



RxJava, Ratpack, Couchbase

Laurent Doguin
Couchbase Developer Advocate
@ldoguin



Laurent Doguin

Couchbase Developer Advocate

@ldoguin | laurent.doguin@couchbase.com





Ratpack ?

Not an actor/singer club





Netty based, fullstack, non-blocking, web framework

In a Nutshell...



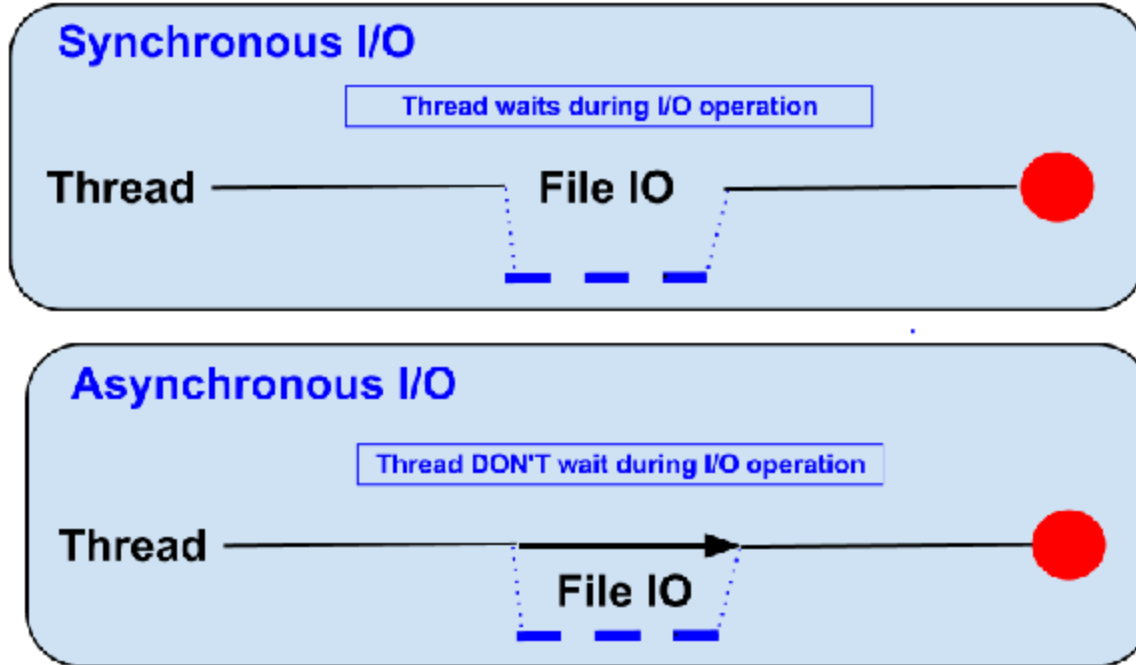
- Asynchronous
- Non-Blocking
- Build on top of Netty
- Unopinionated
- Think about NodeJS or Vert.X
- Promises
- Java 8
- Top Notch Gradle integration
- Strong Groovy Support
- Great for I/O Bound applications

Synchronous/Blocking?



- Wait until the current line of code is executed before moving to the next one
- Why would that be bad, isn't this how most things work?
- What happens when you wait
- Great for I/O Bound applications

Sync, blocking V.S. Async, non-blocking



- Core of Ratpack
- This is where the magic happens
- `RatpackServer.start(`
 - `server -> server.handlers(`
 - `chain -> chain.all(`
 - `ctx -> ctx.render("Hello World")`
 - `)`
 - `)`
- `);`



- GUICE based Modules system
- Can be switched to others
 - Spring Boot support
- Module can be as low level as DI Framework
- Think of it like decorations

Why Ratpack?



