(todo,
                                                     createdTime)));
                                        })
                                .catchError(error -> returnPromise.reject(
                                        "Created time not found"))
                                .invoke();
                            }
                        }
                    }).invokeWithReactor(reactor)
            )
    );
}

private Promise<Long> loadFirstTodoCreateTime(final Session session,
                                              final String id) {
    return invokablePromise(returnPromise ->
            futureToPromise(
                    session.executeAsync(select().all().from("TodoLookup")
                            .where(eq("id", id)).and(gte("updatedTime", 0L)).limit(1))
            )
            .catchError(error -> recordCassandraError("load.todo", error))
            .thenSafe(resultSet -> {
                        final Row row = resultSet.one();
                        if (row == null) {
                            returnPromise.resolve(-1L);
                        } else {
                            returnPromise.resolve((row.getTimestamp("updatedTime")
                                            .getTime()));
                        }
                    }
            ).invokeWithReactor(reactor));
}
```

Next let's show an `any` example. Well, in our contrived example, someone started complaining that every now and then they are getting a timeout error in an upstream service. We decided we would like the data saved, but there is no use making the clients wait. We added a retry reconciliation queue, based on Kafka, and a process that can detect timeout errors and run a reconciliation.

Now we want to change our `addTodo` method to save the added Todo to Cassandra and send it to the retry reconciliation queue. We don't care which one happens first. But, we want at least the enqueue operation to work or the Cassandra update to work (in our contrived example).

Example using any

```
   private void doAddTodo(final Todo todo,
                            final Promise<Boolean> returnPromise,
                            final Session session) {

        reactor.any(
                messageQueue.sendToQueue(todo)
                        .catchError(error -> logger.error("Send to queue failed",
error))
                        .thenSafe(enqueued -> logger.info("Sent to queue")),
                reactor.all(

                        //Call to save Todo item in two table, don't respond until both
calls come back from Cassandra.
                        // First call to cassandra.
                        futureToPromise(
                                session.executeAsync(...//
                        ).catchError(error -> recordCassandraError(...))
                         .thenSafe(resultSet ->handleResultFromAdd(...)),
                        // Second call to cassandra.
                        futureToPromise(
                                session.executeAsync(...
                        ).catchError(error -> recordCassandraError(...))
                         .thenSafe(resultSet -> handleResultFromAdd(...))
                )
        ) .catchError(returnPromise::reject)
          .then(v -> returnPromise.resolve(true))
          .invoke();
    }
```

Where `reactor.all()` will not trigger the final `returnPromise.resolve()` until all promises come back, the `reactor.any()` will trigger as soon as one async call comes back. Reactor all and any provide Timeouts just in case no calls come back, the client is not left hanging. You can use `any()` and `all()` without a reactor by using `Promises.any()` and `Promises.all()`. The reactor also forces the callbacks to happen on this actor's thread.

Action: Extra credit use MessageQueue with reactor.any in the doAddTodo method.

A fuller example using Kafka, multiple services, Promises.all

This example implements a sample subscription service. It calls a third party service. Later we show an example how to use Kafka with Reakt.

Step 1 implement the store operation in SubscriptionRepository

Add the `store` operation in the `SubscriptionRepository` class.

ACTION Edit the file ./src/main/java/io/advantageous/reakt/examples/repository/SubscriptionRepository and finish store method

```
package io.advantageous.reakt.examples.repository;

...

public class SubscriptionRepository {
    private final CassandraTemplate<Subscription> cassandraTemplate;
    ...

    public SubscriptionRepository(final int replicationFactor, final List<URI>
cassandraUris) {
        cassandraTemplate = new CassandraTemplate<>(replicationFactor,
                                                     cassandraUris, TABLE_DEFINITION,
KEYSPACE);
    }

    public Promise<Boolean> store(Subscription subscription){
        return invokablePromise(promise -> {

                }
            );
    });
    }
```

The method `Promise.invokablePromise` 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

```
         mgmt.increment(MGMT_CREATE_KEY);
          repository.store(subscription)
                    .then(result -> {
                        logger.info("subscription created id");
                        promise.resolve(result);
                    })
                    .catchError(error -> {
                        logger.error("Unable to create subscription", error);
                        promise.reject("Unable to create subscription");
                    })
                    .invoke();
```

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

Background on promise handlers

Remember 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

Reminder 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/reakt/examples/template/CassandraTemplate 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 of lost our connection (outage?). The fail fast operation attempts a reconnect.

Step 3 Finish the insert method

ACTION Edit the file ./src/main/java/io/advantageous/reakt/examples/template/CassandraTemplate and finish insert method

```
   public void insert(Promise<Boolean> promise, Insert insert){
            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 create method in the service impl

ACTION Edit the file ./src/main/java/io/advantageous/reakt/examples/service/SubscriptionServiceImpl and finish create method

```
...

@RequestMapping("/subscription-service")
public class SubscriptionServiceImpl implements SubscriptionService {
    private static final String PATH = "/subscription";

    ...

    @Override
    @POST(value = PATH)
    public Promise<Boolean> create(final Subscription subscription) {
        return invokablePromise(promise -> {
            mgmt.increment(MGMT_CREATE_KEY);

            repository.store(subscription)
                    .then(result -> {
                        logger.info("subscription created id");
                        promise.resolve(result);
                    })
                    .catchError(error -> {
                        logger.error("Unable to create subscription", error);
                        promise.reject("Unable to create subscription");
                    })
                    .invoke();

        });
    }
```

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 `SubscriptionRepoTest`.

