Practical Ratpack Promises

27 March 2017

Written by Jon Bevan (https://twitter.com/jdbevan)

There are a lot of excellent resources describing how Ratpack works (http://ldaley.com/post/97376696242/ratpack-execution-model-part-1) and explaining its execution model (http://ldaley.com/post/102495950257/ratpacks-execution-model-in-practice) in practice, and with examples (http://naleid.com/blog/2016/05/01/ratpack-executions-async-plus-serial-not-parallel) too. There's also the official documentation (https://ratpack.io/manual/current/async.html).

However, articles like this one (http://beckje01.com/blog/2014/09/10/ratpack-promise/) only scratch the surface of Promises and are also 2 years old at the time of writing, so I wanted to write about Promises in Ratpack (https://ratpack.io/) from a practical perspective, having used them now for several months.

What's a Promise?

A Promise is an object that will eventually yield or provide a value - often for an asynchronous computation e.g. some kind of network request. Promises are implemented with a fluent interface that allow you to build a chain of actions that will occur when the value of the Promise is yielded. Promises are implemented in Javascript (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise), in Java as Futures (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) and in other programming languages like C++, Python, Scala and R.

It's important to note that Promises don't guarantee that a value will ever be yielded, and additionally, they may fail to provide a value. In Ratpack, a failed/errored Promise is caused by an Exception.

So, how do I use a Ratpack Promise?

Well, a simple if unrealistic example is as follows:

```
Promise.value("Hello World").then { String message ->
    println message
}
```

The Promise.value (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#value-T-) static method call creates a Promise that has a value that is immediately available, which makes this example slighly unrealistic, although it can be a useful method for testing and caching.

The .then (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#then-ratpack.func.Action-) method call on the Promise is what triggers the Promise to be evaluated.

> If `.then` is not called then the code inside the Promise that calculates the value is not called.

.then (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#then-ratpack.func.Action-) also returns void, so you can't chain any additional method calls.

What about a real-world example then?

As I hinted at above, most of the real-world examples are to do with asynchronous calculations which often include some kind of I/O like a network request.

In the work I've been doing recently we use the AWS Java (https://aws.amazon.com/sdk-for-java/) library a lot, so I'll use that in the examples here. The AWS library doesn't know about Ratpack's Promises, but it does perform network requests so we wrap the library calls like this:

```
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
}
```

You can read more about `Blocking.get` in the documentation (https://ratpack.io/manual/current/async.html), but it returns a Promise for the value returned by the nested API call. In this case it returns us a Promise for the CreateTableResult (http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/dynamodbv2/model/CreateTableResult.html) object that Amazon's DynamoDB client returns.

So, a real world example of using Promises inside of a Ratpack request handler could be:

```
void handle(Context context) {
    def createTableRequest = buildTableRequest(context)
    Blocking.get {
        dynamoDbClient.createTable(createTableRequest)
    } then { CreateTableResult result ->
        context.response.send("The table ARN is: ${result.tableDescription.tableArn}")
    }
}
```

For brevity I'm going to mostly omit the `void handle(Context context)...` stuff from the following examples.

What happens if there's a bug, or the Internet breaks?

Promises get broken. The thing that returns or calculates the value we've been promised may break, sometimes because we provide invalid data, sometimes because we haven't authenticated ourselves successfully, sometimes because we write buggy code, sometimes because the network infrastructure we're running on lets us down and sometimes because the external service we interact with has some internal problem of its own.

Ratpack provides a couple ways of dealing with Promises that fail.

onError

We can handle all Exceptions using .onError (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#onError-ratpack.func.Predicate-ratpack.func.Action-):

```
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} onError { Throwable t ->
    log.error("Table creation request failed", t)
    context.response.send("Sorry the table creation failed")
} then { CreateTableResult result ->
    context.response.send("The table ARN is: ${result.tableDescription.tableArn}")
}
```

Or we can handle only specific Exceptions too:

```
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} onError(ResourceInUseException, { ResourceInUseException e ->
    log.error("Table creation request failed", e)
    context.response.send("That table already exists!")
}) then { CreateTableResult result ->
    context.response.send("The table ARN is: ${result.tableDescription.tableArn}")
}
```

> It's important to note that once `.onError` has been called, nothing else in the promise chain is called. It's a terminal operation.

