Mastering Async with Ratpack

Daniel Hyun
– @danhyun (https://github.com/danhyun) @LSpacewalker (https://twitter.com/LSpacewalker)
– 2017 June 1

Table of Contents

1. Details

Code: https://github.com/danhyun/mastering-async-ratpack/

Notes: https://danhyun.github.io/mastering-async-ratpack/

PDF: https://danhyun.github.io/mastering-async-ratpack/notes.pdf

2. Need for Async

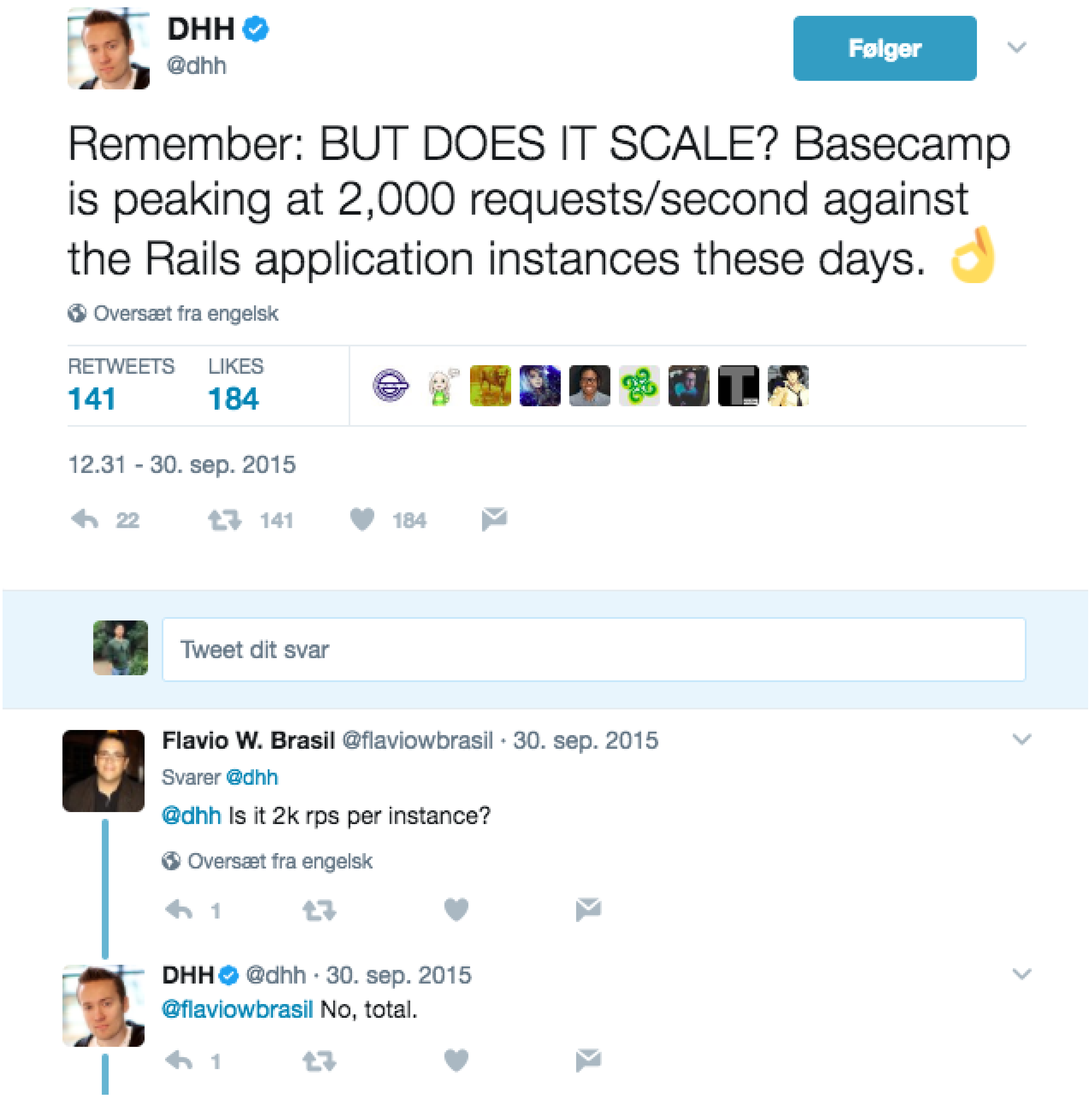
We want to do more with less.