- If you spend too much time on I/O
- Code in an async non-blocking way with a deterministic/synchronous execution style



RxJava



- Blocking is Evil
- Async is good, Better when
 - Reactive
 - Parallelizable
 - Composable
 - Readable

But I can do this already



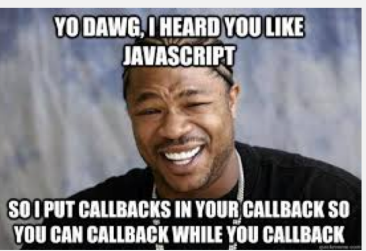
- Callbacks?
 - No composition
 - Callback Hell
- Futures<T>?
 - 'lol'
 - Too easy to block
 - Hard to compose

```
function foo() {
  console.log('foo');
  function bar() {
    console.log('bar');
  }
  bar();
}
```

```
function foo() {
  console.log('foo');
  function bar() {
    console.log('bar');
  }
  bar();
}
```

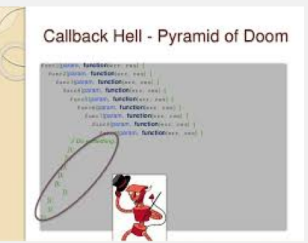
```
function foo() {
  console.log('foo');
  function bar() {
    console.log('bar');
  }
  bar();
}
```

```
function foo() {
  console.log('foo');
  function bar() {
    console.log('bar');
  }
  bar();
}
```



```
var async = require('async');
User.findById(function(err, user) {
  if (err) return handleError(err);
  User.findOne(function(err, friend) {
    if (err) return handleError(err);
    async.parallel(function(err, done) {
      async.waterfall([
        function(err, post) {
          console.log('post id: ' + post.id);
        },
        function(err, comments) {
          console.log('comments: ' + comments);
        }
      ], function(err) {
        if (err) return handleError(err);
        done();
      });
    });
  });
});
```

```
function foo() {
  console.log('foo');
  function bar() {
    console.log('bar');
  }
  bar();
}
```



```
function foo() {
  console.log('foo');
  function bar() {
    console.log('bar');
  }
  bar();
}
```

```
function foo() {
  console.log('foo');
  function bar() {
    console.log('bar');
  }
  bar();
}
```

```
pan.pourWater(function() {
  range.bringToBoil(function() {
    range.lowerHeat(function() {
      pan.adjustFire(function() {
        setTimeout(function() {
          range.turnOff();
        }, 15 * 60 * 1000);
      });
    });
  });
});
```

```
function foo() {
  console.log('foo');
  function bar() {
    console.log('bar');
  }
  bar();
}
```

```
1. app.get('/some/resources', function (req, res) {
2.   db.query('SELECT A ...', function (err, a) {
3.     if (err) return res.end(err);
4.
5.     db.query('SELECT B ... WHERE b = ' + a, function (err, b) {
6.       if (err) return res.end(err);
7.
8.       db.query('SELECT C ... WHERE c = ' + b, function (err, c) {
9.         if (err) return res.end(err);
10.
11.         db.query('SELECT D ... WHERE d = ' + c, function (err, d) {
12.           if (err) return res.end(err);
13.
14.           res.end(d);
15.         });
16.       });
17.     });
18.   });
19. });
```

● FIRST-CLASS COROUTINES
● SUSPEND EXECUTION CONTEXT
● SMALL API
● NOT AVAILABLE YET

```
var async = require('async');
User.findById(function(err, user) {
  if (err) return handleError(err);
  User.findOne(function(err, friend) {
    if (err) return handleError(err);
    async.parallel(function(err, done) {
      async.waterfall([
        function(err, post) {
          console.log('post id: ' + post.id);
        },
        function(err, comments) {
          console.log('comments: ' + comments);
        }
      ], function(err) {
        if (err) return handleError(err);
        done();
      });
    });
  });
});
```

```
1. setTimeout(function() {
2.   console.log('this is a callback 1');
3. }, 1000);
4. console.log('this is a callback 2');
5. setTimeout(function() {
6.   console.log('this is a callback 3');
7. }, 1000);
8. console.log('this is a callback 4');
9. }, 1000);
```

```
CALLBACKS - CALLBACK HELL
var foo = function(callback) {
  console.log('foo');
  if (callback) callback();
};
foo(function() {
  console.log('bar');
});
```

```
1. var floppy = require('floppy');
2. floppy.load('disk1', function (data) {
3.   floppy.prompt('Please insert disk 2', function () {
4.     floppy.load('disk2', function (data2) {
5.       floppy.prompt('Please insert disk 3', function () {
6.         floppy.load('disk3', function (data3) {
7.           // If disk 3 would have existed in 1995
8.           console.log('done');
9.         });
10.      });
11.    });
12.  });
13. });
```

```
function foo() {
  console.log('foo');
  function bar() {
    console.log('bar');
  }
  bar();
}
```

```
function foo() {
  console.log('foo');
  function bar() {
    console.log('bar');
  }
  bar();
}
```