It's also important to note that the location of your error handling methods within the promise chain is significant. Exceptions will only be handled by the onError (or mapError etc) methods if the error has occurred earlier in the promise chain. Here's an example:

```
// Let's assume that buildTableRequest returns Promise<CreateTableRequest> for this example
buildTableRequest(context) onError { Exception e ->
    // This onError closure will only handle errors from promise returned by the buildTableRequest met
hod
    log.error("The request was invalid", e)
    context.clientError(HttpStatus.SC_BAD_REQUEST)
} flatMap { CreateTableRequest createTableRequest ->
    // I'll explain flatMap below
    Blocking.get {
        dynamoDbClient.createTable(createTableRequest)
    }
} onError(ResourceInUseException, { ResourceInUseException e ->
    // This onError closure will only handle ResourceInUseException from between the
    // previous onError handler and this point here
    log.error("Table creation request failed", e)
    context.response.send("That table already exists!")
}) then { CreateTableResult result ->
    context.response.send("The table ARN is: ${result.tableDescription.tableArn}")
}
```

mapError

Sometimes, certain errors are OK and we can recover from them. In this scenario we can use .mapError (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#mapError-ratpack.func.Function-) to convert our exception into some valid object that the rest of our code can consume.

Like `.onError` we can map all exceptions or we can map specific exceptions.

```
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} mapError(ResourceInUseException, { ResourceInUseException e ->
    log.warn("Table creation request failed", e)
    new CreateTableResult()
        .withTableDescription(
            new TableDescription()
                .withTableArn("some-default-value")
                .withTableStatus(TableStatus.ACTIVE))
}) then { CreateTableResult result ->
    context.response.send("The table ARN is: ${result.tableDescription.tableArn}")
}
```

flatMapError

If our error mapping code interacts with a service that returns a promise, then we need to use the .flatMapError (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#flatMapError-ratpack.func.Function-) method, otherwise we'll end up with a promise within a promise.

```
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} flatMapError(ResourceInUseException, { ResourceInUseException e ->
    log.info("Table creation request failed", e)
    // Let's assume this method call returns Promise<CreateTableResult>
    getOriginalCreateTableResultFromExternalCache(context)
}) then { CreateTableResult result ->
    context.response.send("The table ARN is: ${result.tableDescription.tableArn}")
}
```

If we didn't use `.flatMapError` in the example above, then we'd end up with something like this:

```
/**
 * DONT DO THIS
 */
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} mapError(ResourceInUseException, { ResourceInUseException e ->
    log.info("Table creation request failed", e)
    // Let's assume this method call returns Promise<CreateTableResult>
    getOriginalCreateTableResultFromExternalCache(context)
}) then { Promise<CreateTableResult> promise ->
    promise.then { CreateTableResult result ->
        context.response.send("The table ARN is: ${result.tableDescription.tableArn}")
    }
}
```

What if I need to change the promised value?

Very often, the value returned by the services and APIs you use don't return the data you want in the format you want it to be in. In order to change one promised value to another you can use the following methods:

map

For straight forward object manipulation, .map (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#map-ratpack.func.Function-) is your friend:

```
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} map { CreateTableResult result ->
    result.tableDescription.tableArn
} then { String tableArn ->
    context.response.send("The table ARN is: ${tableArn}")
}
```

flatMap

If our value conversion depends on some other promised value, we can use .flatMap (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#flatMap-ratpack.func.Function-) like in the error handling example above:

```
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} flatMap { CreateTableResult result ->
    // Let's assume this method call returns Promise<TableDescription>
    externalCache.put(context, result)
} then { TableDescription description ->
    context.response.send("The table ARN is: ${description.tableArn}")
}
```

And of course we can chain these too:

```
// import static ratpack.jackson.Jackson.fromJson

context.parse(fromJson(Map)) map { Map tableDetails ->
    // This method call synchronously returns a CreateTableRequest
    buildCreateTableRequest(tableDetails)
} flatMap { CreateTableRequest tableRequest ->
    Blocking.get {
        dynamoDbClient.createTable(createTableRequest)
    }
} then { CreateTableResult result ->
    context.response.send("The table ARN is ${result.tableDescription.tableArn}")
}
```

mapIf / flatMapIf

There are also some conditional mapping operations you can use on a promise: .mapIf (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#mapIf-ratpack.func.Predicate-ratpack.func.Function-) and .flatMapIf (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#flatMapIf-ratpack.func.Predicate-ratpack.func.Function-)

I haven't used these personally, but they are part of Ratpack's Promise API.

What if I just want to intercept the promised value?

We use these often for logging, or for fire-and-forget operations.

next

The .next (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#next-ratpack.func.Action-) and .nextOp (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#nextOp-ratpack.func.Function-) calls can be used to do some work using the current value of the Promise at that point in the Promise chain, without changing the value of the Promise.