Resources are expensive: You pay for memory/compute/network usage

```
cost    | $$$                | $$      | $
--------|-------------------------------------------
method  | Multiple Processes | threads | event loop
example | apache             | servlet | netty
----------------------------------------------------
```

http://www.kegel.com/c10k.html

3. Scalability



DHH @dhh

Følger

Remember: BUT DOES IT SCALE? Basecamp is peaking at 2,000 requests/second against the Rails application instances these days. 👌

Oversæt fra engelsk

RETWEETS 141 LIKES 184

12.31 - 30. sep. 2015

22 141 184

Tweet dit svar

Flavio W. Brasil @flaviowbrasil · 30. sep. 2015

Svarer @dhh

@dhh Is it 2k rps per instance?

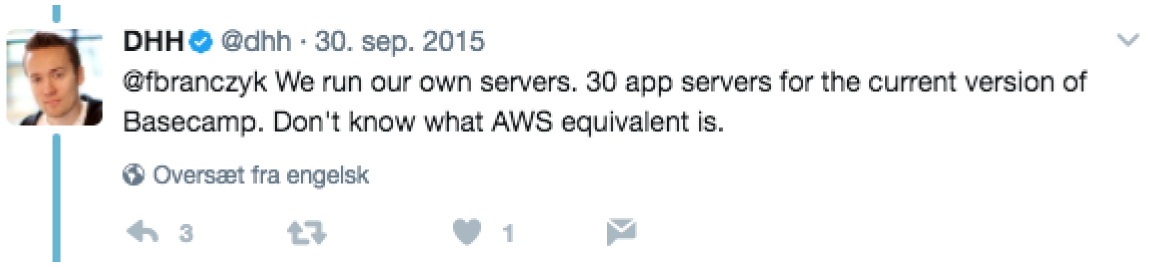
Oversæt fra engelsk

1

DHH @dhh · 30. sep. 2015

@flaviowbrasil No, total.