- Netflix OpenSource
- From Iterator - Iterable (Pull)
- To Observable - Observer(Push)

- Allow to Compose
- Asynchronous code
- Based on events
- Using Observable Sequences



NETFLIX
OSS

Why Reactive?



- New challenges
 - React to user load
 - React to failure
 - Be responsive under load and failure
- Need new solutions
 - Decoupled, event-driven architectures
 - Optimal resource utilization

Why Reactive?





RxJava 101

A Gentle Introduction

RxJava: Introduction



Java implementation for Reactive Extensions

- <https://github.com/ReactiveX>
- A library to compose **asynchronous** and **event-driven** programs through observable sequences.

	single	multiple
sync	T	Iterable<T>
async	Future<T>	Observable<T>



- Observables are the duals of Iterables
- They describe both Latency and Error side effects.

event	Iterable<T> (pull)	Observable<T> (push)
data retrieval	T next()	onNext(T)
error discovery	throws Exception	onError(Exception)
completion	returns	onCompleted()

The Observer subscribes

- and receives events.

A cold Observable

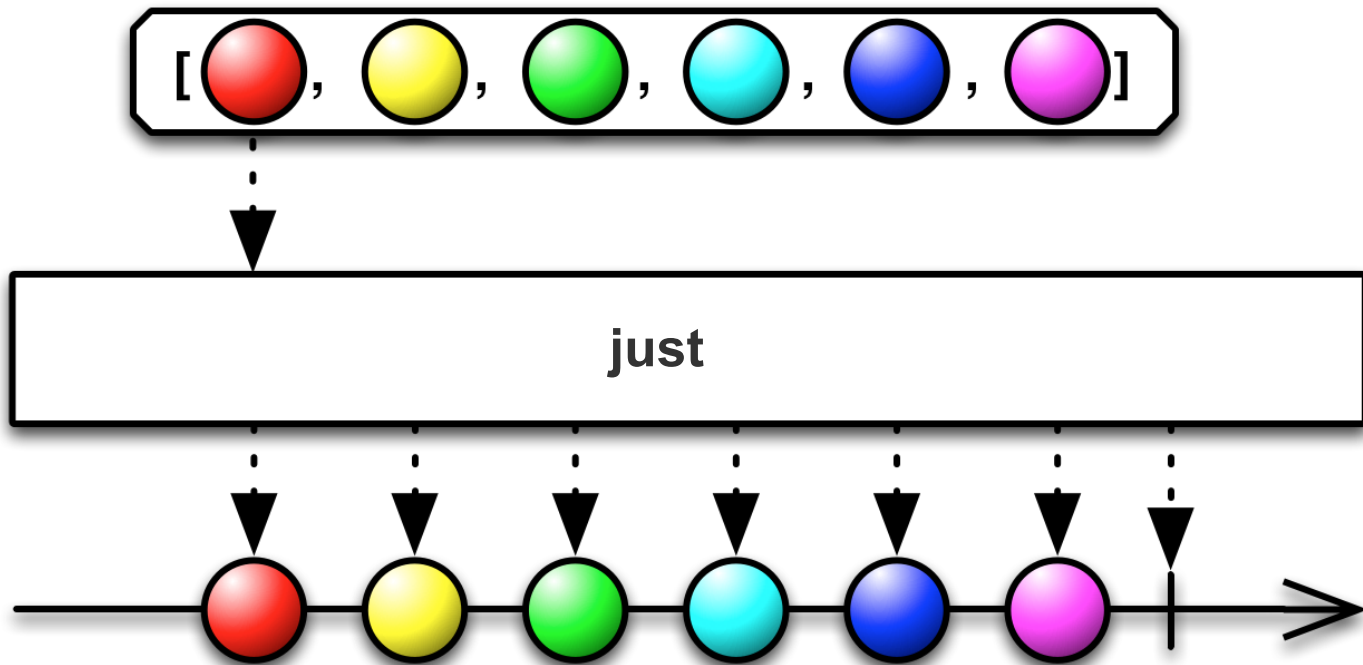
- starts when subscribed.