If the Promise has failed, this call does not get invoked, so in the code below, we'll only log request timings for successful requests.

```
def start = System.currentTimeMillis()
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} next { CreateTableResult result ->
    // This code will not be invoked if the dynamoDbClient.createTable call threw an exception
    def taken = System.currentTimeMillis() - start
    log.info("Table creation request for {} took {}ms", result.tableDescription.tableArn, taken)
} onError { Throwable t ->
    log.error("Table creation request failed", t)
    context.response.send("Sorry the table creation failed")
} then { CreateTableResult result ->
    context.response.send("The table ARN is: ${result.tableDescription.tableArn}")
}
```

wiretap

The .wiretap (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#wiretap-ratpack.func.Action-) call, in contrast to `.next`, does get called regardless of the state of the Promise. Instead of being passed the value of the Promise at that point in the chain, it gets a Result (https://ratpack.io/manual/current/api/ratpack/exec/Result.html) wrapper which can be inspected to see the state of the Promise.

```
def start = System.currentTimeMillis()
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} wiretap { Result<CreateTableResult> result ->
    def taken = System.currentTimeMillis() - start
    if (result.isError()) {
        log.warn("Table creation request failed after {}ms", taken)
    } else {
        log.info("Table creation request for {} took {}ms", result.tableDescription.tableArn, taken)
    }
} onError { Throwable t ->
    log.error("Table creation request failed", t)
    context.response.send("Sorry the table creation failed")
} then { CreateTableResult result ->
    context.response.send("The table ARN is: ${result.tableDescription.tableArn}")
}
```

We use `.wiretap` along with ParallelBatch.yieldAll() (https://ratpack.io/manual/current/api/ratpack/exec/util/ParallelBatch.html#yieldAll--) to determine which of our parallel promises failed.

I want to combine the promised value with another value

left / right

These two methods allow us to convert our promised value into a Pair (https://ratpack.io/manual/current/api/ratpack/func/Pair.html) of values. The .left (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#left-ratpack.func.Function-) method allows us to specify what will be on the 'left' of the Pair (the original Promised value will be on the right), and the .right (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#right-ratpack.func.Function-) method allows us to specify what will be on the 'right' of the Pair (the original Promised value will be on the left).

```
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} left { CreateTableResult result ->
    result.tableDescription
} then { Pair<TableDescription, CreateTableResult> pair ->
    def tableDesc = pair.left
    context.response.send("The table ARN is: ${tableDesc.tableArn}")
}
```

or we could combine this with the `.map` call from earlier:

```
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} map { CreateTableResult result ->
    result.tableDescription
} right { TableDescription desc ->
    desc.tableArn
} then { Pair<TableDescription, String> pair ->
    def tableDesc = pair.left
    def tableArn = pair.right
    context.response.send("The ARN for ${tableDesc.tableName} is: ${tableArn}")
}
```

flatLeft / flatRight

The .flatLeft (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#flatLeft-ratpack.func.Function-) and .flatRight (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#flatRight-ratpack.func.Function-) methods operate in the same way as the `.left` and `.right` calls, but are used when you want to call some method that returns a Promise, and add the value of that Promise into the Pair.

```
Blocking.get {
    dynamoDbClient.createTable(createTableRequest)
} flatRight { CreateTableResult result ->
    def attributes = [created: new AttributeValue(System.currentTimeMillis() as String)]
    def createdItemRequest = new PutItemRequest(result.tableDescription.tableName, attributes)

    // This Blocking.get call will return Promise<PutItemResult>
    Blocking.get {
        dynamoDbClient.putItem(createdItemRequest)
    }
} then { Pair<CreateTableResult, PutItemResult> pair ->
    def tableDescription = pair.left.tableDescription
    def putItemResult = pair.right
    log.info("Consumed capacity: {}", putItemResult.consumedCapacity.capacityUnits)
    context.response.send("The ARN for ${tableDescription.tableName} is: ${tableDescription.tableArn}"
)
}
```

I want to bail out of my promise chain early

As well as the `.then` and `.onError` terminating parts of the Promise chain, you can also specify that the chain should stop processing using a .route (https://ratpack.io/manual/current/api/ratpack/exec/Promise.html#route-ratpack.func.Predicate-ratpack.func.Action-) method call which takes a predicate and an action.