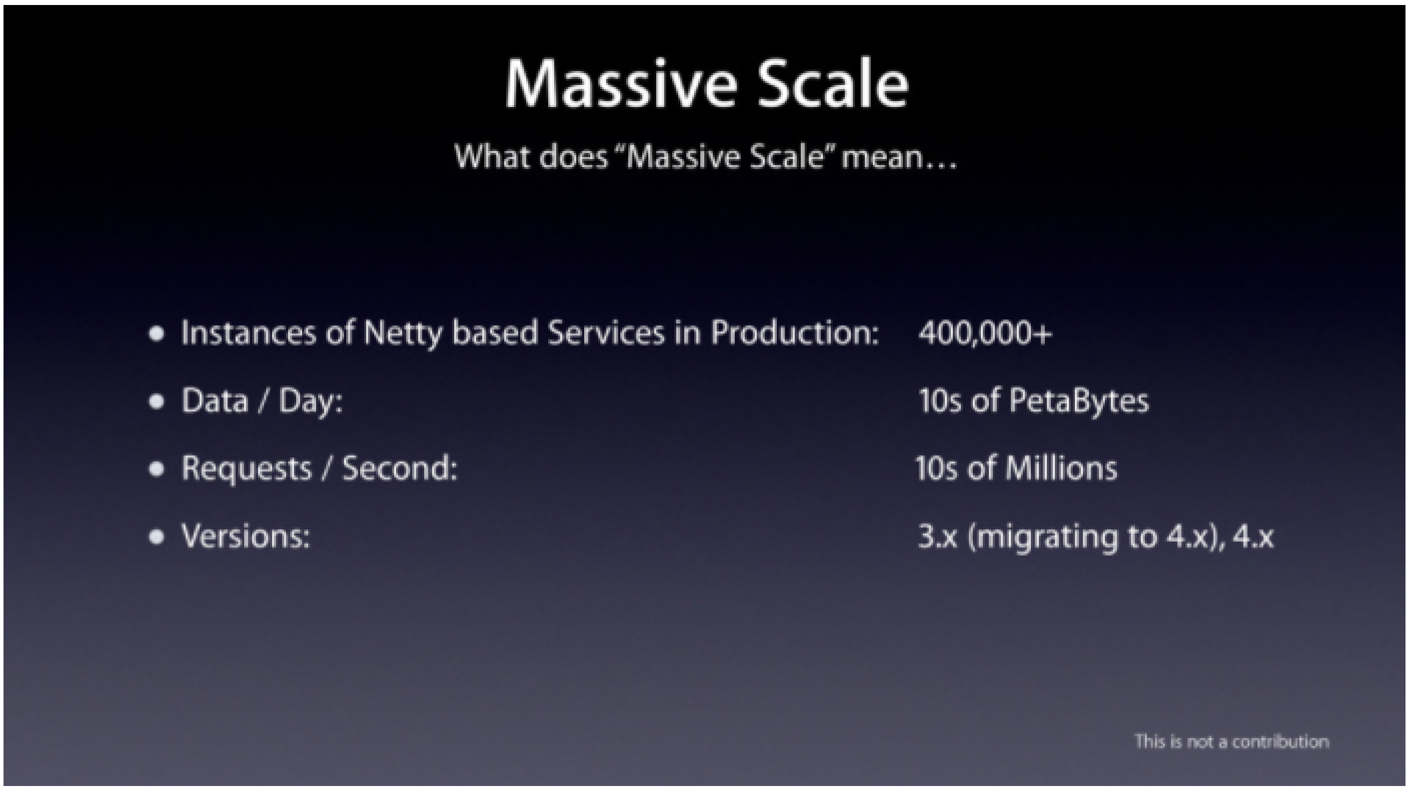
1


DHH @dhh · 30. sep. 2015
@fbranczyk We run our own servers. 30 app servers for the current version of Basecamp. Don't know what AWS equivalent is.

Source: https://twitter.com/dhh/status/649260226521210880

2000 peak rps for 30 servers sounds expensive…

Meanwhile at Apple…


Massive Scale

What does "Massive Scale" mean…

Instances of Netty based Services in Production:	400,000+
Data / Day:	10s of PetaBytes
Requests / Second:	10s of Millions
Versions:	3.x (migrating to 4.x), 4.x

This is not a contribution

https://speakerdeck.com/normanmaurer/connectivity

That's pretty big scale… Still expensive xD

Takeaways

- Need async to reduce footprint, especially important with each service that gets deployed
- Async is difficult, not everyone knows Netty
- Need to find balance between scale and usability

Some arguments

"Bottle neck is db calls"

This depends on the nature of your application. Not every app is a CRUD app, and even those that are serve of static/in memory content

4. State of async in Java land

Threads/Executors/Mutexes/AtomicReferences

We have the means, but…

- Testing
- Readability, ease of understanding
- Resource sharing (memory, system resources)

5. Async is hard

- Non-deterministic
- Callback hell
- Error handling/propagation

How many times have you forgotten to send a response to the user after making some async call in nodejs or Playframework? Susceptible to brain damage trying to track down what is happening.

6. Libraries to the rescue

It's no secret that concurrency is hard. Different concurrency models have found their way into the JVM ecosystem over the years:

- Futures (callbacks)
- Queues
- Fork/Join
- Actors
- Disruptor
- Continuations (Quasar/parallel universe)

Last few models try to limit or eliminate resource sharing between running threads.

https://shipilev.net/blog/2014/jmm-pragmatics/

https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/

7. Ratpack

Ratpack is async and non-blocking (Netty Rocks!)

It provides its own concurrency model (execution model) for managing and handling web requests

8. Ratpack Thread Model

Netty's event loop used for compute bound code:

- Entry point to executing your Chain
- Handling NIO events (e.g. read/write)
- Scheduling/Coordinating executions

> Never block the compute thread! Don't make any syscalls that block CPU until operation completes!

Thread pool for running blocking code

- For long running computations or code that blocks CPU until further notice

9. Ratpack Execution

An execution is a collection of units of work to be executed and managed by Ratpack

These units of work are called execution segment

Any http handling code is always done within an execution

Users schedule execution segments via Promise/Operation/Blocking facilities

Execution segments for a given execution are always executed on the same thread

http://ldaley.com/post/97376696242/ratpack-execution-model-part-1

http://ldaley.com/post/102495950257/ratpacks-execution-model-in-practice

10. Ratpack Async Primitives

- Promise
- Operation
- Blocking

All Ratpack primitives are implemented with the Execution api

If you can’t find a method in Promise/Operation/Blocking you can always build it yourself!