onNext can be called

- 0..N times

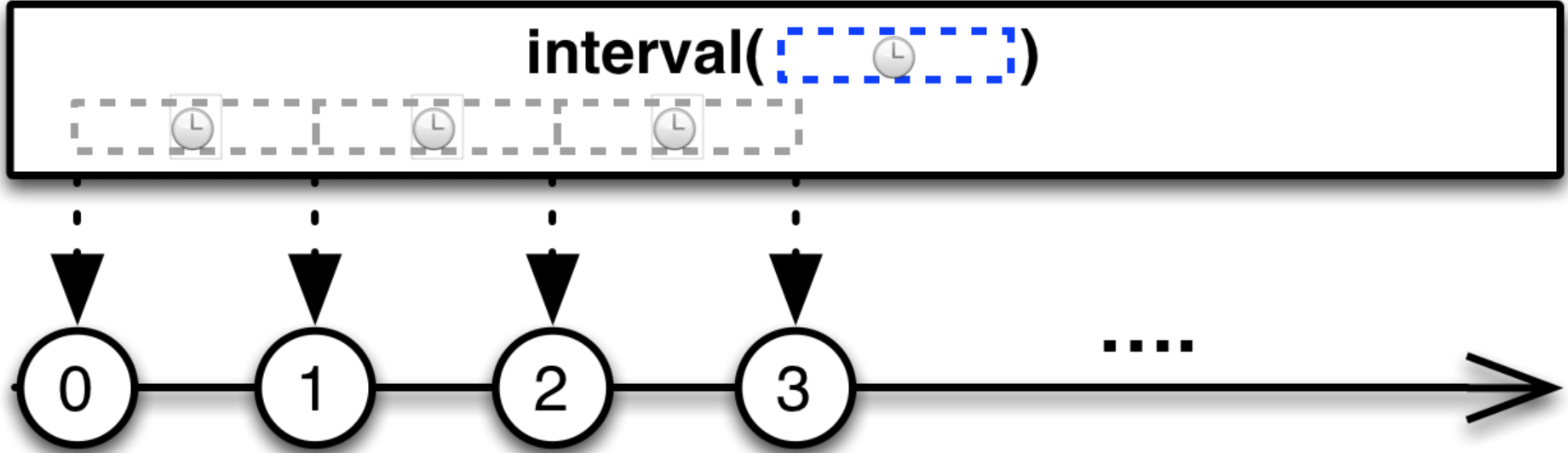
```
1 bucket
2   .async()
3   .get("doc")
4   .subscribe(new Observer<JsonDocument>() {
5       @Override
6       public void onCompleted() {
7           System.out.println("Done");
8       }
9
10      @Override
11      public void onError(Throwable throwable) {
12          throwable.printStackTrace();
13      }
14
15      @Override
16      public void onNext(JsonDocument doc) {
17          System.out.println("Found: " + doc);
18      }
19  });
```

RxJava: Creating Observables



```
1 Observable
2     .just("A", "B", "C")
3     .subscribe(new Action1<String>() {
4         @Override
5         public void call(String s) {
6             System.out.println("Got: " + s);
7         }
8     });
```