> If the predicate for a `.route` call returns true, ONLY the action specified in the `.route` will be executed

route

In this example, we short-circuit the promise chain if the request contains some indicator that this is just a test:

```
// import static ratpack.jackson.Jackson.fromJson

context.parse(fromJson(Map)) route ({ Map requestBody ->
    requestBody.containsKey('test')
}, { Map requestBody ->
    // If the predicate (1st param given to route) returns true, we execute this code
    // but not the rest of the Promise chain
    context.response.send("This request was just a test")
}) map { Map tableDetails ->
    // This will not run if the requestBody contains a 'test' map key
    buildCreateTableRequest(tableDetails)
} flatMap { CreateTableRequest tableRequest ->
    // This will not run if the requestBody contains a 'test' map key
    Blocking.get {
        dynamoDbClient.createTable(createTableRequest)
    }
} then { CreateTableResult result ->
    // This will not run if the requestBody contains a 'test' map key
    context.response.send("The table ARN is ${result.tableDescription.tableArn}")
}
```

onNull

I've never used this but it looked interesting, so I thought I'd mention it here. If the Promised value is null, then this action will be taken:

```
Blocking.get {
    externalCache.get(context.request.queryParams['id'])
} onNull {
    context.clientError(HttpStatus.SC_NOT_FOUND)
} then { String cachedValue ->
    // This won't be executed if a promise of null was returned from the cache
    context.response.send(cachedValue)
}
```

Hopefully some examples, along with documentation links, help understand how the Promises work in Ratpack and how powerful the Promise chain can be!

- 09 Oct 2018 » A strange bug on AWS Lambda (/code/2018/10/09/a-strange-bug-on-aws-lambda)
- 17 Jan 2018 » How to run Karma tests in browsers in Docker (/testing/2018/01/17/how-to-run-karma-tests-in-browsers-in-docker)
- 07 Dec 2017 » Switching from Javascript to Typescript (/code/2017/12/07/switching-from-javascript-to-typescript)
- 30 Oct 2017 » Fun with React event handlers (/code/2017/10/30/fun-with-react-event-handlers)
- 17 Jul 2017 » Switching from Groovy to Java (/code/2017/07/17/switching-from-groovy-to-java)
- 24 May 2017 » Useful Git Aliases (/git/2017/05/24/useful-git-aliases)
- 27 Mar 2017 » Practical Ratpack Promises (/code/2017/03/27/practical-ratpack-promises)
- 03 Nov 2016 » Custom Content in Forms for Confluence Connect (/confluence/2016/11/03/custom-content-in-forms-for-confluence-connect)
- 04 Oct 2016 » Checking user permissions from REST calls (/confluence/2016/10/04/checking-user-permissions-from-rest-calls)
- 30 Sep 2016 » Using the reflection API in Confluence (/confluence/2016/09/30/using-reflection-in-confluence)
- 28 Sep 2016 » Creating a custom Confluence Blueprint (/confluence/2016/09/28/creating-a-custom-confluence-blueprint)

- 06 Sep 2016 » ReactJS in Forms for Confluence Connect (/confluence/2016/09/06/react-js-in-forms-for-confluence-connect)
- 25 Apr 2016 » Migrating to ES6 in Atlassian Add-ons (/code/2016/04/25/migrating-to-es6-in-atlassian-add-ons)
- 17 Mar 2016 » All kinds of things I learnt trying to performance test against Fisheye/Crucible (/performance/2016/03/17/all-kinds-of-things-i-learnt-trying-to-performance-test-against-fisheyecrucible)
- 24 Dec 2015 » Adaptavist's Holiday Gift of Atlassian Deployment Automation (/labs/2015/12/24/christmas-present-opensource)
- 17 Dec 2015 » Getting a Custom Field value safely (/code/java/jira/2015/12/17/getting-custom-field-values-safely)
- 07 Dec 2015 » Putting Google Analytics to work with plugins for Confluence (/confluence/2015/12/07/using-google-analytics-to-track-plugin-usages-in-confluence)
- 02 Dec 2015 » Devoxx Voting, A retrospective (/code/2015/12/02/devoxx-be-voting-a-retrospective)
- 25 Nov 2015 » Some things I've learnt about SingleSelect (/code/groovy/2015/11/25/some-things-Ive-learnt-about-singleselect)
- 15 Oct 2015 » Using SOY for JIRA actions (/2015/10/15/using-soy-for-jira-actions)
- 26 Sep 2015 » Object Reflection in Groovy (/code/groovy/2015/09/26/object-reflection-in-groovy)
- 22 Sep 2015 » Introducing Adaptavist Labs (/labs/2015/09/22/introducing-adaptavist-labs)

code 10 (/categories.html#code-ref)

code 7 (/tags.html#code-ref) ratpack 1 (/tags.html#ratpack-ref) groovy 4 (/tags.html#groovy-ref)

« Previous (/confluence/2016/11/03/custom-content-in-forms-for-confluence-connect) Archive (/archive.html)
Next » (/git/2017/05/24/useful-git-aliases)