11. Promises/Operations

- Creating Promises schedules execution segments
- Promises are executed in the order they were created (except for forked promises)
- They run on cpu or blocking threads, determined at time of creation.
- Easy to adapt with other async libraries (rx-mongo, thread pools)

12. Testability

Ratpack provides ExecHarness test fixture for easy testability

It allows you to run executions without starting a Ratpack server

13. ExecHarness

- java.lang.AutoCloseable
- Convenience methods that let ExecHarness manage start/close

Can use Java 7 try-with-resources using new Parrot project (Bridging the gap between Groovy and Java syntax)

14. ExecHarness#yield

Great for unit testing, executes a given promise and returns the value from the Promise in a blocking fashion

Automatically subscribes to the Promise, no need to call Promise#then

Comes in two varieties:

ExecHarness#yield

Keeps the ExecHarness running

GROOVY

```groovy
given:
ExecHarness execHarness = ExecHarness.harness()  1

when:
ExecResult result = execHarness.yield {  2
  Promise.value('ratpack')  3
}

then:
result.value == 'ratpack'  4

cleanup:
execHarness.close()  5
```

1 Get an instance of ExecHarness

2 Invoke yield on the instance

3 Return a Ratpack Promise from the Closure

4 Extract the value from the Promise

5 Remember to clean up after ourselves!

GROOVY

```groovy
when:
ExecResult result = ExecHarness.yieldSingle {  1
  Promise.value('ratpack')  2
}

then:
result.value == 'ratpack'  3
```

1 Invoke static method ExecHarness.yieldSingle

2 Return Ratpack Promise from Closure to ExecHarness

3 Extract value from Promise

15. ExecHarness#run

Great for seeing Promises in action, closer to coding experience in Ratpack code

No return value

Promises are not automatically subscribed

Comes in two varieties:

ExecHarness#run

Starts and blocks execution until completed

GROOVY
```groovy
given:
ExecHarness execHarness = ExecHarness.harness() 1

expect:
execHarness.run { 2
  Promise.value('ratpack') 3
    .then { String value -> 4
      assert value == 'ratpack' 5
    }
}

cleanup:
execHarness.close() 6
```

1 Get an instance of ExecHarness

2 Pass a closure to ExecHarness#run method

3 Create Ratpack Promise

4 Subscribe to the Promise

5 Use Groovy Power Assert to make assertion on value from Promise

GROOVY
```groovy
expect:
ExecHarness.runSingle { 1
  Promise.value('ratpack') 2
    .then { String value -> 3
      assert value == 'ratpack' 4
    }
}
```

1 Invoke static method ExecHarness.runSingle()

2 Create Ratpack Promise

3 Subscribe to Ratpack Promise

4 Use Groovy Power assert to assert the value resolved from the Promise

When making assertions from within closures you need to make sure that you use Groovy Power Assert. Spock does not apply assertions to values from within the Closure

16. Advanced Async

- Promised

- SerialBatch/ParallelBatch

17. Examples

Promise.sync

GROOVY

```groovy
boolean notExecuted = true
Promise p = Promise.sync {
  notExecuted = false
}

assert notExecuted
```

Promises don’t execute when you create them.

Promise#then

GROOVY

```groovy
given:
Promise<String> p = Promise.sync {
  println 'sync'
  'sync'
}

when:
p.then { s ->
  assert s == 'sync'
}

then:
thrown UnmanagedThreadException
```

Promises execute with you subscribe via Promise#then

Promise.sync yield

GROOVY

```groovy
given:
Promise p = Promise.sync {
  println 'sync'
  'sync'
}

when:
ExecResult result = yield { p }

then:
result.value == 'sync'
```

Promises need to execute in Ratpack managed thread.

ExecHarness provides Ratpack managed threads as do RatpackServers of all varieties.

