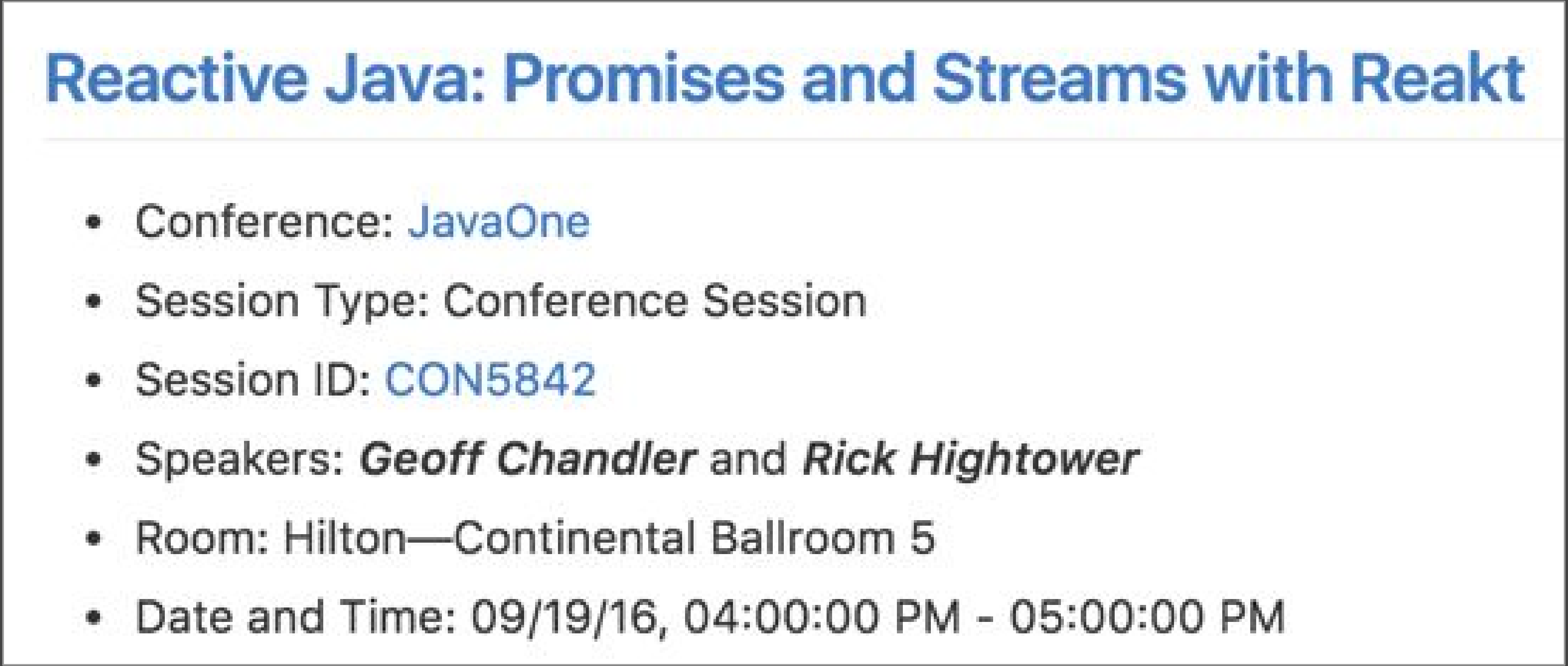


Reactive Java: Promises and Streams with Reakt

- Conference: JavaOne
- Session Type: Conference Session
- Session ID: CON5842
- Speakers: ***Geoff Chandler*** and ***Rick Hightower***
- Room: Hilton—Continental Ballroom 5
- Date and Time: 09/19/16, 04:00:00 PM - 05:00:00 PM

Reactive Java: Promises and Streams with Reakt

Geoff Chandler and Rick Hightower

Speaker Introduction

Geoff Chandler (@SailorGeoff)

Rick Hightower (@RickHigh)

What is Reakt in 30 seconds!

Reakt

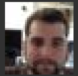
General purpose library for **callback coordination** and **streams**

Implements **JavaScript style Promises** and adapts them to a MT world

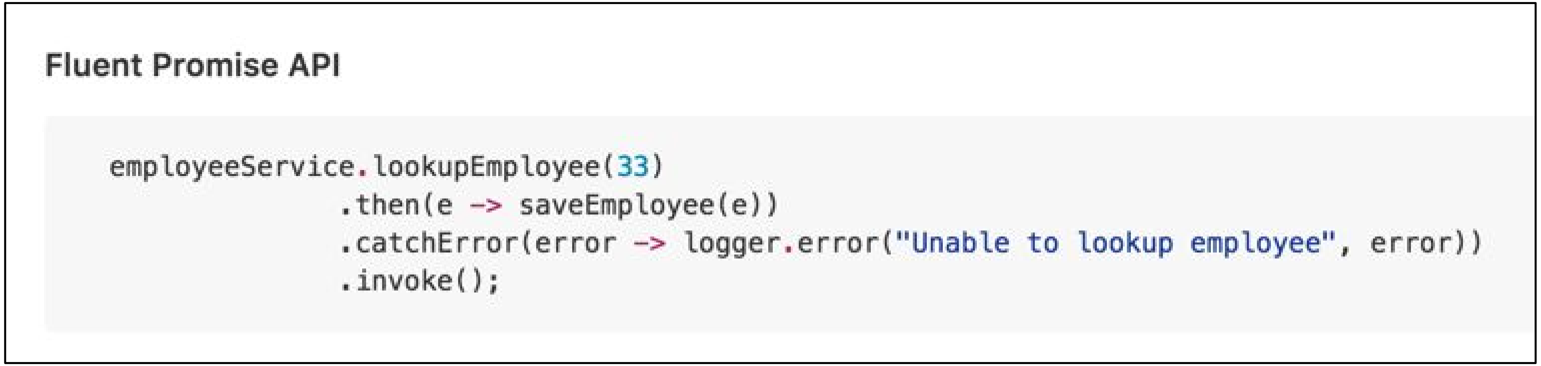
Can be used with

- Reactor pattern systems,
- actor system, event bus system,
- and traditional forms of async Java programming

Lambda expression friendly

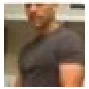


Fluent Lambda Friendly Promise API



Fluent Promise API

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



About the Project

Goals

Small and focused

Easy-to-use

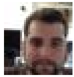
Lambda friendly

Scalar async calls, and streams

Fluent

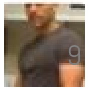
Evolve it (no change for the sake of change but get it right)

- Semantic versioning



More Project Goals

- Should work with
  - actor model
  - MT model
  - Reactor Pattern (event loop)
- Supports
  - Async call coordination
  - Complex call coordination
  - and streams
- Define interfaces and allow for other implementations
  - Allow core classes to be implemented by others



Problem: Async Coordination is Tricky

Results can comeback on foreign threads

Call several async services, one fails or times out? Now what?

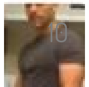
Need to combine results of several calls before you can respond to the caller

What if a downstream service goes down?

What if your async handler throws an exception?

You can’t use a blocking future, or you will tie up the event handling thread

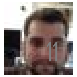
How do you test async services using a standard unit test framework?



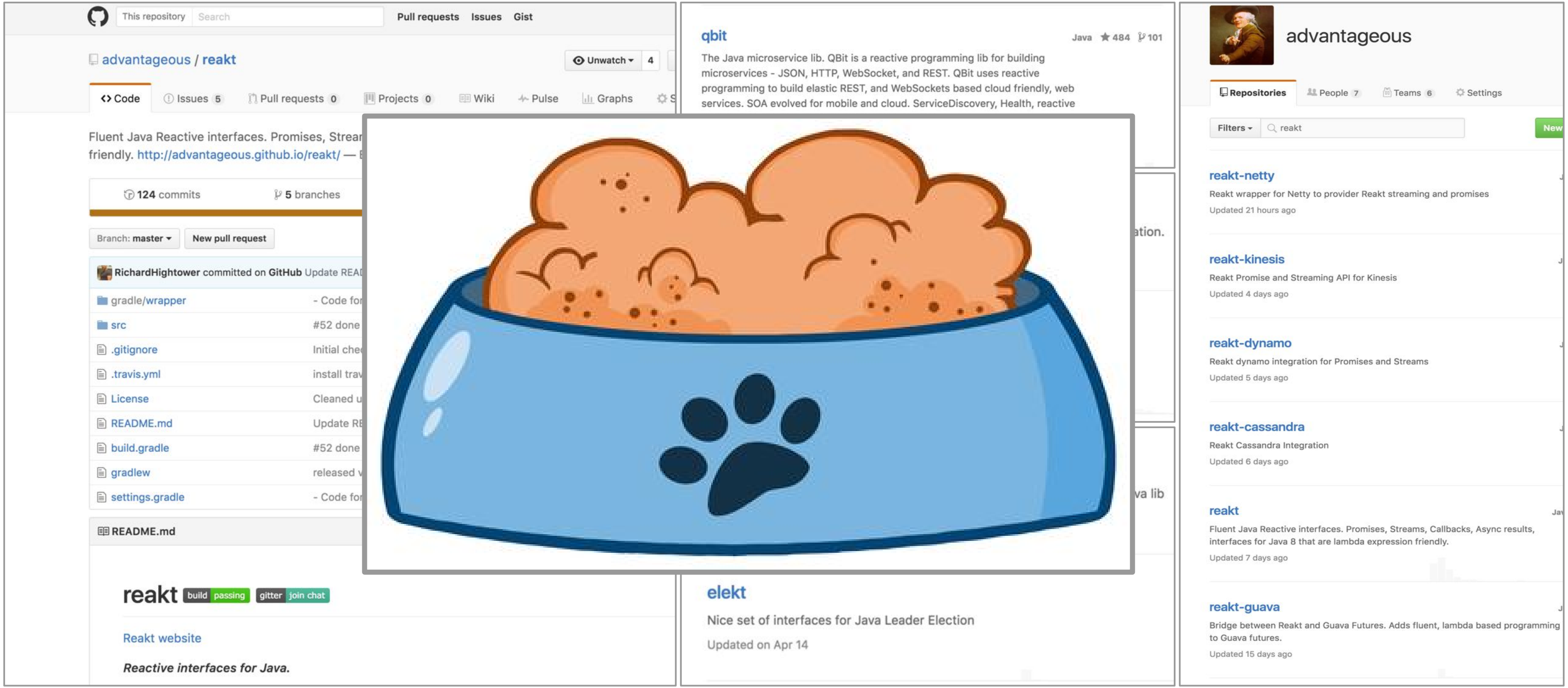
Status

We use it often. We like it.

- Integration libs for **Guava**, **Vert.x**, Netty (in the works), Kinesis, **Cassandra**, DynamoDB, etc.
- Write async call handling for **Lokate**, and **Elekt**
- **QBit** now uses it instead of its own callbacks and async coordination
- **Version 3.1** has major stability improvements (a lot of work was done)
- **Version 4** - Major update to simplify and refine interfaces (after JavaOne)
    - Won’t change how it is used, but will clean up the interfaces and simplify them
    - Delayed until after JavaOne (this will be version 4)



Open Source on GitHub - We use it



advantageous / reakt

Fluent Java Reactive interfaces. Promises, Strea
friendly. http://advantageous.github.io/reakt/ — E

124 commits 5 branches

RichardHightower committed on GitHub Update REA

gradle/wrapper	- Code fo
src	#52 done
.gitignore	Initial che
.travis.yml	install tra
License	Cleaned u
README.md	Update R
build.gradle	#52 done
gradlew	released
settings.gradle	- Code fo

README.md

reakt

Reakt website

Reactive interfaces for Java.

qbit Java ★ 484 ⑂ 101

The Java microservice lib. QBit is a reactive programming lib for building microservices - JSON, HTTP, WebSocket, and REST. QBit uses reactive programming to build elastic REST, and WebSockets based cloud friendly, web services. SOA evolved for mobile and cloud. ServiceDiscovery, Health, reactive

ation.

va lib

elekt

Nice set of interfaces for Java Leader Election

Updated on Apr 14

advantageous

reakt

reakt-netty

Reakt wrapper for Netty to provider Reakt streaming and promises

Updated 21 hours ago

reakt-kinesis

Reakt Promise and Streaming API for Kinesis

Updated 4 days ago

reakt-dynamo

Reakt dynamo integration for Promises and Streams

Updated 5 days ago

reakt-cassandra

Reakt Cassandra Integration

Updated 6 days ago

reakt

Fluent Java Reactive interfaces. Promises, Streams, Callbacks, Async results, interfaces for Java 8 that are lambda expression friendly.

Updated 7 days ago

reakt-guava

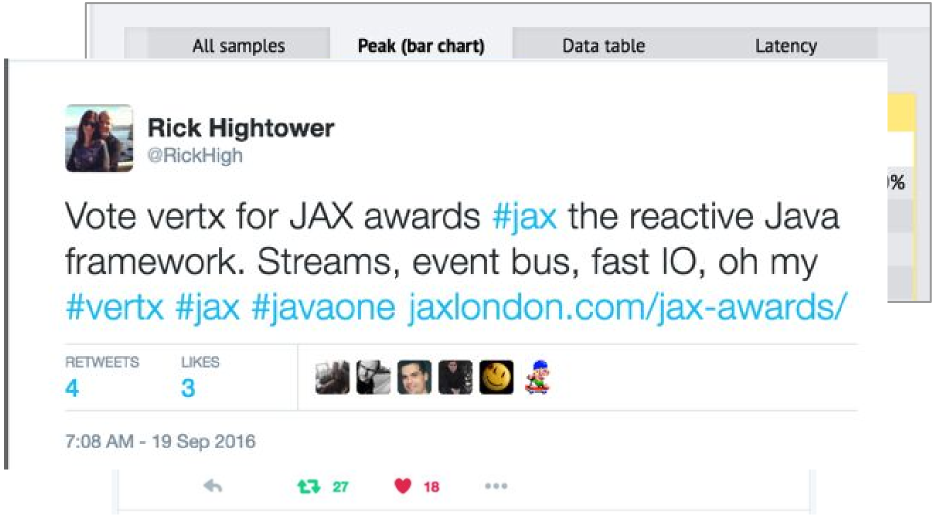
Bridge between Reakt and Guava Futures. Adds fluent, lambda based programming to Guava futures.

Updated 15 days ago

How we got here. Why Promises?

Implementations of Reactor Pattern

- Browser DOM model and client side JavaScript
- AWT/GTK+/Windows C API
- Twisted - Python
- Akka's I/O Layer Architecture
- Node.js - JavaScript
- **Vert.x - (Netty)**
  - **Multi-Reactor pattern**
- Spring 5 Reactor



Rick Hightower
@RickHigh

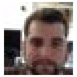
Vote vertx for JAX awards #jax the reactive Java framework. Streams, event bus, fast IO, oh my #vertx #jax #javaone jaxlondon.com/jax-awards/

RETWEETS 4 LIKES 3

7:08 AM - 19 Sep 2016

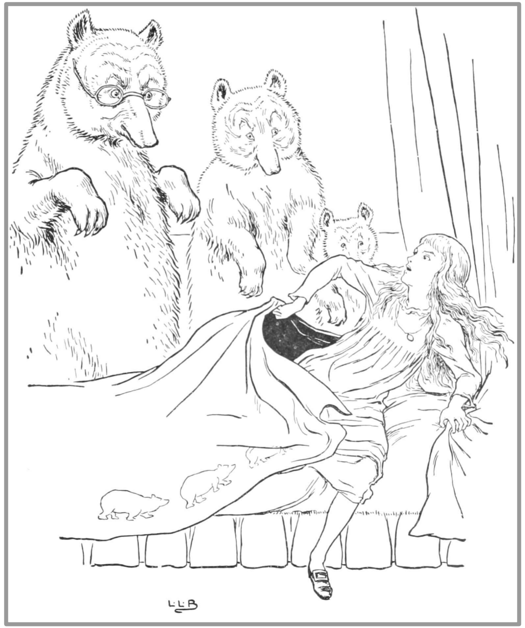
27 18

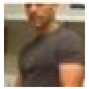
Reactor pattern frameworks do well in the IO performance wars and are typically at the top of the **Techempower** benchmarks



Our experience with Async, Reactor Pattern, and Actors

- Worked with **Actor system** / high-speed messaging / PoCs (2011)
- Used **Vert.x** to handle large amount of traffic on less resources (2012)
- Wrote **QBit** to batch stream service calls to optimize thread-hand off and IO throughput (2013)
- Needed high-speed call coordination for **OAuth rate limiter** (2014)
  - fronting many backend services
  - Worked on QBit Reactor but interface was better than before but still too complicated
- Worked on many microservices in 12 factor cloud env - lots of async call coordination and circuit breaker - lots of retries (2015)
- Worked on Vert.x / QBit project / Node.js project (2016)
  - Started using Node.js / JavaScript promises for client libs
  - Nice abstraction for dealing with async service calls
  - JS Promises were just right





Why async, responsive and reactive?

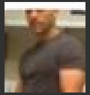
Reactive Manifesto - what is it and why it matters

Reactor Pattern - Most common form of reactive programming, adopts Promises

Microservices

- Async programming prefered for resiliency
- Avoiding cascading failures (“**synchronous calls considered harmful**”)

Async programming is key: How do you manages async call coordination?