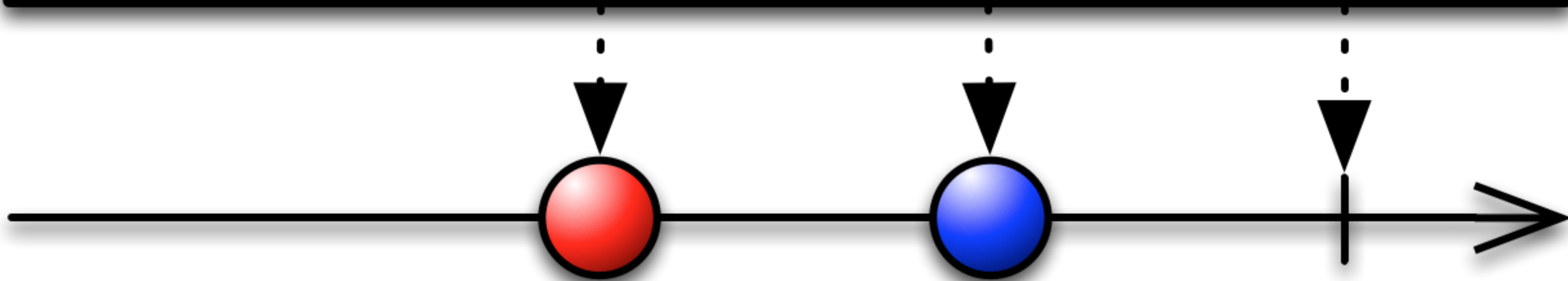
RxJava: Creating Observables





```
1 // 0 ..... 1 ..... 2 .....  
2 Observable  
3     .interval(2, TimeUnit.SECONDS)  
4     .subscribe(System.out::println);
```

```
create { onNext  ; onNext  ; onComplete }
```