Promise.sync run

GROOVY

```
p.then { String s ->
  println 'then'
  assert s == 'sync'
}
```

Promise.value

GROOVY

```
given:
Promise p = Promise.value('value')  1

when:
ExecResult result = yield { p }

then:
result.value == 'value'
```

1 Promise.value creates promise from an already available value, unlike Promise.sync which will wait until the promise is subscribed in order to generate the value.

Blocking.get()

GROOVY

```
given:
Promise p = Blocking.get {
  'from blocking'
}

when:
ExecResult result = yield { p }

then:
result.value == 'from blocking'
```

Blocking runs on different threadpool

GROOVY

```groovy
given:
Closure getCurrentThreadName = {
  return Thread.currentThread().name.split('-')[1]
}

and:
Promise p = Blocking.get {
  Thread.sleep(1000)
  getCurrentThreadName()
} map { String nameFromBlocking ->
  String name = getCurrentThreadName()
  return [nameFromBlocking, name].join(' -> ')
}

when:
ExecResult result = yield { p }

then:
result.value == 'blocking -> compute'
```

Promise.async

GROOVY

```groovy
Thread externalThirdPartyAsyncLibraryWithCallback(Closure callback) {
  Thread.start {
    println 'Thread started'
    (1..5).each { i ->
      println(i)
      sleep(1000)
    }

    callback('async computation complete')
    println 'Thread finished'
  }
}

given:
Promise p = Promise.async { Downstream downstream ->
  println 'async start'
  externalThirdPartyAsyncLibraryWithCallback(
    downstream.&success
  )
  println 'async end'
}

when:
ExecResult result = yield { p }

then:
result.value == 'async computation complete'
```

You can think of Operations as a `Promise<Void>` , they don't share a common type but there are ways to switch back and forth betwteen Promises and Operations

Operation.of

GROOVY

```
Operation.of {  1
  println 'hello from operation'
}.then {  2
  println 'nothing returned from operation'
}
```

1 Factory to queue up an Operation

2 Note that Operations don’t return anything so there is nothing to receive in the subscriber

Operation to Promise

GROOVY

```
Promise<Void> p = Operation.of {
  println 'hello from operation'
}.promise()  1

p.map { v ->
  assert v == null  2
  println "v is void $v"
  "promise"
}.then { String msg ->  3
  println "We transformed from operation to $msg"
}
```

1 Invoke Operation#promise to create a Promise<Void>

2 See that we get null

3 Note that we can still work with this transformed Promise

Promise to Operation

GROOVY

```
Promise.value("foo")
  .operation { String msg ->  1
    println "found value $msg"
  }
  .then {
    println "no value emitted from operation"
    assert it == null  2
  }
```

1 We can turn a Promise into an Operation, however note that we still get the previous Promise value

2 See that Operation doesn’t return anything

17.1. Anatomy of a Promise

GROOVY

```
Promise.sync {
  return 'hello'
}.map { s ->
  s.toUpperCase()
}.then { s ->
  ctx.render(s)
}
```

As the methods `sync`, `map`, `then` are invoked, execution segments get queued.

GROOVY

```
[{Promise.sync { return 'hello' }.map { s -> s.toUpperCase() }.then { s ->
ctx.render(s) }}]
           |                       |                            |
           |                       |                            |
           v                       |                            |
[{}, { return 'hello' }]           v                            |
[{}, { return 'hello' }, { s -> s.toUpperCase() }]              v
[{}, { return 'hello' }, { s -> s.toUpperCase() }, { s -> ctx.render(s) }]
```

The output from the first promise is then used as input for the second segment, et cetera.

GROOVY

```
Promise.sync {
  return 'hello'
}.flatMap { s ->
  Promise.sync {
    s.toUpperCase()
  }
}.then { s ->
  ctx.render(s)
}
```

GROOVY

```
[{Promise.sync { return 'hello' }.flatMap { s -> Promise.sync { s.toUpperCase() }
}.then { s -> ctx.render(s) }}]
           |                       |                                                  |
           |                       |                                                  |
           v                       |                                                  |
[{}, { return 'hello' }]           v                                                  |
[{}, { return 'hello' }, { s -> Promise.sync { s -> s.toUpperCase() }}]               v
[{}, { return 'hello' }, { s -> Promise.sync { s -> s.toUpperCase() }}, { s ->
ctx.render(s)}]
                                         |
                                         |_____________________________________
                                                                              |
                                                                              v
[{}, { return 'hello' }, { s -> Promise.sync { s -> s.toUpperCase() }}, { s ->
s.toUpperCase() }, { s -> ctx.render(s)}]
```

Flatmap will queue up the promise, if you use map instead it just passes the promise to the next execution segment in the queue.