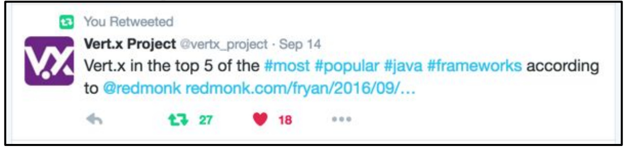
Why the Semantics of ES6 Promises?

Reactor Pattern

Reactor pattern:

- event-driven,
- handlers register for events,
- events can come from multiple sources,
- single-threaded system - for handling events
- handles multiple event loops
- Can aggregate events from other IO threads

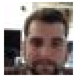
What is the most popular **Reactor Pattern ecosystem**? **Oldest**? **Most common**? Who has to live, breathe and sleep async calls? What do they use? Most popular Java one?


You Retweeted

Vert.x Project @vertx_project · Sep 14

Vert.x in the top 5 of the #most #popular #java #frameworks according to @redmonk redmonk.com/fryan/2016/09/…

27 18



Promise from most popular Reactor system

AHA : This looks nice and makes sense

JS client side most popular reactor pattern of all time

What does JavaScript use to simplify async callback coordination?

Promises!

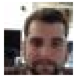
Node.js

Most popular server-side reactor pattern, and growing

What does JavaScript use to simplify async callback coordination?

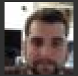
Promises!

Reakt was born!



AHA Moment!

- Wrote client libs in Java with Reakt Promises and ES6 (JavaScript) with/Promises
- Code looked very similar
- Semantics were same
- Slight syntactic differences
- “Wow!” That is clean
- Hard to look at old way



Reakt Details

Reakt Concepts

Promise: Handler for registering events from an async call

Callback: Handler for resolving responses to an async call (scalar async result) / **Mostly internal**

Stream: Like a Callback but has N numbers or results (stream of results)

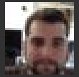
Breaker: Async circuit breakers

Expected: Results that could be missing

Reactor: Replays callbacks on Actor, Verticle or event handler thread (event loop)

repeating tasks, delay tasks

Result: Async result, success, failure, result from a Callback or Stream



Promise

Promise Concepts

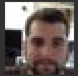
Like ES6 **promises** can be:

Completed States:

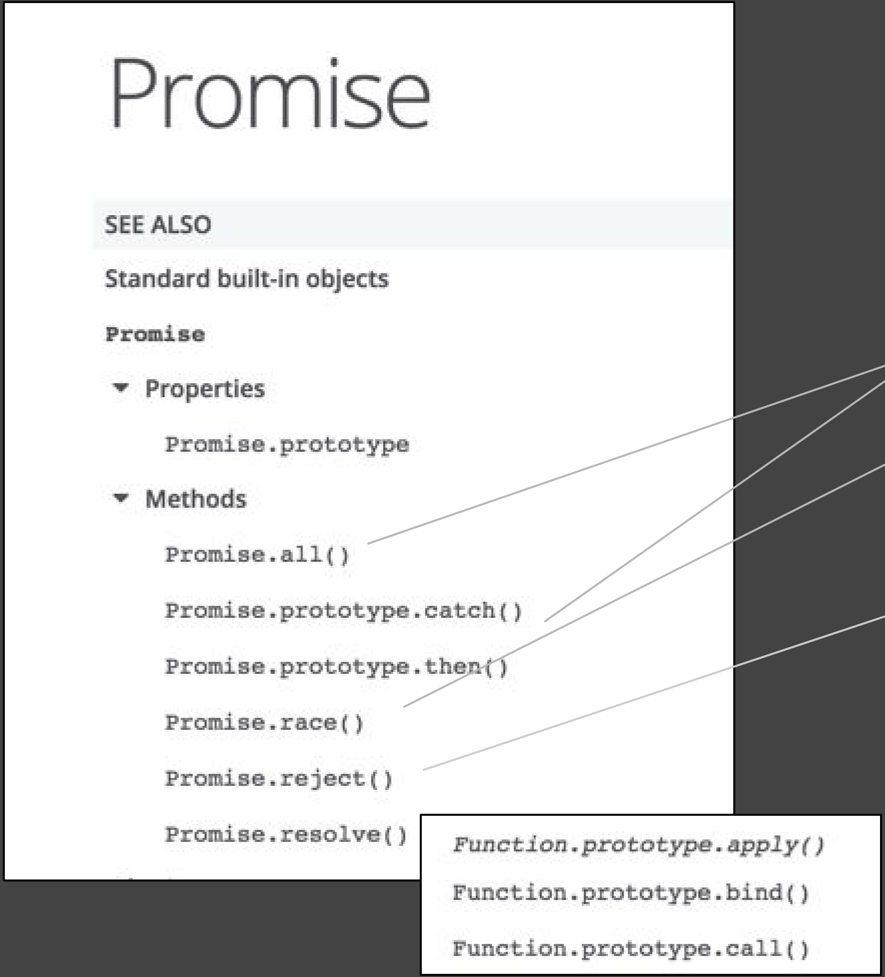
- **Resolved**: callback/action relating to the promise succeeded
- **Rejected**: callback/action relating to the promise failed

When a promise has been **resolved** or **rejected** it is marked **completed**

- **Completed**: callback/action has been fulfilled/resolved or rejected



ES6 Promise vs Reakt



Promise

SEE ALSO

Standard built-in objects

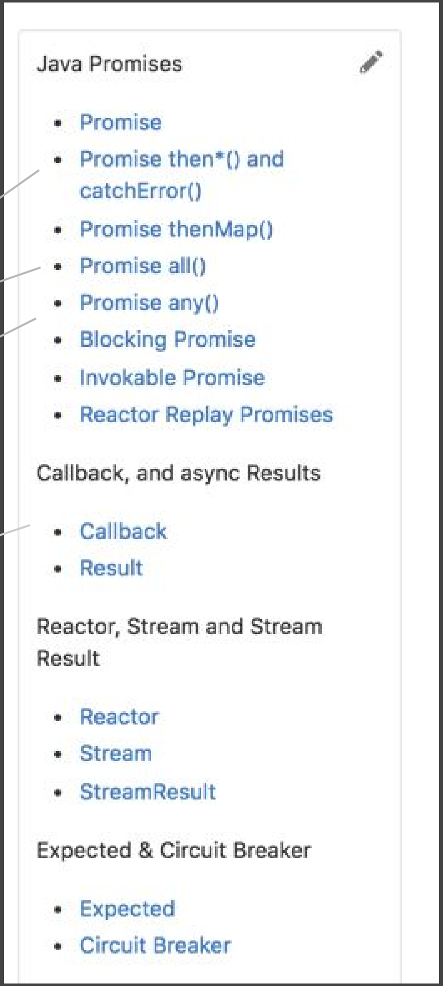
`Promise`

- Properties
  - `Promise.prototype`
- Methods
  - `Promise.all()`
  - `Promise.prototype.catch()`
  - `Promise.prototype.then()`
  - `Promise.race()`
  - `Promise.reject()`
  - `Promise.resolve()`

`Function.prototype.apply()`

`Function.prototype.bind()`

`Function.prototype.call()`



Java Promises

- Promise
- Promise then*() and catchError()
- Promise thenMap()
- Promise all()
- Promise any()
- Blocking Promise
- Invokable Promise
- Reactor Replay Promises

Callback, and async Results

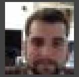
- Callback
- Result

Reactor, Stream and Stream Result

- Reactor
- Stream
- StreamResult

Expected & Circuit Breaker

- Expected
- Circuit Breaker



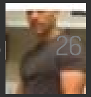
Special concerns with Java MT

JavaScript is **single-threaded** - Java is not.

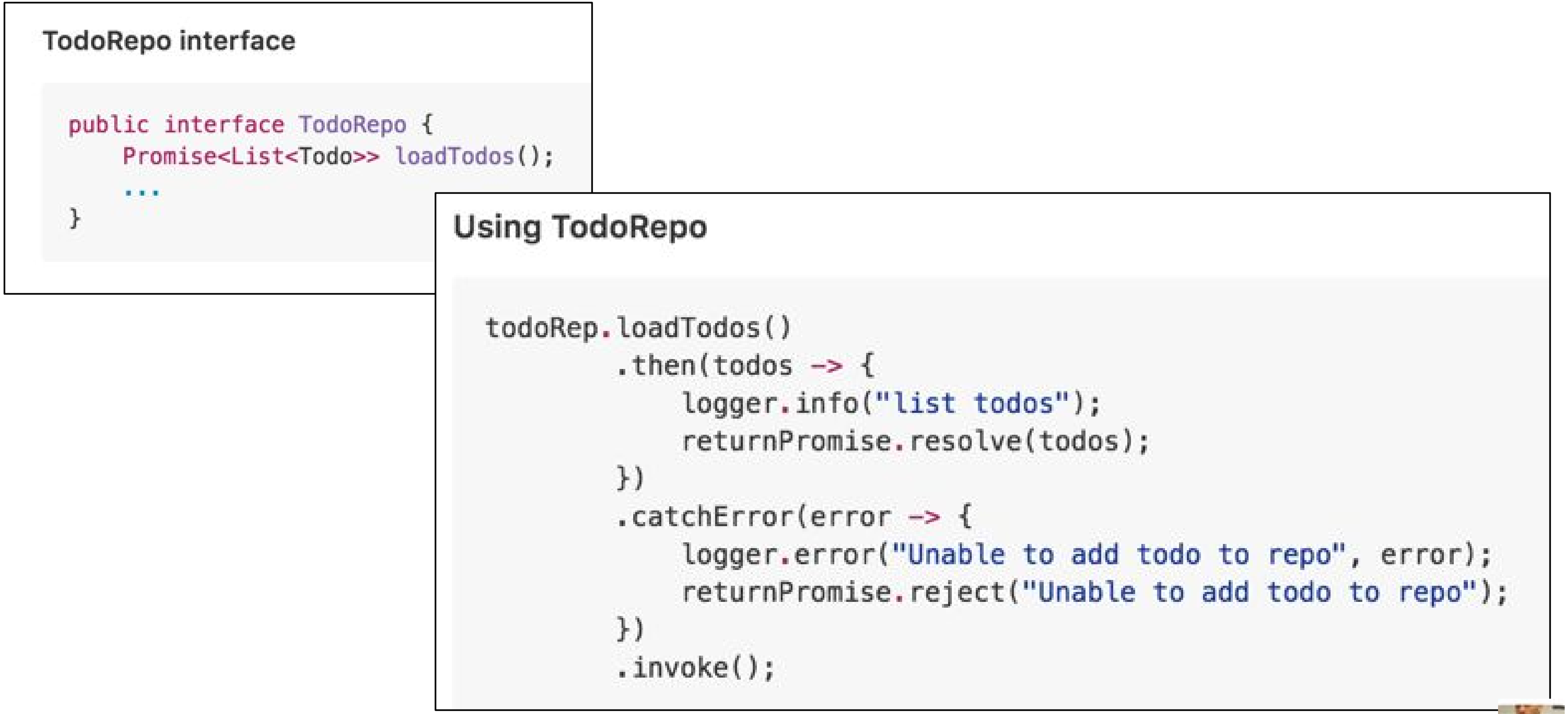
Three types of Reakt promises:

- Callback promises (async) (**Promise**)
- Blocking promises (for unit testing and legacy integration) (**BlockingPromise**)
- Replay promises (**ReplayPromise**)
  - allow promises to be handled on the same thread as caller
  - Works with Vert.x verticles, QBit service actors, other actors and even bus reactor (Netty)
  - Replay promises can have timeouts

Promises in Reakt can be **invokeable** which allow for a fluent registry of handlers before invocatio



Using Promise

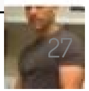


TodoRepo interface

```
public interface TodoRepo {
    Promise<List<Todo>> loadTodos();
    ...
}
```

Using TodoRepo

```
todoRep.loadTodos()
        .then(todos -> {
            logger.info("list todos");
            returnPromise.resolve(todos);
        })
        .catchError(error -> {
            logger.error("Unable to add todo to repo", error);
            returnPromise.reject("Unable to add todo to repo");
        })
        .invoke();
```



Handler methods

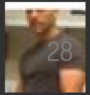
then() - use handle result of async calls

thenExpected() - use handle async calls whose result could be null

thenSafe() - like **then** but handles exception from handler

thenSafeExpected() - same as **thenSafe** but result could be null

catchError() - handles an exception

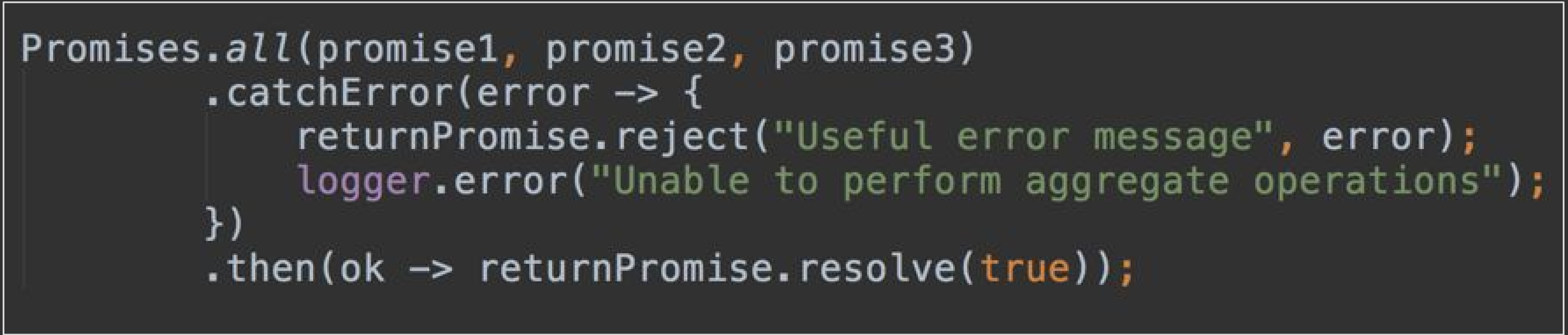


Promises.all

Promises.all(promises)

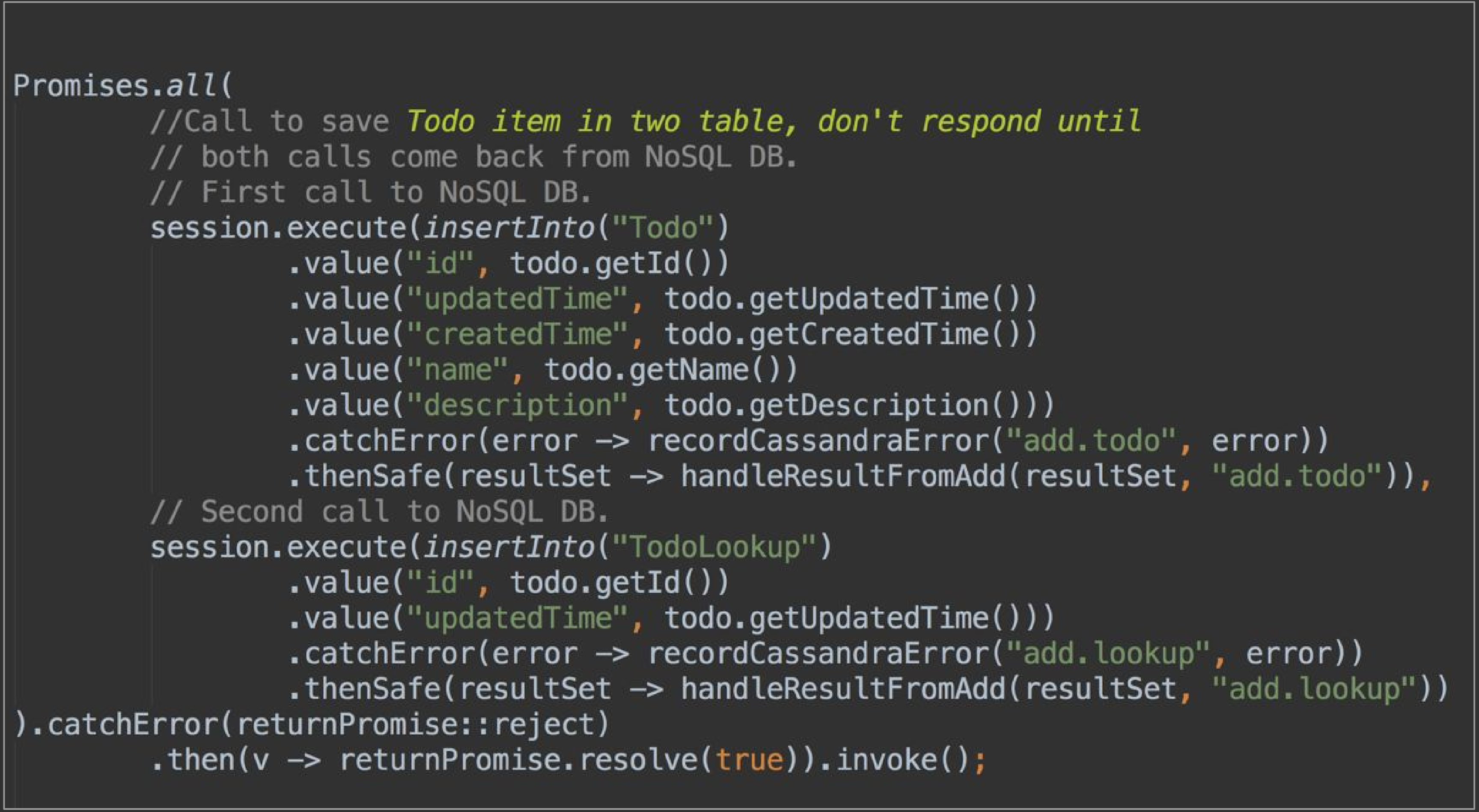
You create a promise whose **then** or **catchError** trigger (it resolves) when all promises passed async return (all resolve)