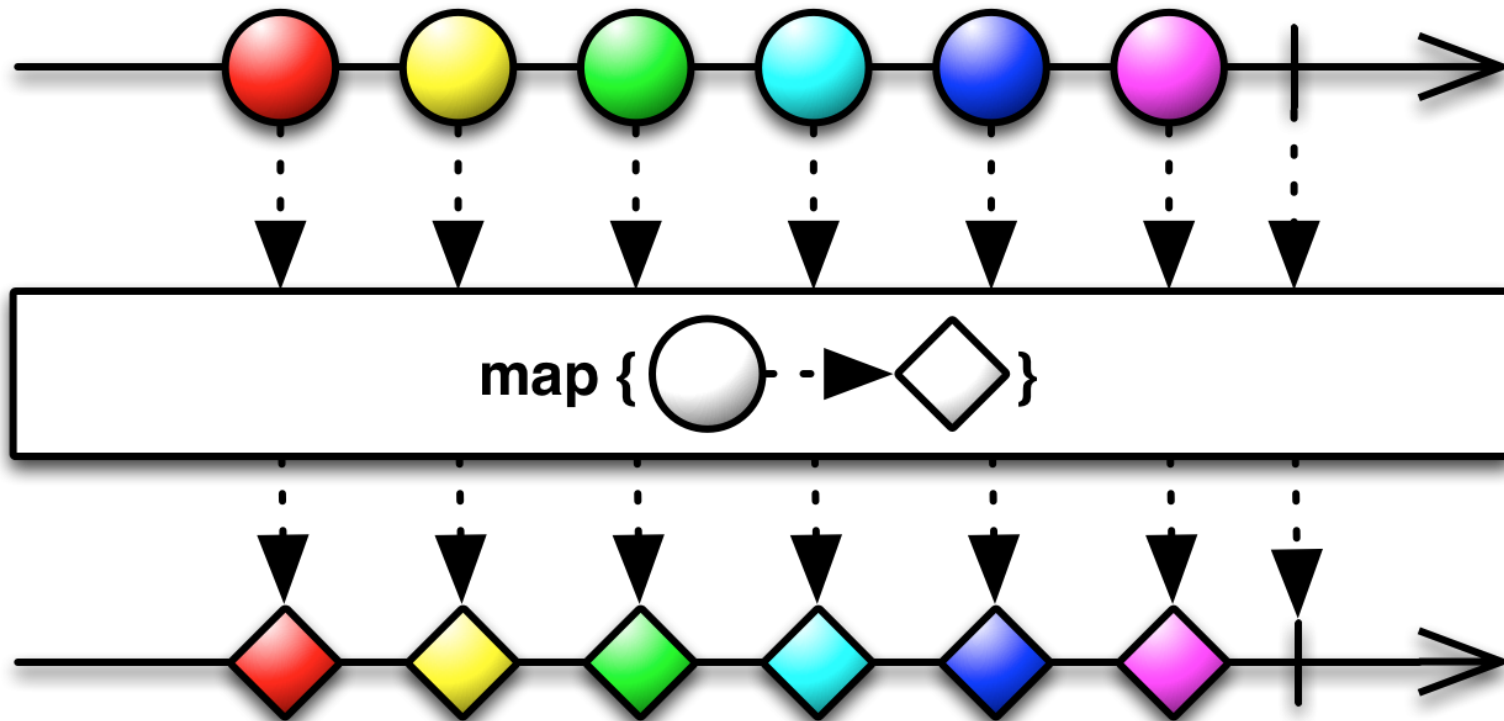


RxJava: Creating Observables



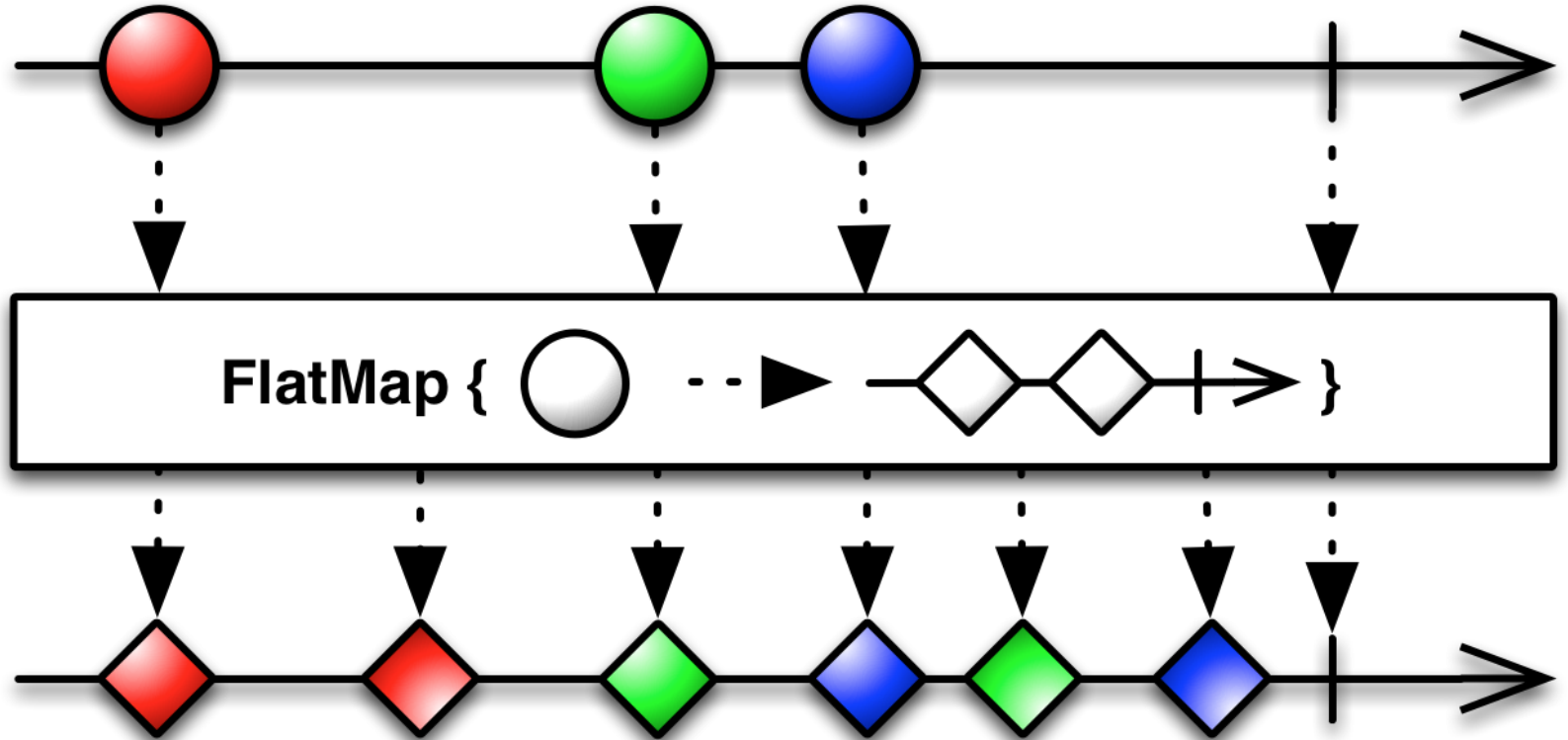
```
1  Observable.create(new Observable.OnSubscribe<String>() {
2      @Override
3      public void call(Subscriber<? super String> subscriber) {
4          try {
5              subscriber.onNext("Hello Subscriber!");
6              subscriber.onCompleted();
7          } catch (Exception ex) {
8              subscriber.onError(ex);
9          }
10     }
11 }).subscribe();
```