Handling errors

Exceptions can be thrown from Promises

Exception thrown from Promise

GROOVY

```groovy
given:
Promise p = Promise.sync {
  throw new Exception("oh no")
}

when:
yield { p }.valueOrThrow

then:
thrown Exception
```

But we can handle it and short circuit

Promise#onError

GROOVY

```groovy
when:
p = Promise.sync {
  throw new Exception("oh no")
}.onError { Exception e ->
  println e.message
}.map {
  'map'
}

and:
def value = yield { p }.valueOrThrow

then:
notThrown Exception
value != 'map'
value == null
```

Or we can handle and continue processing

Promise#mapError

GROOVY

```
given:
Promise p = Promise.sync {
  throw new Exception("oh no")
} mapError { Throwable t ->
  'default value'
} map({ String s -> s.toUpperCase()} as Function)

when:
String value = yield { p }.valueOrThrow

then:
notThrown Exception
value == 'DEFAULT VALUE'
```

Promises are immutable, methods like `Promise#map` always return new promises

Promises are immutable

GROOVY

```
given:
boolean exception = false

when:
Promise p = Promise.sync { throw new Exception() }

p.onError {  1
  exception = true
  println 'oops'
}
p.map { 'map' }  2

and:
yield { p }.valueOrThrow

then:
exception == false
thrown Exception
```

1 Promise#onError returns a *new* Promise

2 Promise#map returns a *new* Promise

Promise api allows you to chain promise manipulation

GROOVY

```
when:
p = Promise.sync { throw new Exception() }
  .onError {  1
  exception = true
  println 'oops'
}
.map { 'map' }

and:
yield { p }.valueOrThrow

then:
exception
notThrown Exception
```

1 Promise#onError returns a new Promise

Error mapping also available in flatmap flavor Promise#flatmapError.

Promise#map

GROOVY

```
Promise.value(3)
  .map { int i -> 'A' * i }  1
  .then { String s ->
    println 'from map'
    assert s == 'AAA'
  }
```

1 Promise#map runs on compute thread, don’t block here

Promise#flatMap

GROOVY

```
Promise.value(3)
  .flatMap { int i ->  1
    Blocking.get { 'A' * i }  2
  }.then { String s ->  3
    println 'from flatMap blocking'
    assert s == 'AAA'
  }
```

1 Promise#flatMap runs on compute thread, don’t block here

2 Since Blocking#get returns a Promise, need to use Promise#flatMap in order to "unpack" the nested promise and continue working with the value as in <3>

Promise#blockingMap

GROOVY

```groovy
Promise.value(3)
  .blockingMap { int i ->  1
    'A' * i
  }.then { String s ->
    println 'from blockingMap'
    assert s == 'AAA'
  }
```

1 A convenience method for executing blocking code inline without having to do `Promise#flatMap { Blocking.get {} }` as in the previous example

Promise#flatMap with async

GROOVY

```groovy
Promise.value(3)
  .flatMap { int i ->
    Promise.async { Downstream d ->  1
      Thread.start {
        println 'starting thread'
        Thread.sleep(100)
        d.success('A' * i)
      }
    }
  }.then { String s ->
    println 'from flatMap async Thread'
    assert s == 'AAA'
  }
```

1 Integrating with an externally managed threadpool/async library

Promise#left/right

GROOVY

```groovy
given:
List<Map<String, Object>> users = [[name: 'danny', interests: ['dancing', 'cooking']]]
Map<String, List<String>> interests = [dancing: ['fun'], cooking: ['a great catch'],
kotlin: ['with it']]

expect:
run {
  Blocking.get {
    users.find { Map user -> user.name == 'danny' }  1
  }.right { Map user ->  2
    user.interests.collectMany { interest ->  3
      interests[interest]
    }
  }.map { Pair<Map, List<String>> pair ->  4
    "${pair.left.name} is ${pair.right.join(' and is ')}"
  }.then { String msg ->
    assert msg == 'danny is fun and is a great catch'
  }
}
```