If any promise fails, the **all** promise fails

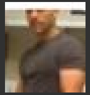


```
Promises.all(promise1, promise2, promise3)
        .catchError(error -> {
            returnPromise.reject("Useful error message", error);
            logger.error("Unable to perform aggregate operations");
        })
        .then(ok -> returnPromise.resolve(true));
```

Promises.all



```
Promises.all(
        //Call to save Todo item in two table, don't respond until
        // both calls come back from NoSQL DB.
        // First call to NoSQL DB.
        session.execute(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 NoSQL DB.
        session.execute(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();
```

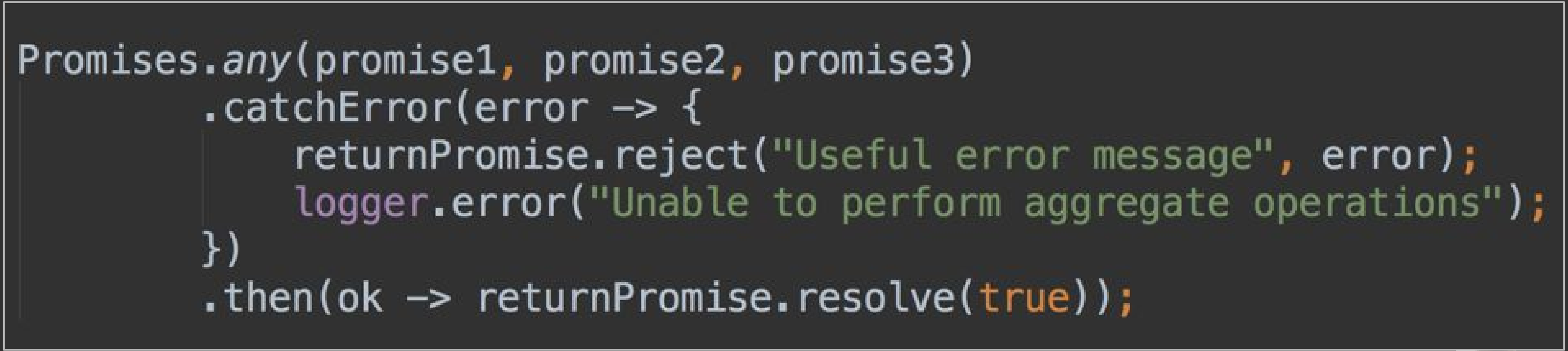


Promises.any

Promise.any(promises)

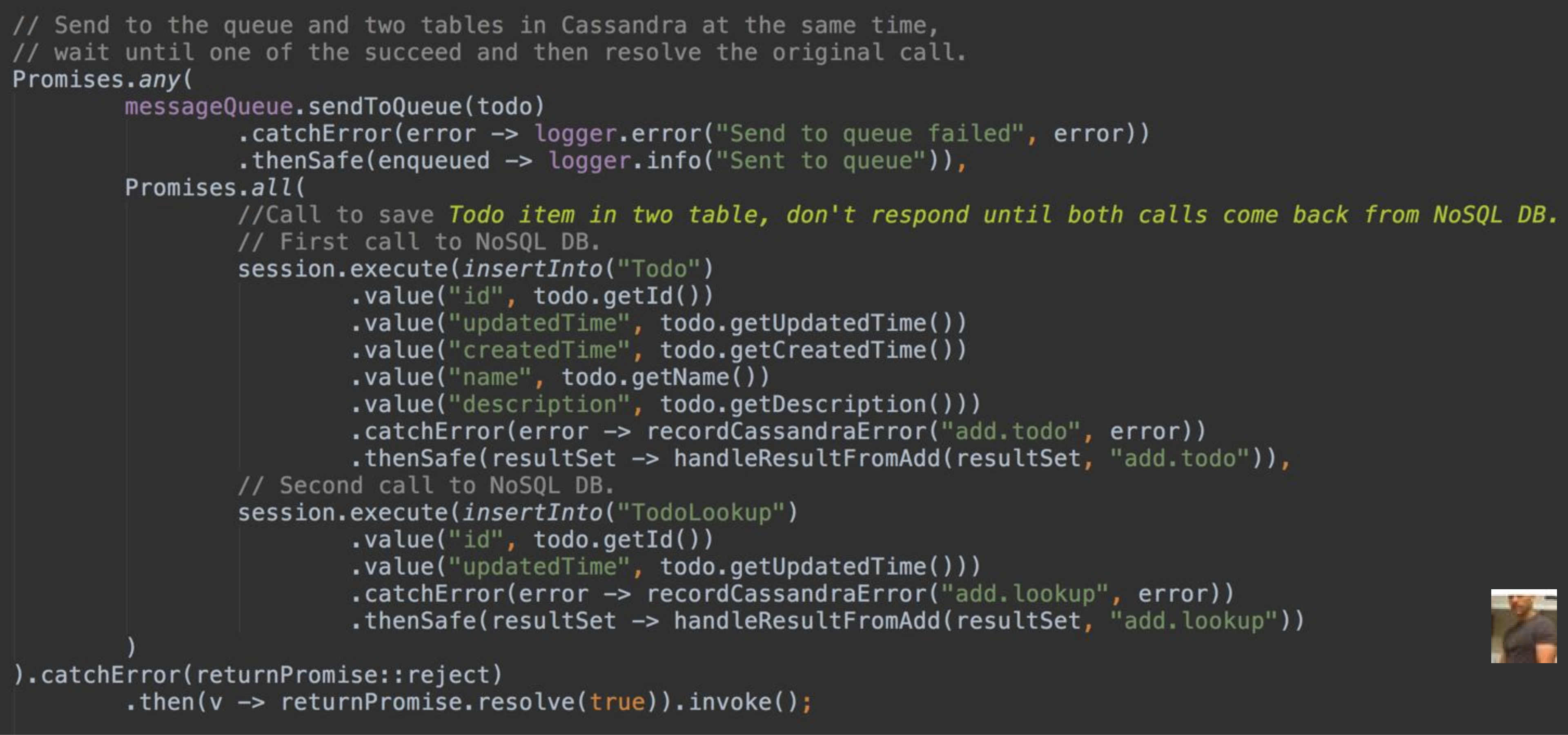
Creates promise that resolves or rejects

as soon as one of the promises **resolves**



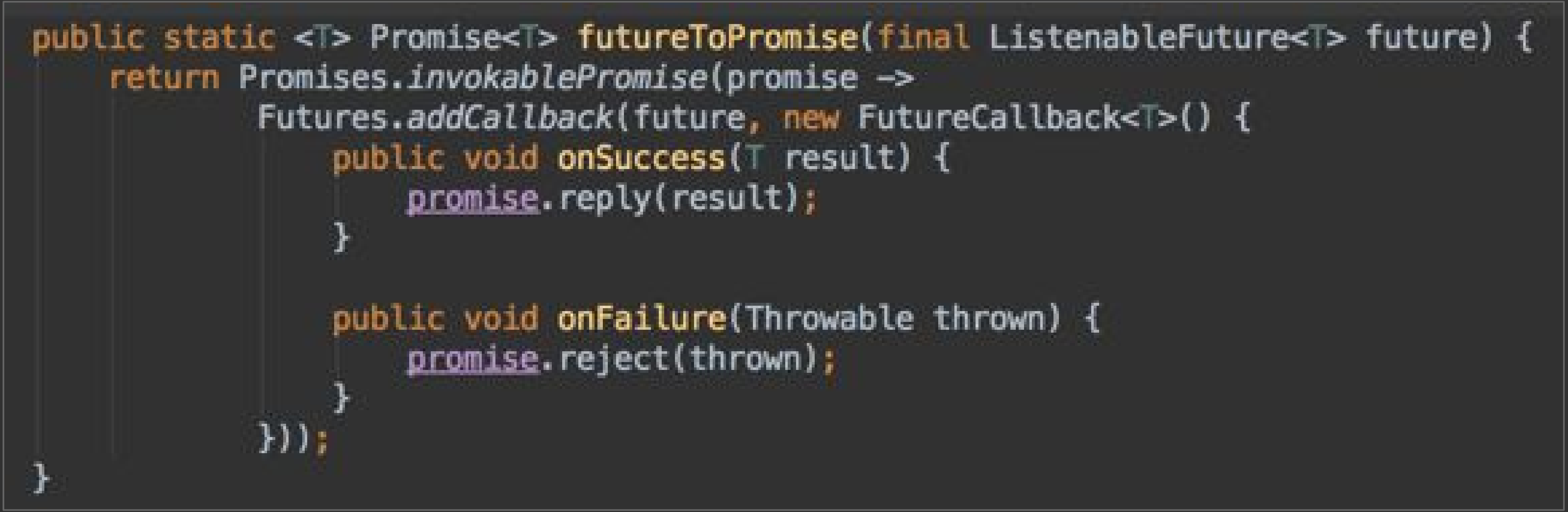
```
Promises.any(promise1, promise2, promise3)
        .catchError(error -> {
            returnPromise.reject("Useful error message", error);
            logger.error("Unable to perform aggregate operations");
        })
        .then(ok -> returnPromise.resolve(true));
```

Promises.any() and Promise.all()

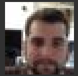


```
// Send to the queue and two tables in Cassandra at the same time,
// wait until one of the succeed and then resolve the original call.
Promises.any(
        messageQueue.sendToQueue(todo)
                .catchError(error -> logger.error("Send to queue failed", error))
                .thenSafe(enqueued -> logger.info("Sent to queue")),
        Promises.all(
                //Call to save Todo item in two table, don't respond until both calls come back from NoSQL DB.
                // First call to NoSQL DB.
                session.execute(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 NoSQL DB.
                session.execute(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();
```

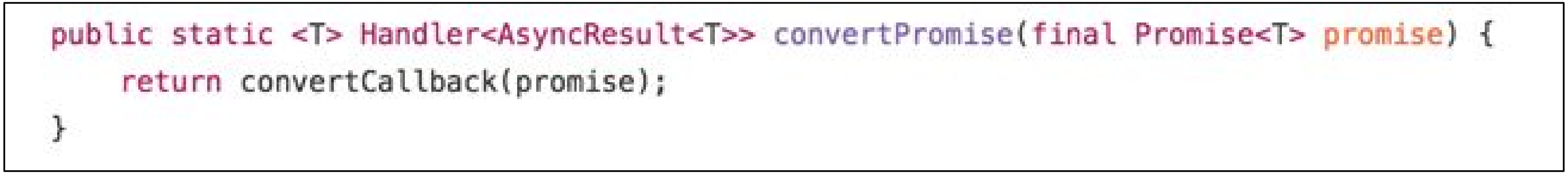
Promises.invokeablePromise (Reakt-Guava)



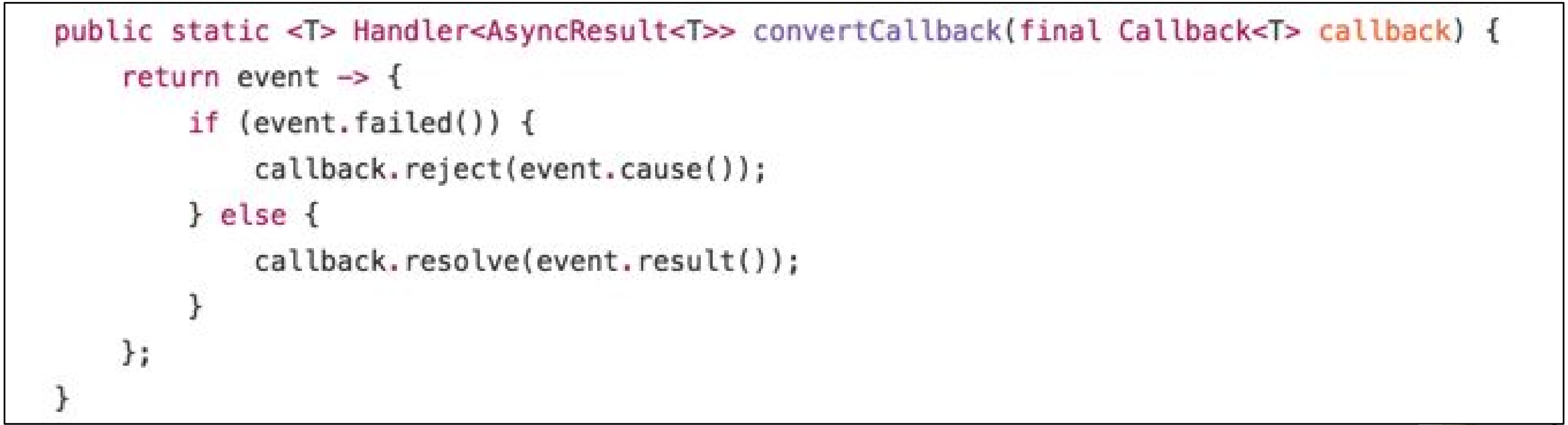
```
public static <T> Promise<T> futureToPromise(final ListenableFuture<T> future) {
    return Promises.invokablePromise(promise ->
            Futures.addCallback(future, new FutureCallback<T>() {
                public void onSuccess(T result) {
                    promise.reply(result);
                }

                public void onFailure(Throwable thrown) {
                    promise.reject(thrown);
                }
            }));
}
```



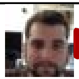
Easy to integrate w/ async libs - reakt-vertx



```
public static <T> Handler<AsyncResult<T>> convertPromise(final Promise<T> promise) {
    return convertCallback(promise);
}
```



```
public static <T> Handler<AsyncResult<T>> convertCallback(final Callback<T> callback) {
    return event -> {
        if (event.failed()) {
            callback.reject(event.cause());
        } else {
            callback.resolve(event.result());
        }
    };
}
```



Reactor

Reactor

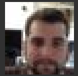
Manages callbacks (**ReplayPromises**) that execute in caller's thread (thread safe, async callbacks)

- Promise handlers that are triggered in caller’s thread
- Timeouts for async calls

Manages tasks that run in the caller's thread

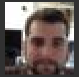
- **Repeating tasks** that run in a caller's thread
- **one shot** timed tasks that run in the caller's thread