RxJava: Transforming Observables



```
1 bucket
2     .async()
3     .get("doc")
4     .map(doc -> doc.content())
5     .subscribe(new Action1<JsonObject>() {
6         @Override
7         public void call(JsonObject content) {
8             System.out.println("Found: " + content);
9         }
10    });
```

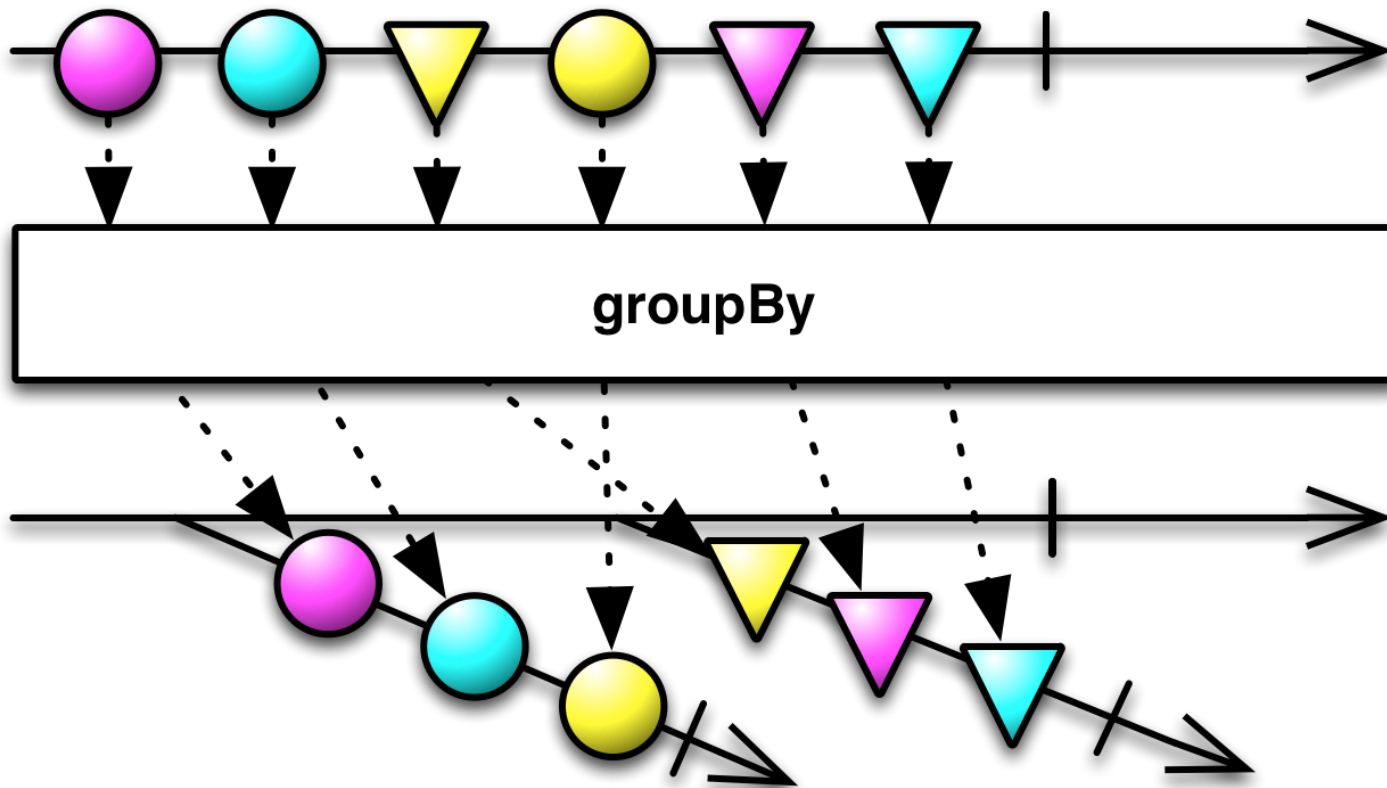
RxJava: Transforming Observables





```
1 bucket
2     .query(ViewQuery.from("beers", "by_name").limit(100)) // query
3     .flatMap(AsyncViewResult::rows) // stream each row
4     .flatMap(AsyncViewRow::document) // grab doc for each row
5     .filter(doc -> doc.content().getDouble("abv") > 5.0) // filter beer by abv
6     .count() // count all filtered beers
7     .timeout(10, TimeUnit.SECONDS) // overall timeout
8     .subscribe(System.out::println); // print
```