1 Imagine some blocking jdbc lookup by name

2 Promise#right takes the result of the previous promise and creates a tuple like graph datastructure called Pair. The value returned from the closure/lambda is then pushed to the "right" position of the Pair

3 Imagine some in memory lookup for interests by user that we just looked up from <1>

4 The result from the previous call is of type `Pair<A, B>` where `A` is the result of the `Blocking.get{}` call and `B` is ther result of `Promise#right` call

Pairs are handy when working with small number of arguments.

You can also nest Pairs, you can have something like `Pair<Pair<A, B>, C>` but this quickly becomes hard to track.

If only Java had tuple support :(

This is also available in flatMap flavor Promise#flatLeft/flatRight

- ParallelBatch, SerialBatch You'll often want to take a list of promises and transform them into a list of resolved values. ParallelBatch and SerialBatch help achieve this.

GROOVY

```
given:
Random random = new Random(1)
Closure<Promise> getPrice = { int id ->  1
  Blocking.get {
   [id: id, price: random.nextInt(10)]
  }
}

expect:
run {
  Promise.value(50)
    .map { int i -> (1..i) }
    .flatMap { ids ->
      List<Promise> promises = ids.collect { int id -> getPrice(id) }  2
      batch(promises).yield()  3
    }.map { List<Map<String, Object>> results ->  4
      results.price.sum()
    }.map { int price ->
      "Total is \$${price}"
    }.then { String msg ->
      assert msg == 'Total is $238'
    }
}

where:

type        | batch  5
'serial'    | SerialBatch.&of
'parallel'  | ParallelBatch.&of
```

1 Simulate price lookup in a blocking manner

2 Use a Groovy range to generate a list of promises to lookup the price for the given id

3 Invoke either `SerialBatch.of` or `ParallelBatch.of` and yield a `Promise<List<Map<String, Object>>>`

4 Use the handy value as a single list

5 Spock datatable for making `batch` pluggable

Batch is great for when you have `List<Promise<A>>` but want to work with `Promise<List<A>>` in subsequent calculations.

Forking execution

Promise#fork

GROOVY

```groovy
given:
List list = []
Closure addToList = { 1
  println it
  list << it
  it
}

expect:
run {
  Blocking.get {
    addToList('foo') 2
  }.right(
    Promise.async { Downstream d ->
      d.success(addToList('bar')) 3
    }.fork() 4
  ).map { Pair<String, String> pair ->
    pair.left + pair.right
  }.then { String msg ->
    assert list == ['bar', 'foo']  5
    assert msg == 'foobar'
  }
}
```

1 Convenience closure for printing and adding to list

2 Add foo in a blocking manner

3 Add bar in async manner

4 Fork the bar async promise

5 Assert that bar was entered before foo

When forking Promises, they execute immediately!

"Flow control"

Promise#mapIf

GROOVY

```groovy
given:
Closure fizzbuzzer = { candidate ->  1
  Promise.sync {
      println "${LocalDateTime.now()} EXECUTING FIZZBUZZ for $candidate"
      candidate
    }
    .mapIf({i -> i instanceof Number && i % 3 == 0 && i % 5 == 0}, { 'fizzbuzz' })  2
    .mapIf({i -> i instanceof Number && i % 3 == 0}, { 'fizz' })  2
    .mapIf({i -> i instanceof Number && i % 5 == 0}, { 'buzz' })  2
}

expect:
run {
  Promise.value(15)
    .map { 1..it }
    .flatMap { range ->
      ParallelBatch.of(range.collect { fizzbuzzer(it) }).yield()  3
    }.then { list ->  4
      assert list == [
        1, 2, 'fizz', 4, 'buzz',
        'fizz', 7, 8, 'fizz', 'buzz',
        11, 'fizz', 13, 14, 'fizzbuzz'
      ]
    }
}
```

1 Convenience closure

2 Example of using `Promise#mapIf(Predicate, Function)`, only maps if predicate passes

3 Execute these promises in parallel

4 Assert that our list is fizzbuzzed correctly

Also available in flatMap variety Promise#flatMapIf

Other useful methods:

- Promise#onNull
- Promise#route

Promise#route is a terminating call, the rest of the promise chain is no longer executed!!