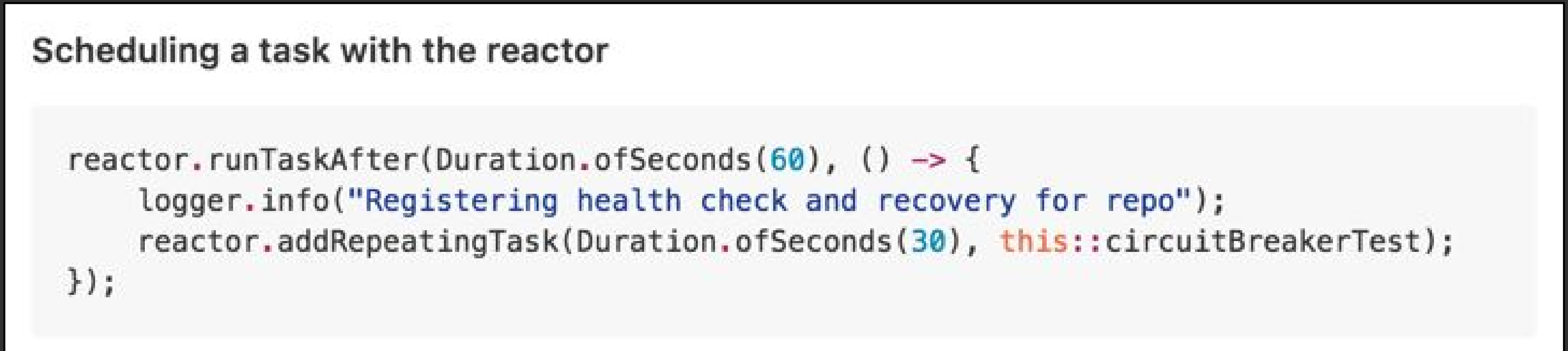
Adapts to event loop, Verticle, Actor



Notable Reactor Methods

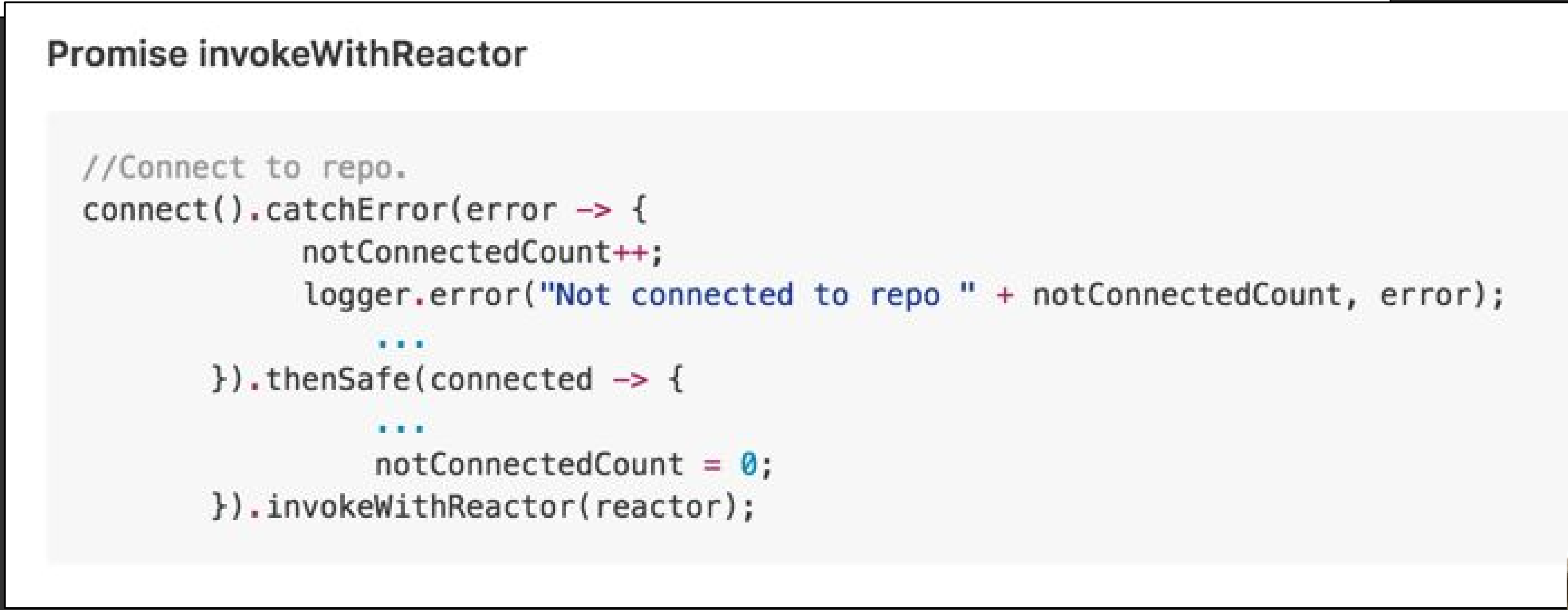
- **addRepeatingTask(interval, runnable)** add a task that repeats every interval
- **runTaskAfter(afterInterval, runnable)** run a task after an interval expires
- **deferRun(runnable)** run a task on this thread as soon as you can
- **all(...)** creates a all promise; resolves with **Reactor** (you can pass a timeout)
- **any(...)** create any promise with **Reactor** (you can pass a timeout)
- **promise()** creates a **ReplayPromise** so Reactor manages promise (you can pass a timeout)





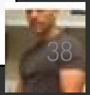
Scheduling a task with the reactor

```
reactor.runTaskAfter(Duration.ofSeconds(60), () -> {
    logger.info("Registering health check and recovery for repo");
    reactor.addRepeatingTask(Duration.ofSeconds(30), this::circuitBreakerTest);
});
```

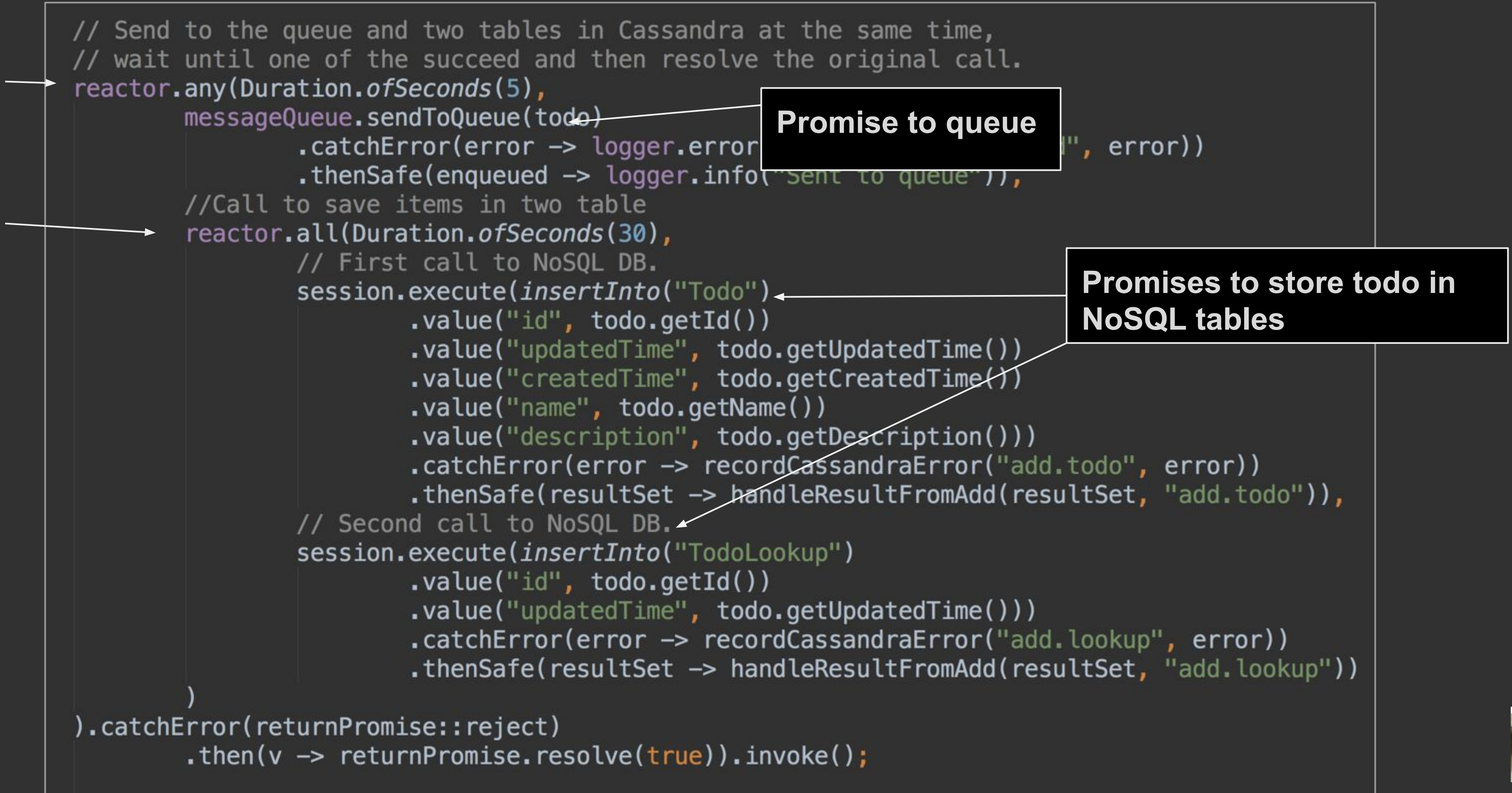


Promise invokeWithReactor

```
//Connect to repo.
connect().catchError(error -> {
            notConnectedCount++;
            logger.error("Not connected to repo " + notConnectedCount, error);
                ...
       }).thenSafe(connected -> {
                ...
                notConnectedCount = 0;
       }).invokeWithReactor(reactor);
```



Reactor, any, all, timeouts



```
// Send to the queue and two tables in Cassandra at the same time,
// wait until one of the succeed and then resolve the original call.
reactor.any(Duration.ofSeconds(5),
        messageQueue.sendToQueue(todo)
                .catchError(error -> logger.error                    ", error))
                .thenSafe(enqueued -> logger.info("Sent to queue")),
        //Call to save items in two table
        reactor.all(Duration.ofSeconds(30),
                // First call to NoSQL DB.
                session.execute(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 NoSQL DB.
                session.execute(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();
```

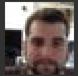
Promise to queue

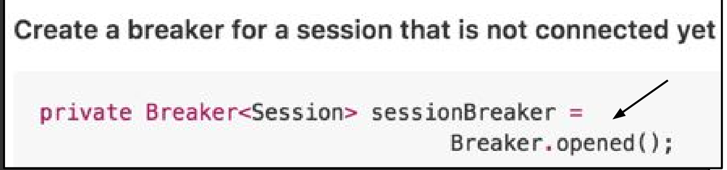
Promises to store todo in NoSQL tables

Circuit breaker

Circuit Breaker

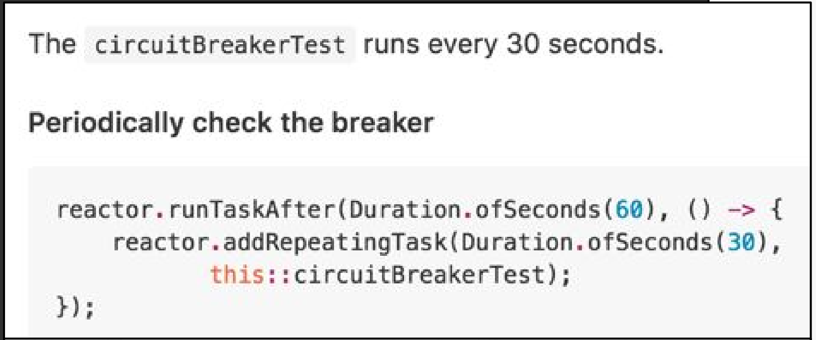
- **Breaker** is short for circuit breaker
- Wraps access to a service, resource, repo, connection, queue, etc.
- Tracks errors which can trigger the **breaker** to **open**
- **Breaker** is just an interface / contract
  - Implementers can be creative in what is considered an open or broken breaker





Create a breaker for a session that is not connected yet

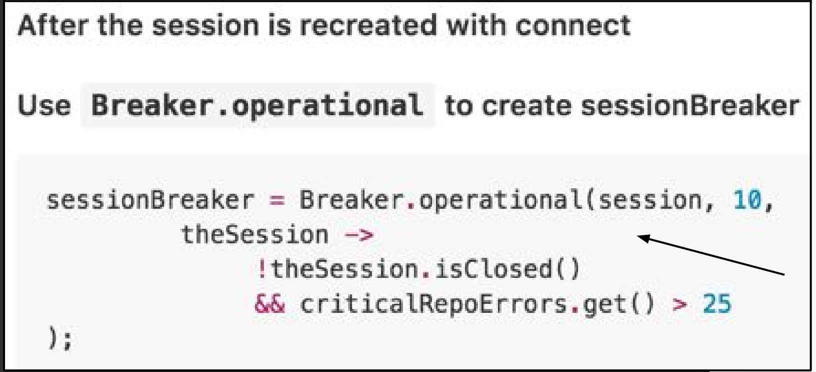
```java
private Breaker<Session> sessionBreaker =
                                   Breaker.opened();
```



The `circuitBreakerTest` runs every 30 seconds.

Periodically check the breaker

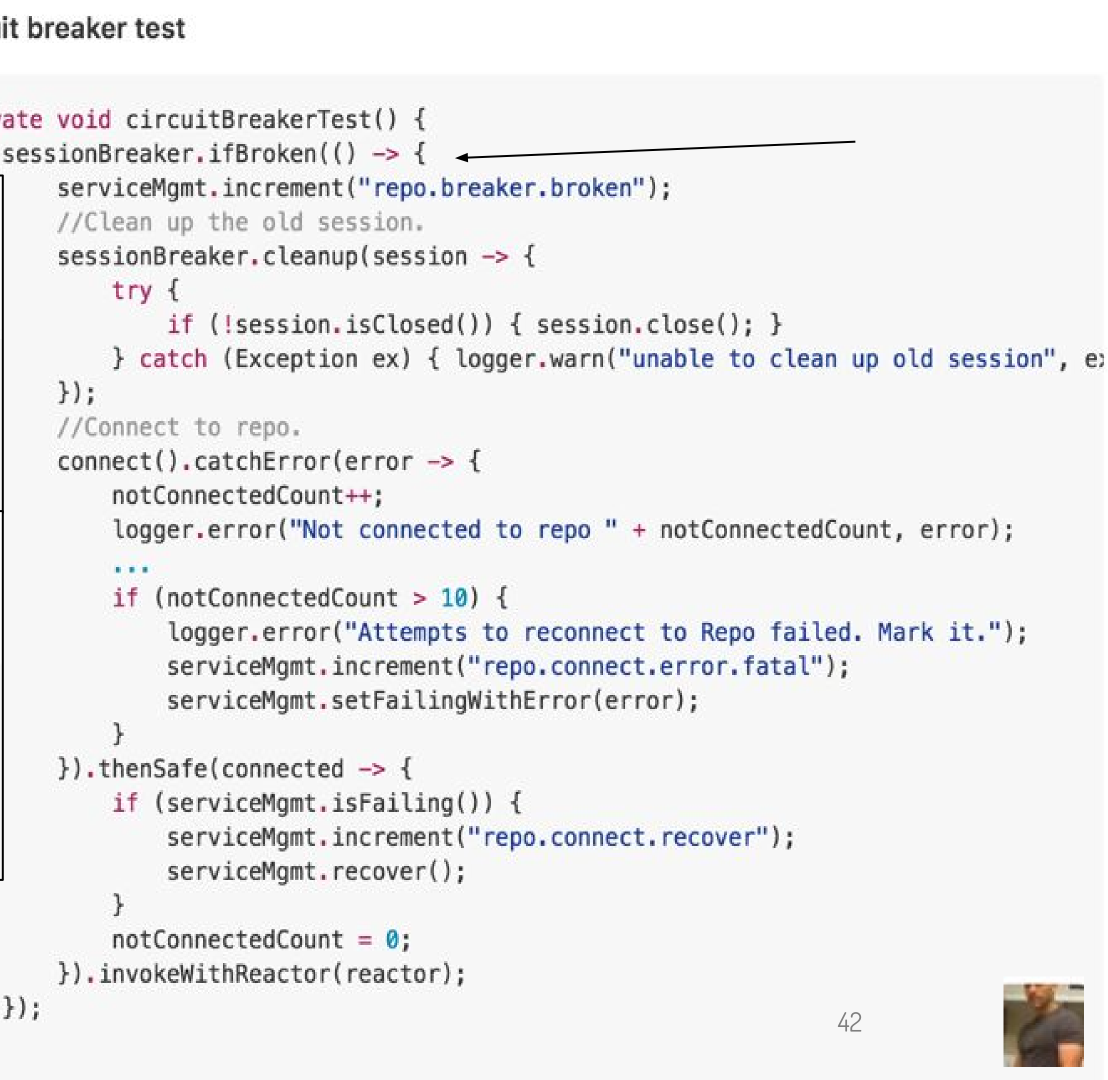
```java
reactor.runTaskAfter(Duration.ofSeconds(60), () -> {
    reactor.addRepeatingTask(Duration.ofSeconds(30),
          this::circuitBreakerTest);
});
```



After the session is recreated with connect

Use `Breaker.operational` to create sessionBreaker

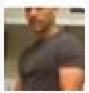
```java
sessionBreaker = Breaker.operational(session, 10,
         theSession ->
              !theSession.isClosed()
              && criticalRepoErrors.get() > 25
);
```