Step 5 Run the test

src/test/java/io/advantageous/reakt/examples/repository/SubscriptionRepoTest

```
@Category(DockerTest.class)
public class SubscriptionRepoTest {
    private SubscriptionRepository repository;
    private Subscription subscription;

    @Before
    public void before() throws Exception {
        String id = UUID.randomUUID().toString();
        String thirdPartyId = UUID.randomUUID().toString();
        String name = "test subscription";
        long createTime = System.currentTimeMillis();

        subscription = new Subscription(id, name, thirdPartyId, createTime);


        repository = new SubscriptionRepository(1,
ConfigUtils.getConfig("subscription")
                                                              .getConfig("cassandra")
                                                              .getUriList("uris"));

        repository.connect().invokeAsBlockingPromise().get();
        Thread.sleep(1000);
    }

    @After
    public void after() throws Exception {
        repository.close();
    }

    @Test
    public void testStore() throws Exception {
        final Promise<Boolean> promise = repository.store(subscription)
                                                    .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 create 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 Subscription

```
$ curl -X POST http://localhost:8082/v1/subscription-service/subscription \
  -d '{"name":"test", "thirdPartyId":"1234"}' -H "Content-type: application/json" | jq
.
```

The above uses curl to POST JSON Subscription to our example.

ACTION Read Subscriptions

```
$ curl http://localhost:8082/v1/subscription-service/subscription | jq .
```

You should be able to see the Subscription 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 a `Subscription` that we want to track the number of errors and the number of successes. If you go back to the create method (`SubscriptionServiceImpl.create`), you will notice that we do track the number of times `create` has been called (by calling `mgmt.increment(MGMT_CREATE_KEY);`).

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.)

As you might recall, QBit implements a service actor model (similar to Akka type actors), and Vert.x implements a Reactor model (like Node.js). A Reactor would work well in a Verticle, actor or a service actor.

Reactor Methods we could use

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 (`SubscriptionServiceImpl`), we can use the`Promise.invokeWithReactor` method as follows:

ACTION Edit src/main/java/io/advantageous/reakt/examples/ervice/SubscriptionServiceImpl.java

```
...

@RequestMapping("/subscription-service")
public class SubscriptionServiceImpl implements SubscriptionService {
    private static final String PATH = "/subscription";

    ...

    @Override
    @POST(value = PATH)
    public Promise<Boolean> create(final Subscription subscription) {
        return invokablePromise(promise -> {
            mgmt.increment(MGMT_CREATE_KEY);

            repository.store(subscription)
                    .then(result -> {
                        logger.info("subscription created id");
                        promise.resolve(result);
                    })
                    .catchError(error -> {
                        logger.error("Unable to create subscription", error);
                        promise.reject("Unable to create subscription");
                    })
                    .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:8082/v1/subscription-service/subscription \
  -d '{"name":"test", "thirdPartyId":"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).

Example using Kafka with Reakt (streams)

We have this message service interface.

```
public interface MessageService {

    Promise<Boolean> publish(Message message);

}
```

We want to provide an implementation that uses Kafka to send publish a message to a Kafka topic and then consume that message and log the message.

The `MessageService` impl will use a `Producer` that uses an invokakable promise to call call the Kafka producer API async, then uses the `promise.resolve` and `promise.reject` of the invokeable to bridge from the async Kafka world to the Reakt world.

Using Kafka async lib with Reakt to publish a message

```
   public Promise<Boolean> send(String key, String message) {
        return invokablePromise(promise ->
                producer.send(new ProducerRecord<>(topic, key, message), (m, e) -> {
                    if (m != null) {

                        logger.info("message " + message + " sent to partition(" +
m.partition() + "), " +
                                "offset(" + m.offset() + ") at " +
System.currentTimeMillis());

                        promise.resolve(true);
                    } else {
                        promise.reject(e);
                    }
                }));
```

```
    }
```

Now we use a `Reakt` stream to subscribe to a Kafka message stream for a give topic. Again we use the `Reakt`labmda friendly `Stream` and adapt it to the `Kafka` stream.

Using Kafka async lib with Reakt to consume a stream of messages

```
   public Promise<Boolean> consume(String topic, Stream<String> stream) {
        return invokablePromise(promise -> {
            KafkaStream<byte[], byte[]> kafkaStream =
                    consumer.createMessageStreams(
                            new HashMap<String, Integer>(){
                                {
                                    put(topic, 1);
                                }
                            }
                    ).get(topic).get(0);
            executor.submit(() -> kafkaStream.forEach(
                    data -> stream.reply(new String(data.message()))));
        });
    }
```

Action: Modify MessageServiceImpl (src/main/java/io/advantageous/reakt/examples/service/MessageServiceImpl.java)

Just modify the file and follow the TODO instructions.

Action: Modify Consumer (src/main/java/io/advantageous/reakt/examples/messaging/Consumer.java)

Just modify the file and follow the TODO instructions.

Action: Modify Producer (src/main/java/io/advantageous/reakt/examples/messaging/Producer.java)

Just modify the file and follow the TODO instructions.

ACTION Run it

```
$ gradle clean build run
```

ACTION Hit it with rest a few times

```
$ curl -X POST http://localhost:8082/v1/subscription-service/message \
  -d '{"name":"test", "id":1234}' -H "Content-type: application/json" | jq .
```