Promise#route(Predicate, Action)

GROOVY

```
Promise.sync(LocalDateTime.&now)
  .route({ldt -> (ldt.get(ChronoField.MILLI_OF_SECOND) & 1) == 0}, { ldt ->  1
    println "TERMINATING: $ldt is even"
  }).map { ldt ->
    println "MAPPING LocalDateTime to String"
    "$ldt is odd"
  }.then { String msg ->
    println "SUCCESS: $msg"
  }
```

1 Note that we terminate the promise chain here if predicate passes

Throttling Promises

Throttle acts as a semaphore only allowing n number of promises to run in parallel.

Promise#throttled

GROOVY

```
given:
Throttle throttle = Throttle.ofSize(3)  1
Closure fizzbuzzer = { candidate ->
  Blocking.get {
      Thread.sleep(2000)
      println "${LocalDateTime.now()} EXECUTING FIZZBUZZ for $candidate"
      candidate
    }
    .mapIf({i -> i instanceof Number && i % 3 == 0 && i % 5 == 0}, { 'fizzbuzz' })
    .mapIf({i -> i instanceof Number && i % 3 == 0}, { 'fizz' })
    .mapIf({i -> i instanceof Number && i % 5 == 0}, { 'buzz' })
    .throttled(throttle)  2
}

expect:
run {
  Promise.value(15)
    .map { 1..it }
    .flatMap { range ->
      ParallelBatch.of(range.collect { fizzbuzzer(it) }).yield()
    }
    .then { list ->
      assert list == [
        1, 2, 'fizz', 4, 'buzz',
        'fizz', 7, 8, 'fizz', 'buzz',
        11, 'fizz', 13, 14, 'fizzbuzz'
      ]
    }
}
```

1 Declare a throttle of size 3

2 Make sure that the promise we submit to the ParallelBatch is throttled

Sample output

```
2017-05-30T07:42:48.294 EXECUTING FIZZBUZZ for 4
2017-05-30T07:42:48.308 EXECUTING FIZZBUZZ for 5
2017-05-30T07:42:48.294 EXECUTING FIZZBUZZ for 1
2017-05-30T07:42:50.367 EXECUTING FIZZBUZZ for 10
2017-05-30T07:42:50.367 EXECUTING FIZZBUZZ for 12
2017-05-30T07:42:50.367 EXECUTING FIZZBUZZ for 2
2017-05-30T07:42:52.371 EXECUTING FIZZBUZZ for 13
2017-05-30T07:42:52.371 EXECUTING FIZZBUZZ for 9
2017-05-30T07:42:52.371 EXECUTING FIZZBUZZ for 3
2017-05-30T07:42:54.374 EXECUTING FIZZBUZZ for 7
2017-05-30T07:42:54.374 EXECUTING FIZZBUZZ for 6
2017-05-30T07:42:54.374 EXECUTING FIZZBUZZ for 11
2017-05-30T07:42:56.377 EXECUTING FIZZBUZZ for 8
2017-05-30T07:42:56.377 EXECUTING FIZZBUZZ for 15
2017-05-30T07:42:56.377 EXECUTING FIZZBUZZ for 14
```

You can see there is a tight grouping of 3 per time period

Spying

If you wish to observe items as they get processed in the promise chain, you can make use of Promise#wiretap

Promise#wiretap

GROOVY

```
Promise.sync(LocalDateTime.&now)
  .map { ldt -> ldt.getDayOfWeek() }
  .wiretap { Result<DayOfWeek> r ->  1
    println "Found: ${r.value.getDisplayName(TextStyle.FULL_STANDALONE, Locale.KOREA)}"
 2
  }.then { DayOfWeek dow ->  3
    println "Today is $dow"
  }
```

1 Invoke wiretap method

2 Note that we have to invoke Result#getValue in order to get the value produced from the previous Promise

3 Note that the next processor gets the same result as the wiretap

18. Best practices

- Avoid multiple then blocks
- Try to linearize/flatten data flow

Last updated 2017-05-31 09:33:19 EDT