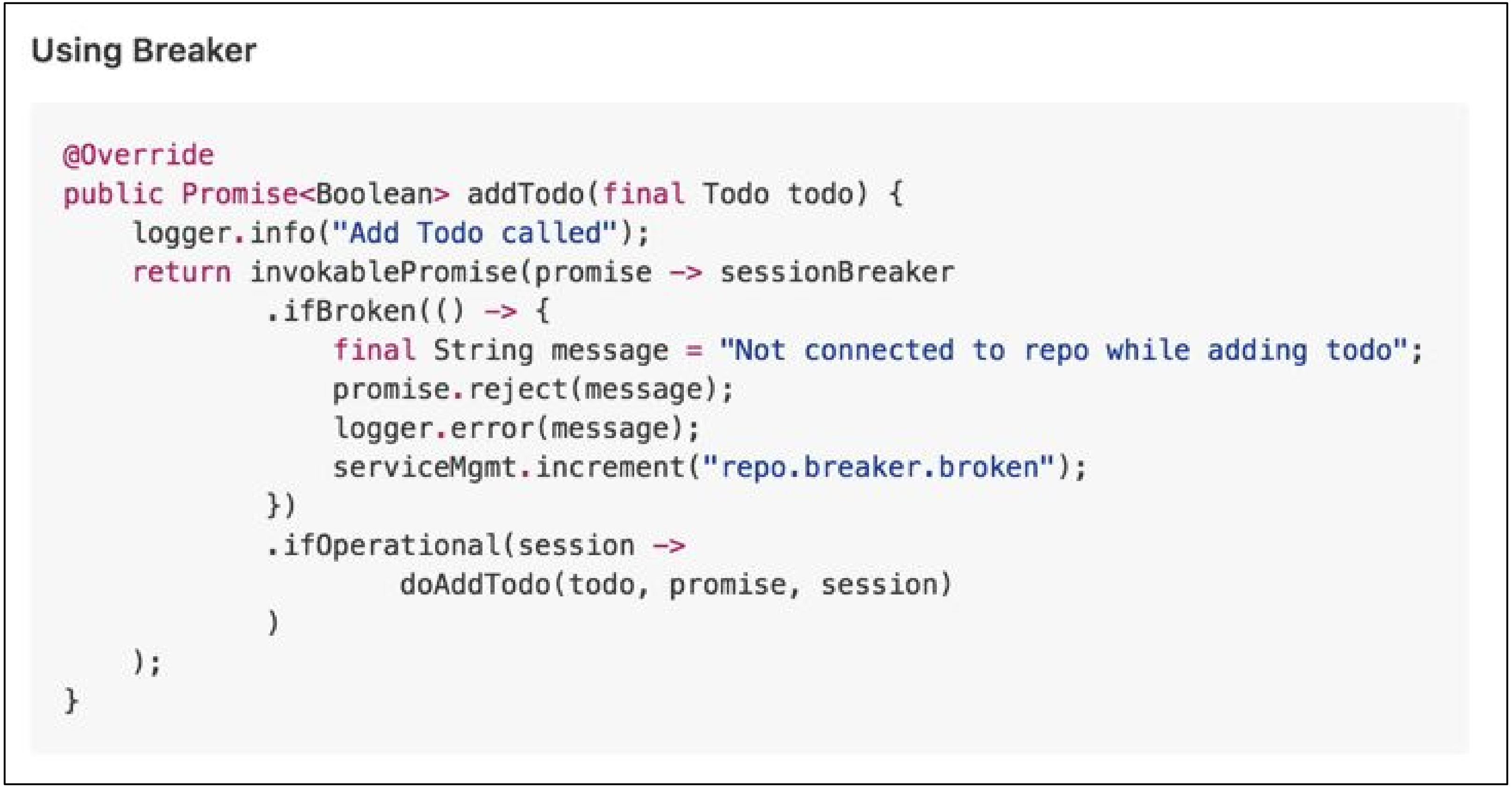
Circuit breaker test

```java
private void circuitBreakerTest() {
    sessionBreaker.ifBroken(() -> {
        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", e
        });
        //Connect to repo.
        connect().catchError(error -> {
            notConnectedCount++;
            logger.error("Not connected to repo " + notConnectedCount, error);
            ...
            if (notConnectedCount > 10) {
                logger.error("Attempts to reconnect to Repo failed. Mark it.");
                serviceMgmt.increment("repo.connect.error.fatal");
                serviceMgmt.setFailingWithError(error);
            }
        }).thenSafe(connected -> {
            if (serviceMgmt.isFailing()) {
                serviceMgmt.increment("repo.connect.recover");
                serviceMgmt.recover();
            }
            notConnectedCount = 0;
        }).invokeWithReactor(reactor);
    });
}
```

Come to the lab tomorrow to
See a full use case using
Async circuit breakers

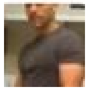


Using breaker



Using Breaker

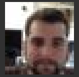
```
@Override
public Promise<Boolean> addTodo(final Todo todo) {
    logger.info("Add Todo called");
    return invokablePromise(promise -> sessionBreaker
            .ifBroken(() -> {
                final String message = "Not connected to repo while adding todo";
                promise.reject(message);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



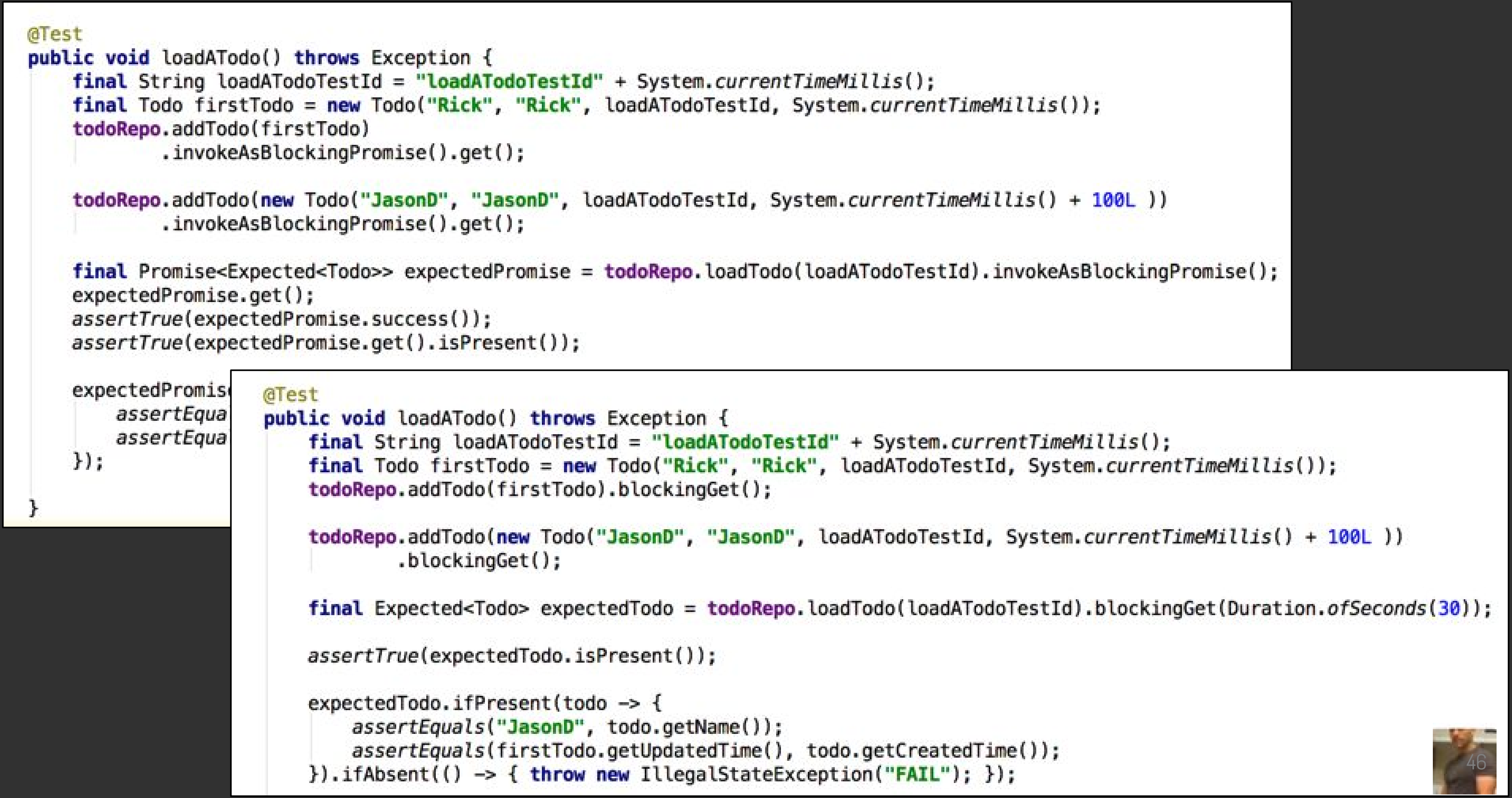
Blocking promises

Blocking promises

- Legacy integration
- Unit testing
- Prototypes
- Batch jobs that don’t need async
    - (so you don’t have to have two libs)



Unit test using blocking promise using invokeAsBlockingPromise then blockingGet



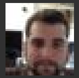
```java
@Test
public void loadATodo() throws Exception {
    final String loadATodoTestId = "loadATodoTestId" + System.currentTimeMillis();
    final Todo firstTodo = new Todo("Rick", "Rick", loadATodoTestId, System.currentTimeMillis());
    todoRepo.addTodo(firstTodo)
            .invokeAsBlockingPromise().get();

    todoRepo.addTodo(new Todo("JasonD", "JasonD", loadATodoTestId, System.currentTimeMillis() + 100L ))
            .invokeAsBlockingPromise().get();

    final Promise<Expected<Todo>> expectedPromise = todoRepo.loadTodo(loadATodoTestId).invokeAsBlockingPromise();
    expectedPromise.get();
    assertTrue(expectedPromise.success());
    assertTrue(expectedPromise.get().isPresent());

    expectedPromis
        assertEqua
        assertEqua
    });

}
```

```java
@Test
public void loadATodo() throws Exception {
    final String loadATodoTestId = "loadATodoTestId" + System.currentTimeMillis();
    final Todo firstTodo = new Todo("Rick", "Rick", loadATodoTestId, System.currentTimeMillis());
    todoRepo.addTodo(firstTodo).blockingGet();

    todoRepo.addTodo(new Todo("JasonD", "JasonD", loadATodoTestId, System.currentTimeMillis() + 100L ))
            .blockingGet();

    final Expected<Todo> expectedTodo = todoRepo.loadTodo(loadATodoTestId).blockingGet(Duration.ofSeconds(30));

    assertTrue(expectedTodo.isPresent());

    expectedTodo.ifPresent(todo -> {
        assertEquals("JasonD", todo.getName());
        assertEquals(firstTodo.getUpdatedTime(), todo.getCreatedTime());
    }).ifAbsent(() -> { throw new IllegalStateException("FAIL"); });
```

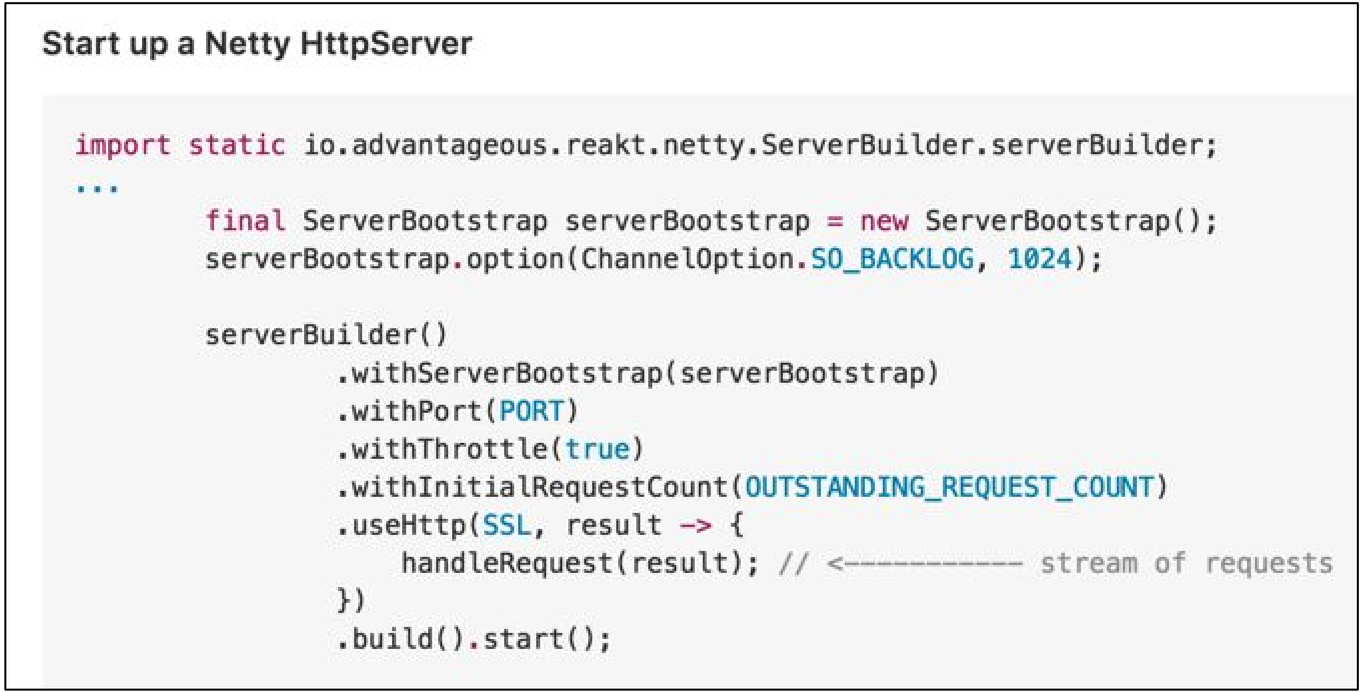
Stream

Stream

- Handler for N results
- While a **Callback** and **Promise** is for one **Result**, a **Stream** is for N results
- **Callback/Promise** for **Scalar** returns
- **Stream** is for many returns
- Similar to Java 9 Flow, RxJava or Reactive Streams
- Java 8/9 **lambda expression** friendly
- (Fuller example as extra material on slide deck depending on time go to end of slide deck or just cover next two slides)
- StreamResult, **then**(), **catchError**(), **cancel**(), **request**(count)

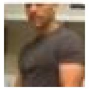


streamResult.cancel(), streamResult.request(count)

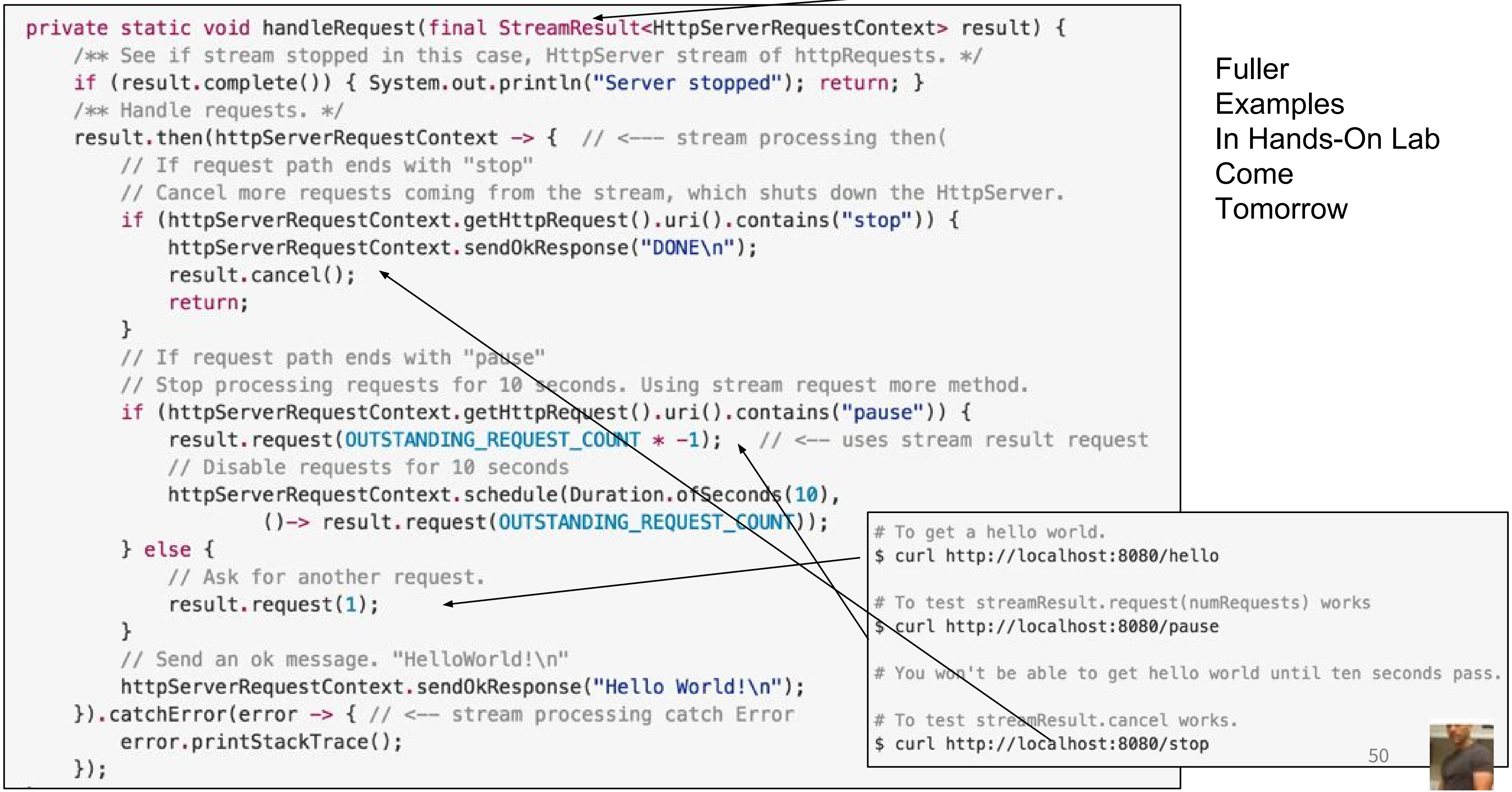


Start up a Netty HttpServer

```
import static io.advantageous.reakt.netty.ServerBuilder.serverBuilder;
...
        final ServerBootstrap serverBootstrap = new ServerBootstrap();
        serverBootstrap.option(ChannelOption.SO_BACKLOG, 1024);

        serverBuilder()
                .withServerBootstrap(serverBootstrap)
                .withPort(PORT)
                .withThrottle(true)
                .withInitialRequestCount(OUTSTANDING_REQUEST_COUNT)
                .useHttp(SSL, result -> {
                    handleRequest(result); // <----------- stream of requests
                })
                .build().start();
```



streamResult.cancel(), streamResult.request(count) (part 2)

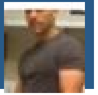


```java
private static void handleRequest(final StreamResult<HttpServerRequestContext> result) {
    /** See if stream stopped in this case, HttpServer stream of httpRequests. */
    if (result.complete()) { System.out.println("Server stopped"); return; }
    /** Handle requests. */
    result.then(httpServerRequestContext -> {  // <--- stream processing then(
        // If request path ends with "stop"
        // Cancel more requests coming from the stream, which shuts down the HttpServer.
        if (httpServerRequestContext.getHttpRequest().uri().contains("stop")) {
            httpServerRequestContext.sendOkResponse("DONE\n");
            result.cancel();
            return;
        }
        // If request path ends with "pause"
        // Stop processing requests for 10 seconds. Using stream request more method.
        if (httpServerRequestContext.getHttpRequest().uri().contains("pause")) {
            result.request(OUTSTANDING_REQUEST_COUNT * -1);   // <-- uses stream result request
            // Disable requests for 10 seconds
            httpServerRequestContext.schedule(Duration.ofSeconds(10),
                    ()-> result.request(OUTSTANDING_REQUEST_COUNT));
        } else {
            // Ask for another request.
            result.request(1);
        }
        // Send an ok message. "HelloWorld!\n"
        httpServerRequestContext.sendOkResponse("Hello World!\n");
    }).catchError(error -> { // <-- stream processing catch Error
        error.printStackTrace();
    });
```

```
# To get a hello world.
$ curl http://localhost:8080/hello

# To test streamResult.request(numRequests) works
$ curl http://localhost:8080/pause

# You won't be able to get hello world until ten seconds pass.

# To test streamResult.cancel works.
$ curl http://localhost:8080/stop
```

Fuller
Examples
In Hands-On Lab
Come
Tomorrow

Example that combines Reakt: Reactor/Stream/Promises



Example Recommendation Service

- Recommendation service
- Watch what user does, then suggest recommended items
- Recommendation service runs many recommendation engines per microservice

Worked example will show

User recommendation service

Delay giving recommendations to a user until that user is loaded from a backend service store

Users are streamed in (uses streams)

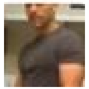
Stream comes in on foreign thread and we use reactor to move handler to service actor thread

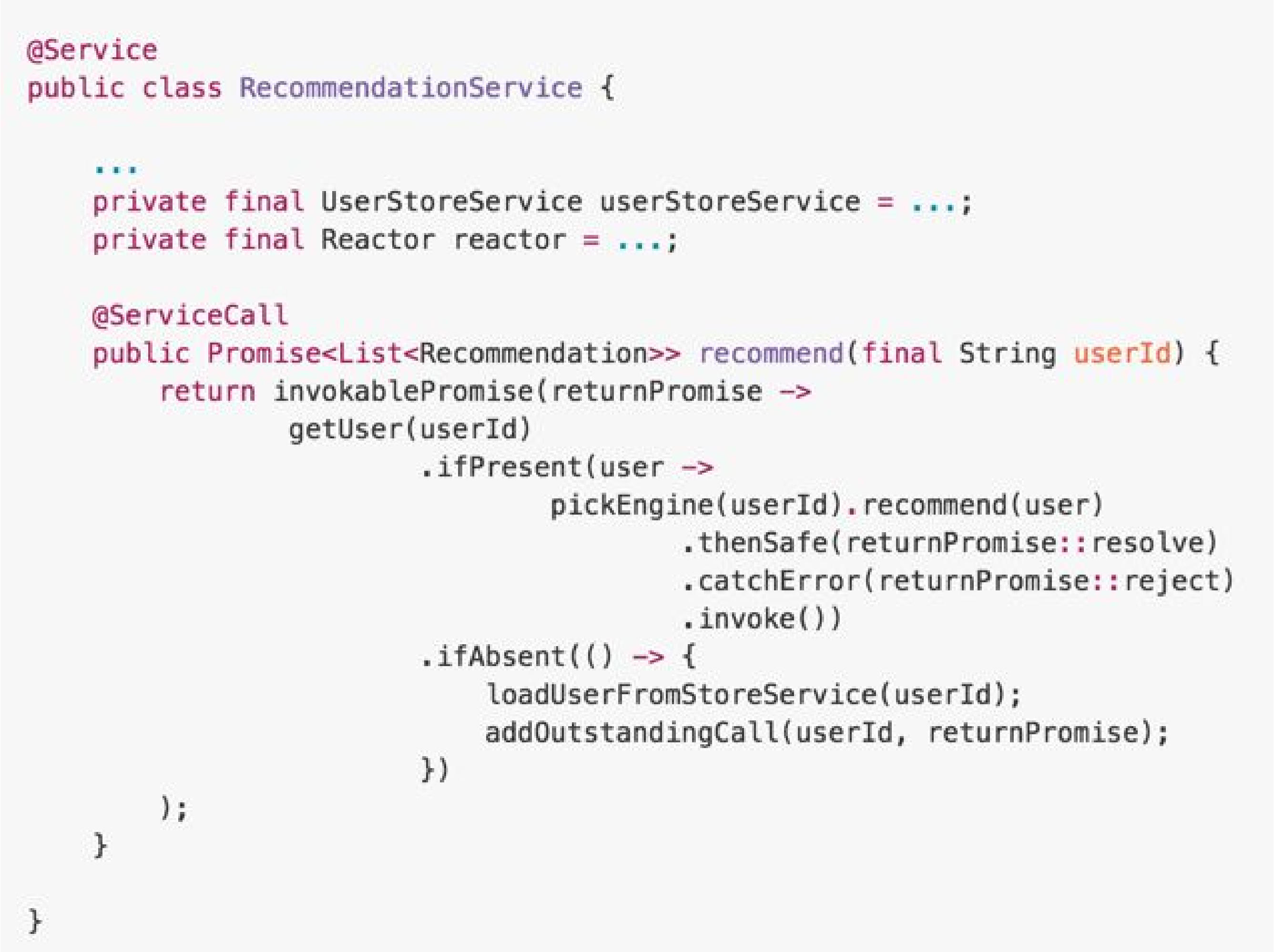
If user is already in service actor, then recommend a list of recommendations right away

If user not in system, batch load user from backend service store

Requests are batched to reduce IO overhead

Users can come from many sources from service store (cache, disk cache, DB), and are delivered as soon as found in a continuous stream of user lists

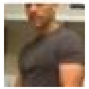


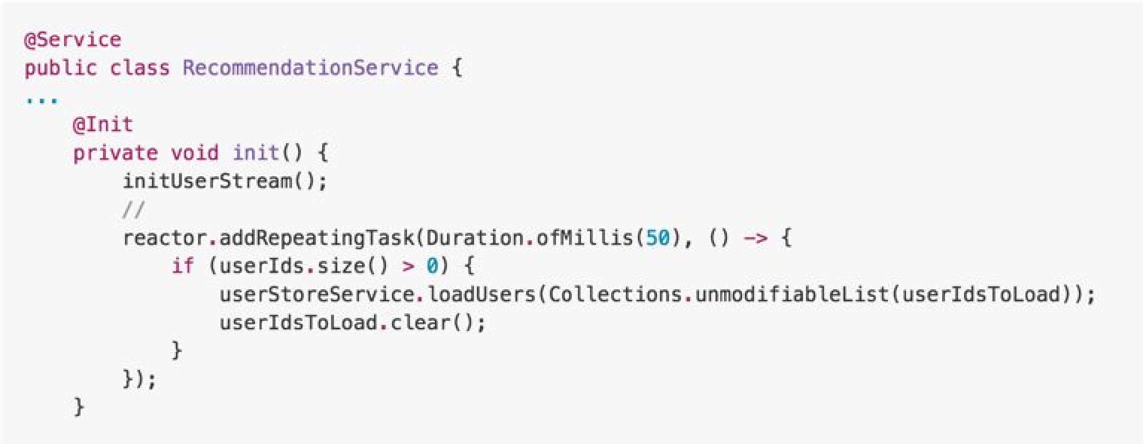


```
@Service
public class RecommendationService {

    ...
    private final UserStoreService userStoreService = ...;
    private final Reactor reactor = ...;

    @ServiceCall
    public Promise<List<Recommendation>> recommend(final String userId) {
        return invokablePromise(returnPromise ->
                getUser(userId)
                        .ifPresent(user ->
                                pickEngine(userId).recommend(user)
                                        .thenSafe(returnPromise::resolve)
                                        .catchError(returnPromise::reject)
                                        .invoke())
                        .ifAbsent(() -> {
                            loadUserFromStoreService(userId);
                            addOutstandingCall(userId, returnPromise);
                        })
        );
    }

}
```

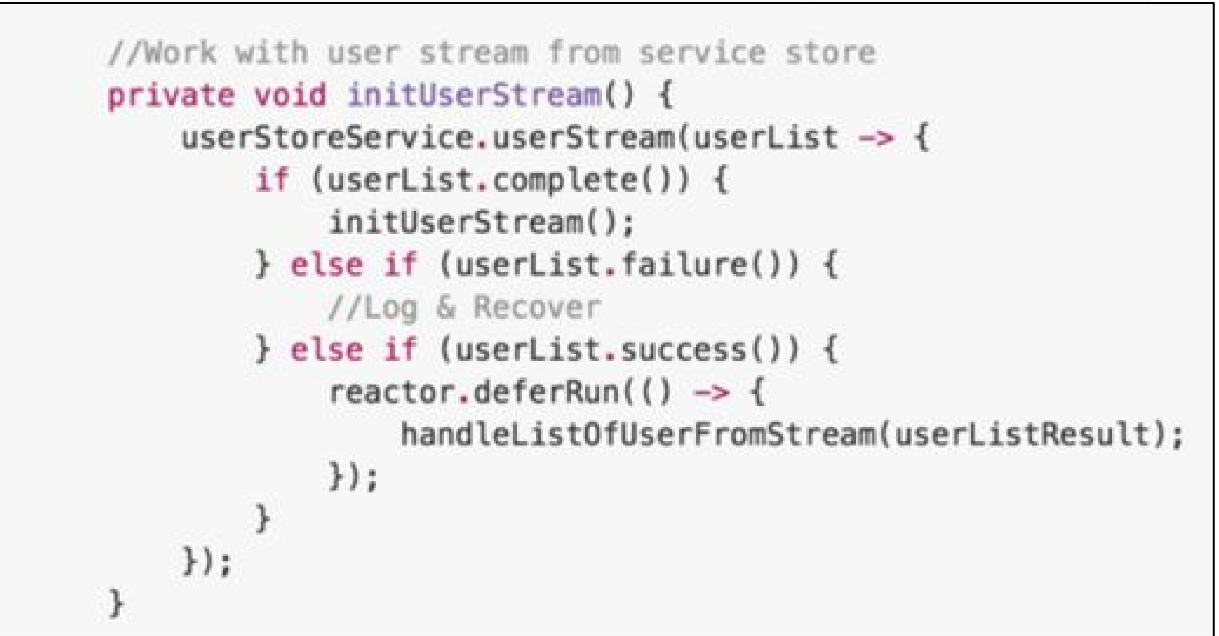
Async call recommendation engine if user present

If not add user id to call batch, add to outstanding call for user.





```
@Service
public class RecommendationService {
...
    @Init
    private void init() {
        initUserStream();
        //
        reactor.addRepeatingTask(Duration.ofMillis(50), () -> {
            if (userIds.size() > 0) {
                userStoreService.loadUsers(Collections.unmodifiableList(userIdsToLoad));
                userIdsToLoad.clear();
            }
        });
    }
```

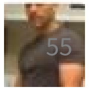


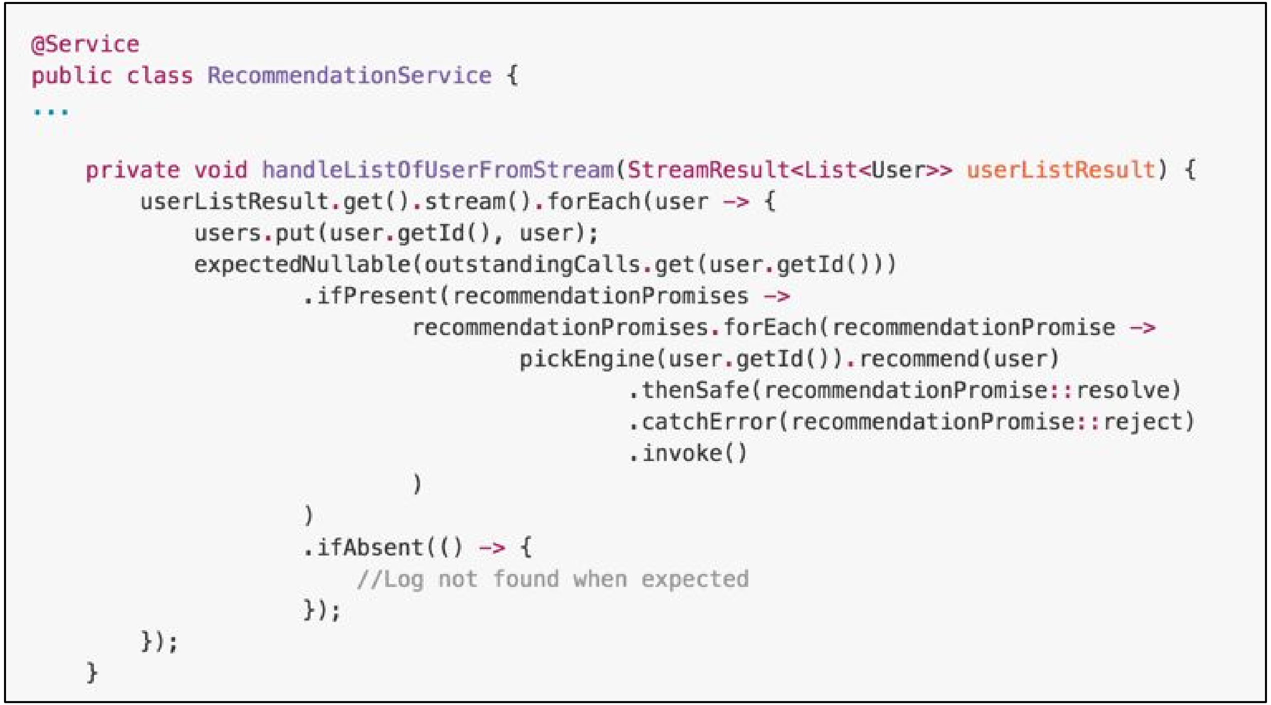
```
//Work with user stream from service store
private void initUserStream() {
    userStoreService.userStream(userList -> {
        if (userList.complete()) {
            initUserStream();
        } else if (userList.failure()) {
            //Log & Recover
        } else if (userList.success()) {
            reactor.deferRun(() -> {
                handleListOfUserFromStream(userListResult);
            });
        }
    });
}
```

Every 50 ms check to see if the userIdsToLoad is greater than 0, If so request those users now.

When a user is not found loadUserFromStoreService is called. If there are 100, outstanding requests, then load those users now.

Listen to the userStoreService’s userStream



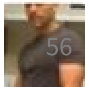


```
@Service
public class RecommendationService {
...

    private void handleListOfUserFromStream(StreamResult<List<User>> userListResult) {
        userListResult.get().stream().forEach(user -> {
            users.put(user.getId(), user);
            expectedNullable(outstandingCalls.get(user.getId()))
                    .ifPresent(recommendationPromises ->
                            recommendationPromises.forEach(recommendationPromise ->
                                    pickEngine(user.getId()).recommend(user)
                                            .thenSafe(recommendationPromise::resolve)
                                            .catchError(recommendationPromise::reject)
                                            .invoke()
                            )
                    )
                    .ifAbsent(() -> {
                        //Log not found when expected
                    });
        });
    }
```

Process the stream result.

Populate the user map (or use a Simple cache with an expiry and a max number of users allowed to be in system).

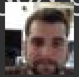
Since the user is now loaded, see if their are outstanding calls (promises) and resolve those calls.

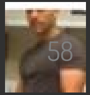


Next steps

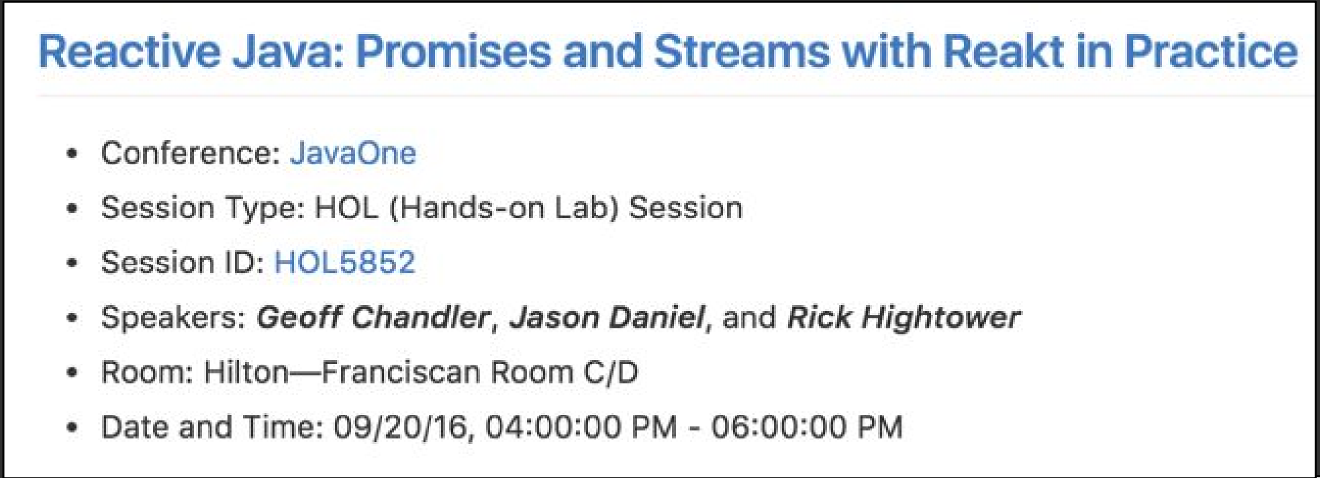
Next steps

1) Get rid of invoke and detect when a frame drops (let Geoff explain this one)
2) Simplify interface for Promise/Callback Reakt 4.0
   a) We use semantic versioning, but even from version to version so far interfaces are fairly compatible for 97% of use cases
3) More reakt libs ***
4) Refine streaming interface
5) Add more support for Vert.x reakt
   a) Support streaming via Reakt
   b) Create client wrappers, event bus wrapper, etc.
6) More integration libs
   a) Spring Reactor, Servlet async, Java EE, JMS, Kafka, Reactive Streaming, Rxjava, Java 9 Flow



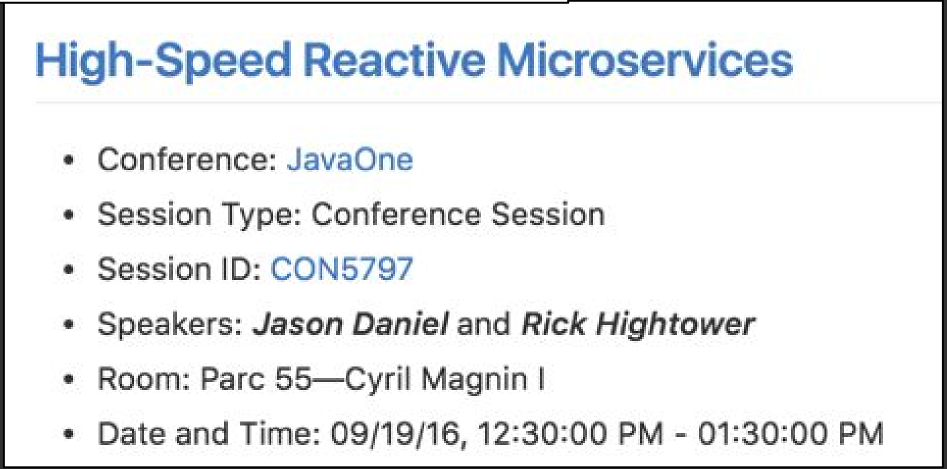


Related Talks



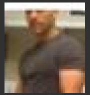
Reactive Java: Promises and Streams with Reakt in Practice

- Conference: JavaOne
- Session Type: HOL (Hands-on Lab) Session
- Session ID: HOL5852
- Speakers: ***Geoff Chandler***, ***Jason Daniel***, and ***Rick Hightower***
- Room: Hilton—Franciscan Room C/D
- Date and Time: 09/20/16, 04:00:00 PM - 06:00:00 PM



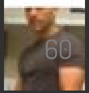
High-Speed Reactive Microservices

- Conference: JavaOne
- Session Type: Conference Session
- Session ID: CON5797
- Speakers: ***Jason Daniel*** and ***Rick Hightower***
- Room: Parc 55—Cyril Magnin I
- Date and Time: 09/19/16, 12:30:00 PM - 01:30:00 PM



Conclusion

- Reakt provides an easy-to-use lib for handling async callbacks
  - It uses Promise concepts from ES6 which seem well thought out and natural
  - We worked with many async libs and wrote a few our self, and really like the ES6 terminology and ease of use
  - Since Java is MT and JavaScript is not there are some differences
  - Java 8/9 lambda expression friendly
- Async call coordination can be difficult but all promises, any promises, reactor with replay promises and timeouts make it easier
- Reakt is evolving and we welcome feedback and contributions (bug reports, pull requests, articles, blogs, etc.)



Extra Material

Author Bio

Author Geoff Chandler

Senior Director at a large Media Company.

Works with Node.js, Cassandra, Mesos, QBit, EC2, and reactive programming. Major

Contributor to QBit, Spring Boot, Reakt, and more.

Creator of Lokate, ddp-client-java, guicefx, and various devops tools for gradle.

Author Bio Rick Hightower

Rick frequently writes about and develops high-speed microservices. He focuses on streaming, monitoring, alerting, and deploying microservices. He was the founding developer of QBit and Reakt as well as Boon.

Worked example

These are extra slides for a worked stream example

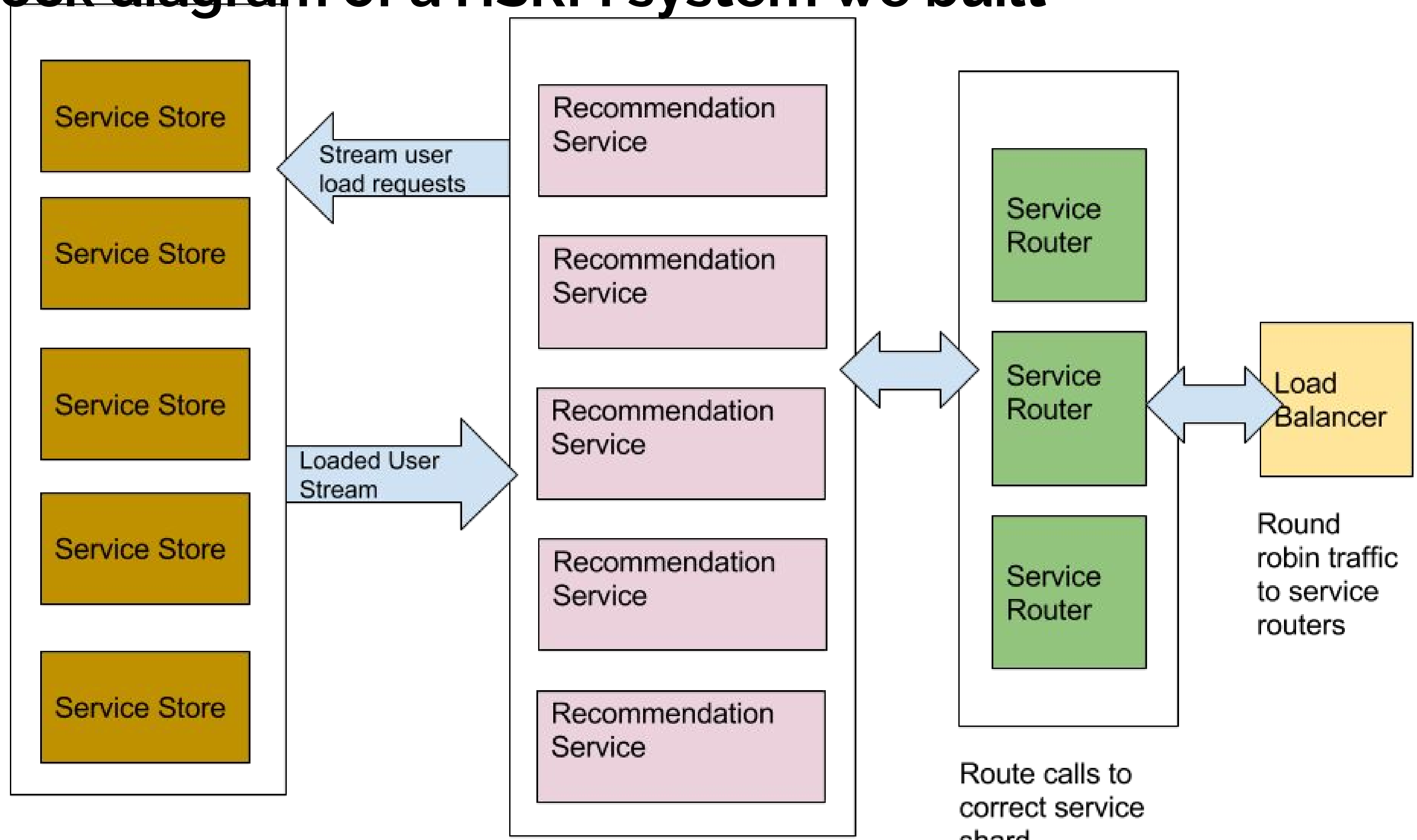
Example

Take from a real world scenario which gave birth to use using Vert.x, and later creating QBit and Reakt

Example uses Streams, and Promises

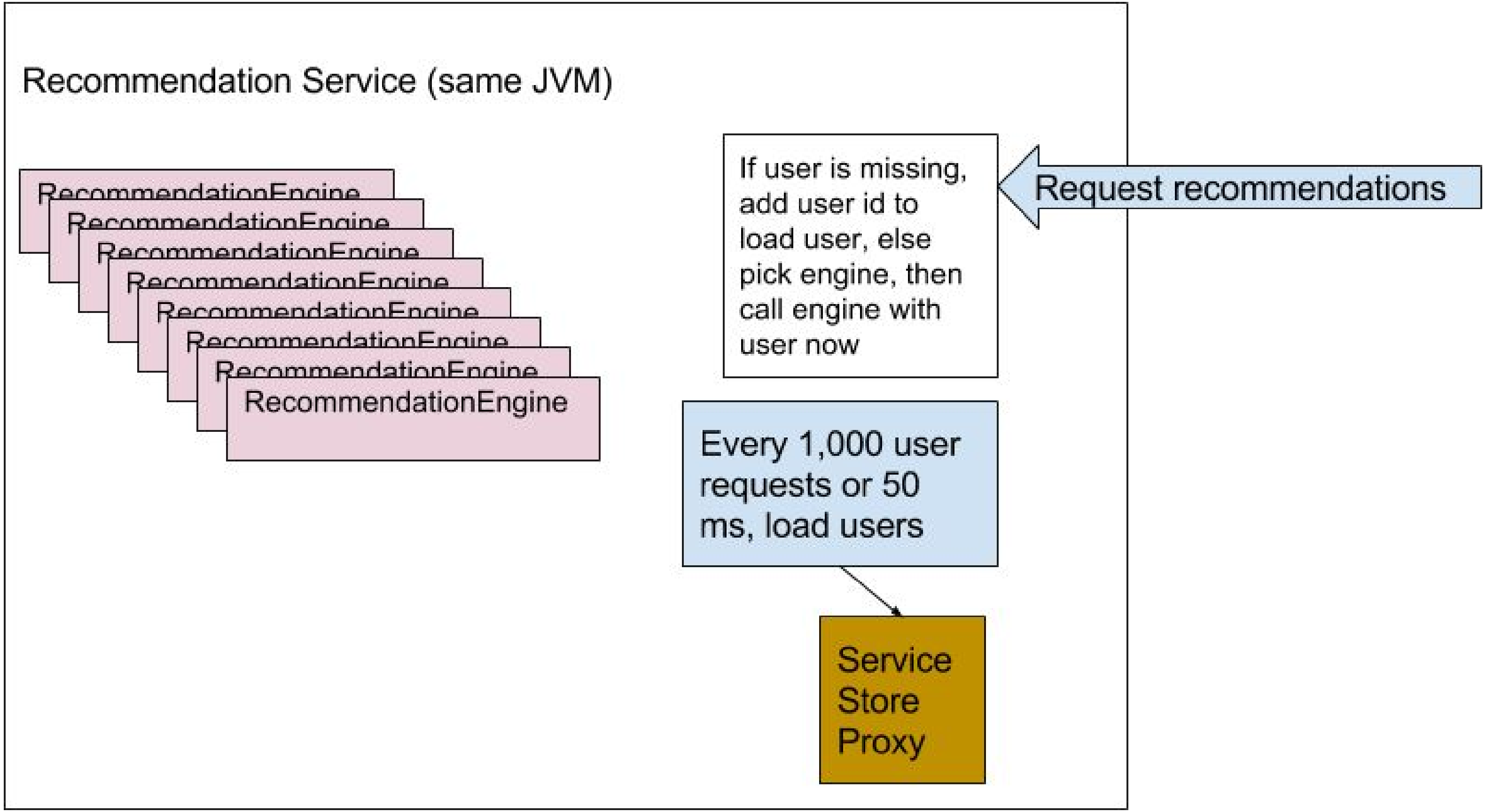
This is not the actual code from the actual project (this is just an example)

Block diagram of a HSRM system we built



Round robin traffic to service routers

Route calls to correct service shard



Recommendation Service (same JVM)

RecommendationEngine

RecommendationEngine

RecommendationEngine

RecommendationEngine

RecommendationEngine

RecommendationEngine

RecommendationEngine

RecommendationEngine

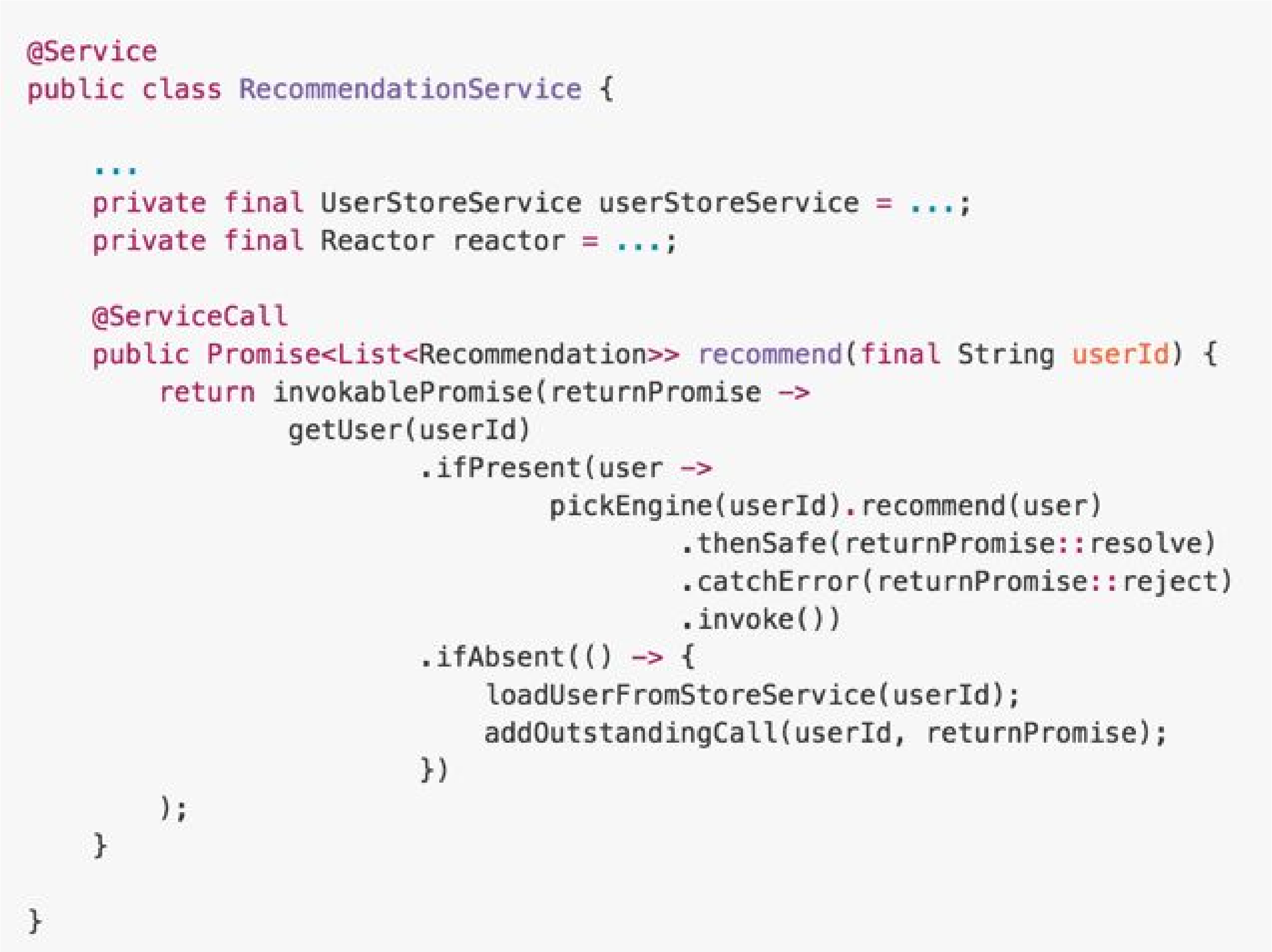
If user is missing, add user id to load user, else pick engine, then call engine with user now

Request recommendations

Every 1,000 user requests or 50 ms, load users

Service Store Proxy

```
async call loadRecommendations(user) :
                    Promise<List<Recommendations>>
```

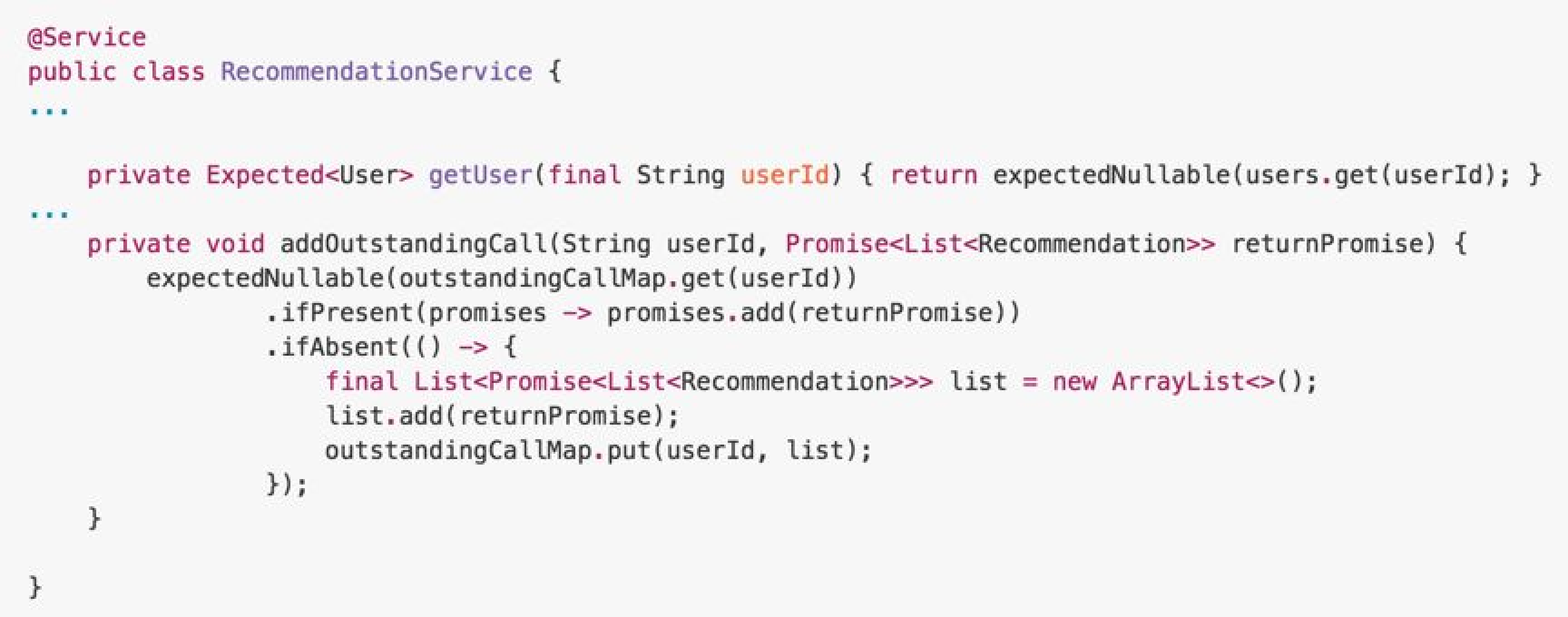


```java
@Service
public class RecommendationService {

    ...
    private final UserStoreService userStoreService = ...;
    private final Reactor reactor = ...;

    @ServiceCall
    public Promise<List<Recommendation>> recommend(final String userId) {
        return invokablePromise(returnPromise ->
                getUser(userId)
                        .ifPresent(user ->
                                pickEngine(userId).recommend(user)
                                        .thenSafe(returnPromise::resolve)
                                        .catchError(returnPromise::reject)
                                        .invoke())
                        .ifAbsent(() -> {
                            loadUserFromStoreService(userId);
                            addOutstandingCall(userId, returnPromise);
                        })
        );
    }

}
```

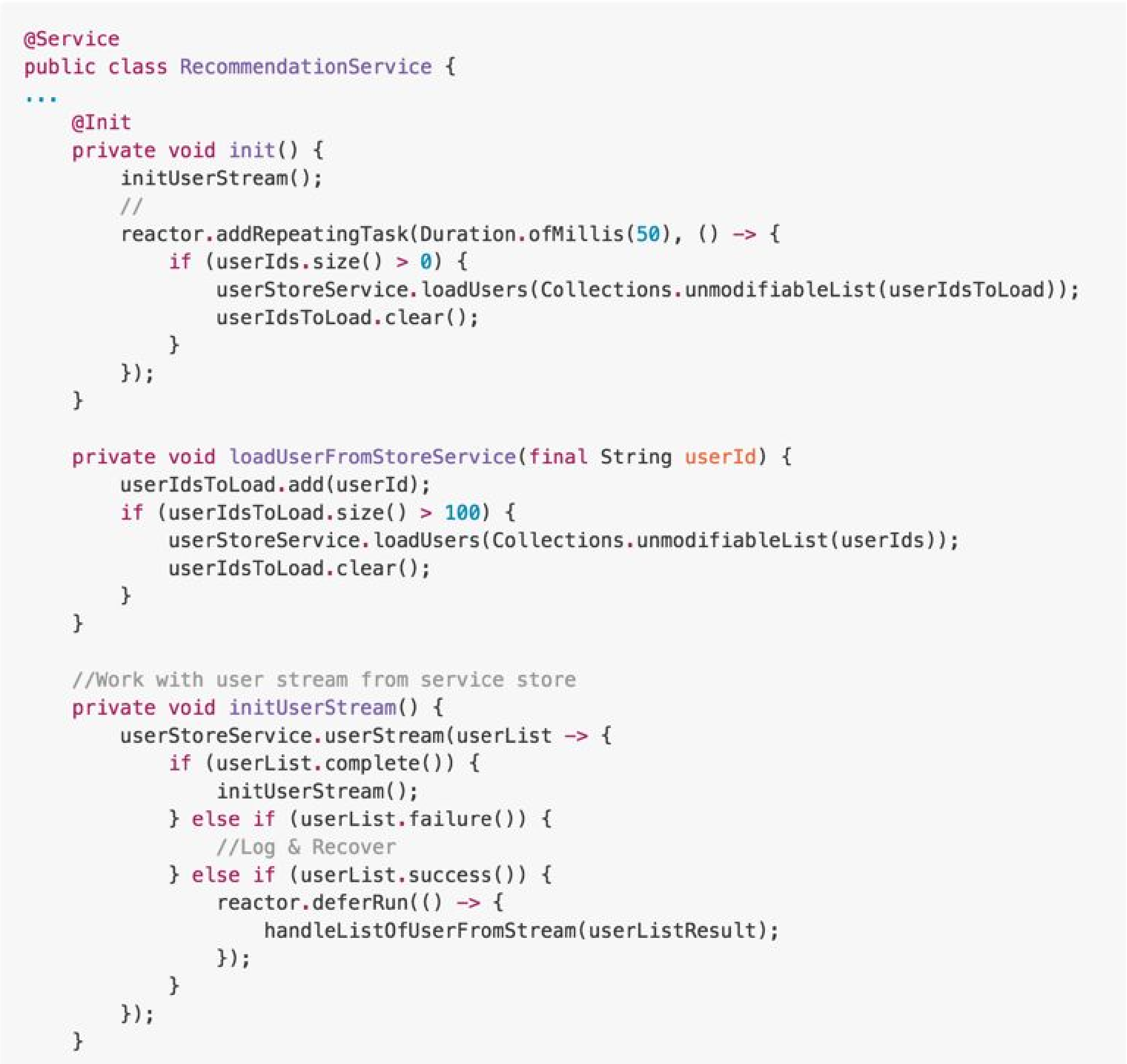
Async call recommendation engine if user present

If not add user id to call batch, add to outstanding call for user.

Adding an outstanding call (promise)



```
@Service
public class RecommendationService {
...

    private Expected<User> getUser(final String userId) { return expectedNullable(users.get(userId); }
...
    private void addOutstandingCall(String userId, Promise<List<Recommendation>> returnPromise) {
        expectedNullable(outstandingCallMap.get(userId))
                .ifPresent(promises -> promises.add(returnPromise))
                .ifAbsent(() -> {
                    final List<Promise<List<Recommendation>>> list = new ArrayList<>();
                    list.add(returnPromise);
                    outstandingCallMap.put(userId, list);
                });
    }

}
```

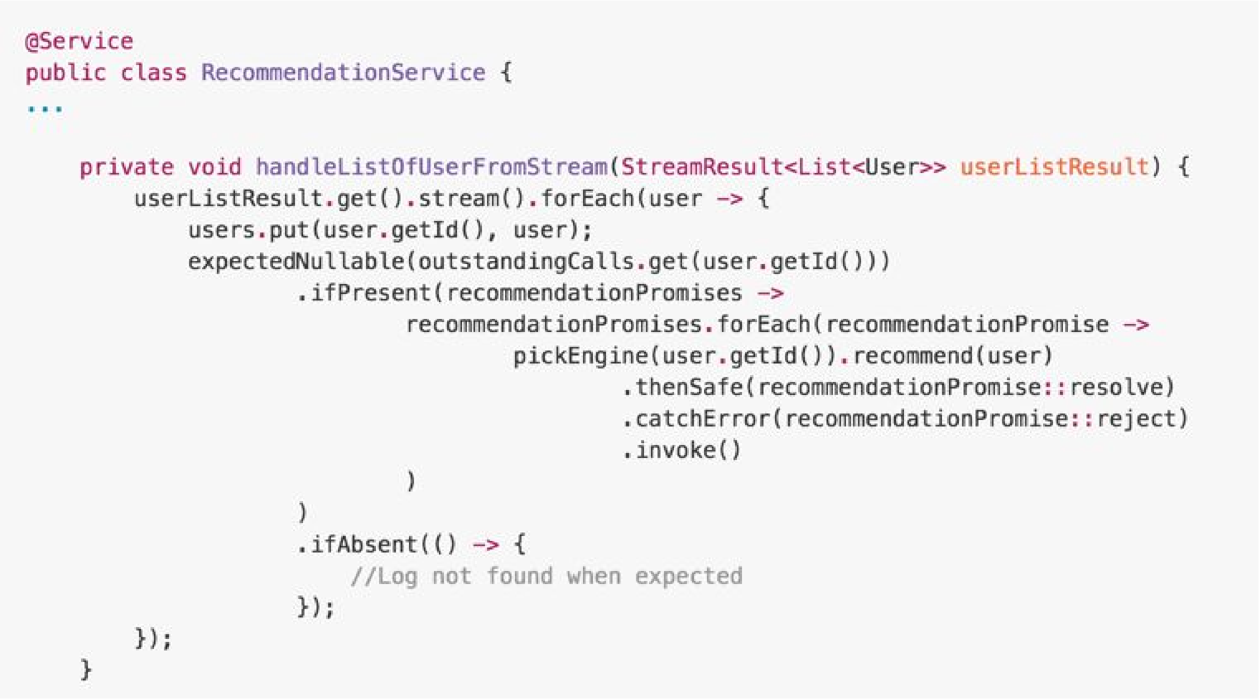


```
@Service
public class RecommendationService {
...
    @Init
    private void init() {
        initUserStream();
        //
        reactor.addRepeatingTask(Duration.ofMillis(50), () -> {
            if (userIds.size() > 0) {
                userStoreService.loadUsers(Collections.unmodifiableList(userIdsToLoad));
                userIdsToLoad.clear();
            }
        });
    }

    private void loadUserFromStoreService(final String userId) {
        userIdsToLoad.add(userId);
        if (userIdsToLoad.size() > 100) {
            userStoreService.loadUsers(Collections.unmodifiableList(userIds));
            userIdsToLoad.clear();
        }
    }

    //Work with user stream from service store
    private void initUserStream() {
        userStoreService.userStream(userList -> {
            if (userList.complete()) {
                initUserStream();
            } else if (userList.failure()) {
                //Log & Recover
            } else if (userList.success()) {
                reactor.deferRun(() -> {
                    handleListOfUserFromStream(userListResult);
                });
            }
        });
    }
```

Every 50 ms check to see if the userIdsToLoad is greater than 0, If so request those users now.

When a user is not found loadUserFromStoreService is called. If there are 100, outstanding requests, then load those users now.

Listen to the userStoreService's userStream



```
@Service
public class RecommendationService {
...

    private void handleListOfUserFromStream(StreamResult<List<User>> userListResult) {
        userListResult.get().stream().forEach(user -> {
            users.put(user.getId(), user);
            expectedNullable(outstandingCalls.get(user.getId()))
                    .ifPresent(recommendationPromises ->
                            recommendationPromises.forEach(recommendationPromise ->
                                    pickEngine(user.getId()).recommend(user)
                                            .thenSafe(recommendationPromise::resolve)
                                            .catchError(recommendationPromise::reject)
                                            .invoke()
                            )
                    )
                    .ifAbsent(() -> {
                        //Log not found when expected
                    });
        });
    }
```

Process the stream result.

Populate the user map (or use a Simple cache with an expiry and a max number of users allowed to be in system).

Since the user is now loaded, see if their are outstanding calls (promises) and resolve those calls.

Slides we pulled out

Streams vs Service Calls

Microservices / RESTful services / SOA services

REST / HTTP calls common denominator

Even messaging can be request/reply

Streams vs. Service Calls

- Level of abstraction differences,
- Calls can be streamed, Results can be streamed
- What level of abstraction fits the problem you are trying to solve
- Are streams an implementation details or a direct concept?

Related projects

- QBit Java Microservice (built on top of Vert.x for IO)
  - Using Reakt reactor to manage callbacks,
  - REST and WebSocket services (WebSocket RPC) use Reakt Promises and Reakt Callbacks
- Lokate - service discovery lib for DNS-A, DNS-SRV, Consul, Mesos, Marathon
  - Uses Reakt invokeable promises (Vert.x for IO)
- Elekt - leadership lib that uses tools like Consul to do leadership election (uses promises)
- Reakt-Guava - Reakt Bridge to Guava listable futures
- Reakt-Vertx - Reakt Bridge for Vert.x AsyncCallbackHandler
- Reakt-DynamoDB - Reakt wrapper for async DynamoDB
- Reakt-Cassandra - Reakt wrapper for async Cassandra access

Promise

- Promises can be used for all manners of async programming
  - not just Reactor Pattern
- You can use it with standard Java Lib
- Bridges for **Guava**, **Vert.x** and **Cassandra**
- **QBit** uses it (Service Actor/Microservices),
- **Lokate** (Discovery),
- **Elekt** (Leadership election),
- Clean interface for dealing with async programming

Other Async Models

- Messaging (Golang, Erlang, RabbitMQ, JMS, Kafka)
- Actors (Erlang, Akka)
- Active Objects (Akka types actors, DCOM)
- Common problems when dealing with handling calls to services:
    - Handling the call

Reactor works with

Works with Reactor Architecture (Vert.x, Spring Reactor)

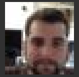
Works with Actor model and Active Objects (Akka actors, Akka typed actor, QBit, etc.)

ReplayPromises need a Reactor

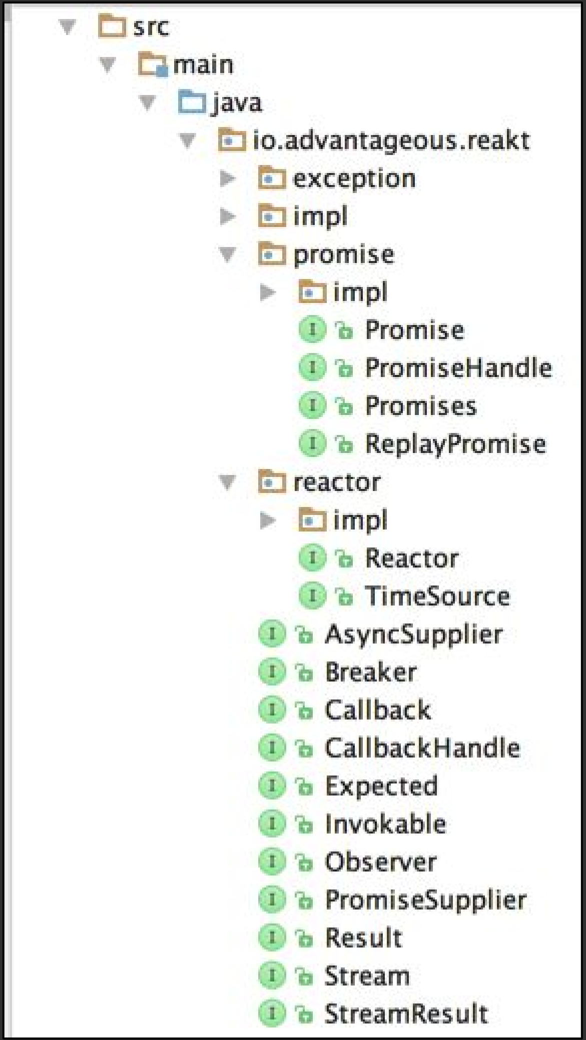
Reactor is an interface

Replace it with one optimized for your environment

Or manage ReplayPromises and tasks with something else like a Reactor



Review



```
▼ src
  ▼ main
    ▼ java
      ▼ io.advantageous.reakt
        ▶ exception
        ▶ impl
        ▼ promise
          ▶ impl
            Promise
            PromiseHandle
            Promises
            ReplayPromise
        ▼ reactor
          ▶ impl
            Reactor
            TimeSource
          AsyncSupplier
          Breaker
          Callback
          CallbackHandle
          Expected
          Invokable
          Observer
          PromiseSupplier
          Result
          Stream
          StreamResult
```

Most of these
Should be familiar to you