RxJava: Transforming Observables

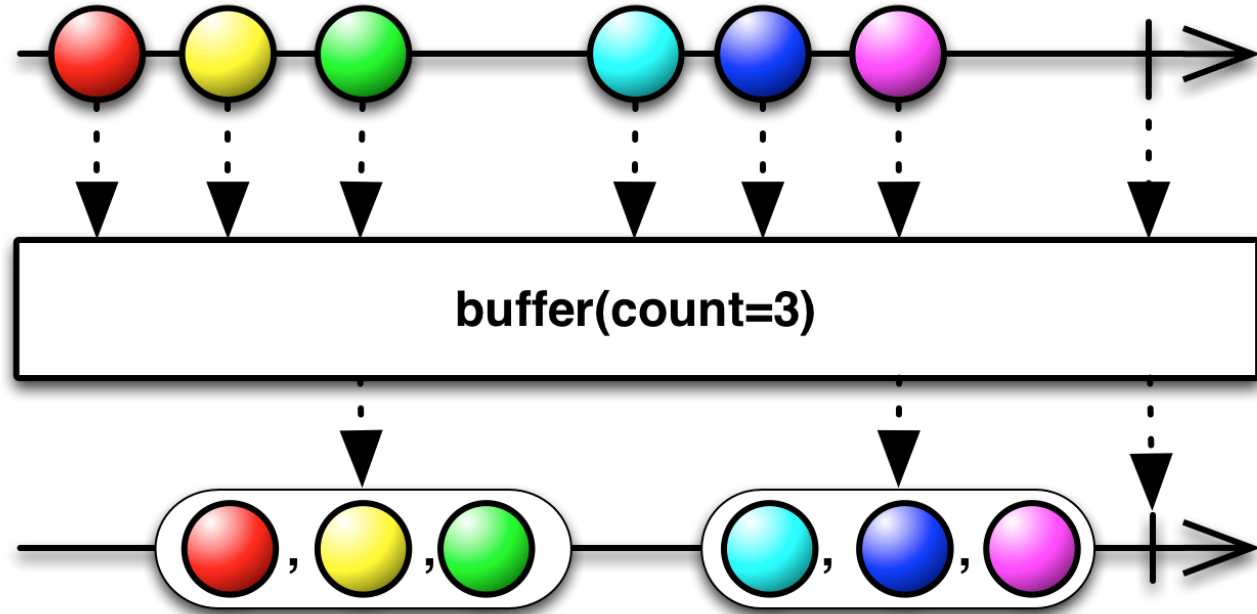


RxJava: Transforming Observables



```
1 bucket
2     .query(select("*").from("beer-sample"))
3     .flatMap(AsyncQueryResult::rows)
4     .groupBy(result -> result.value().getString("type"))
5     .subscribe(grouped ->
6         grouped
7             .count()
8             .subscribe(cnt -> System.out.println(grouped.getKey() + ": " + cnt)))
9
10 );
```

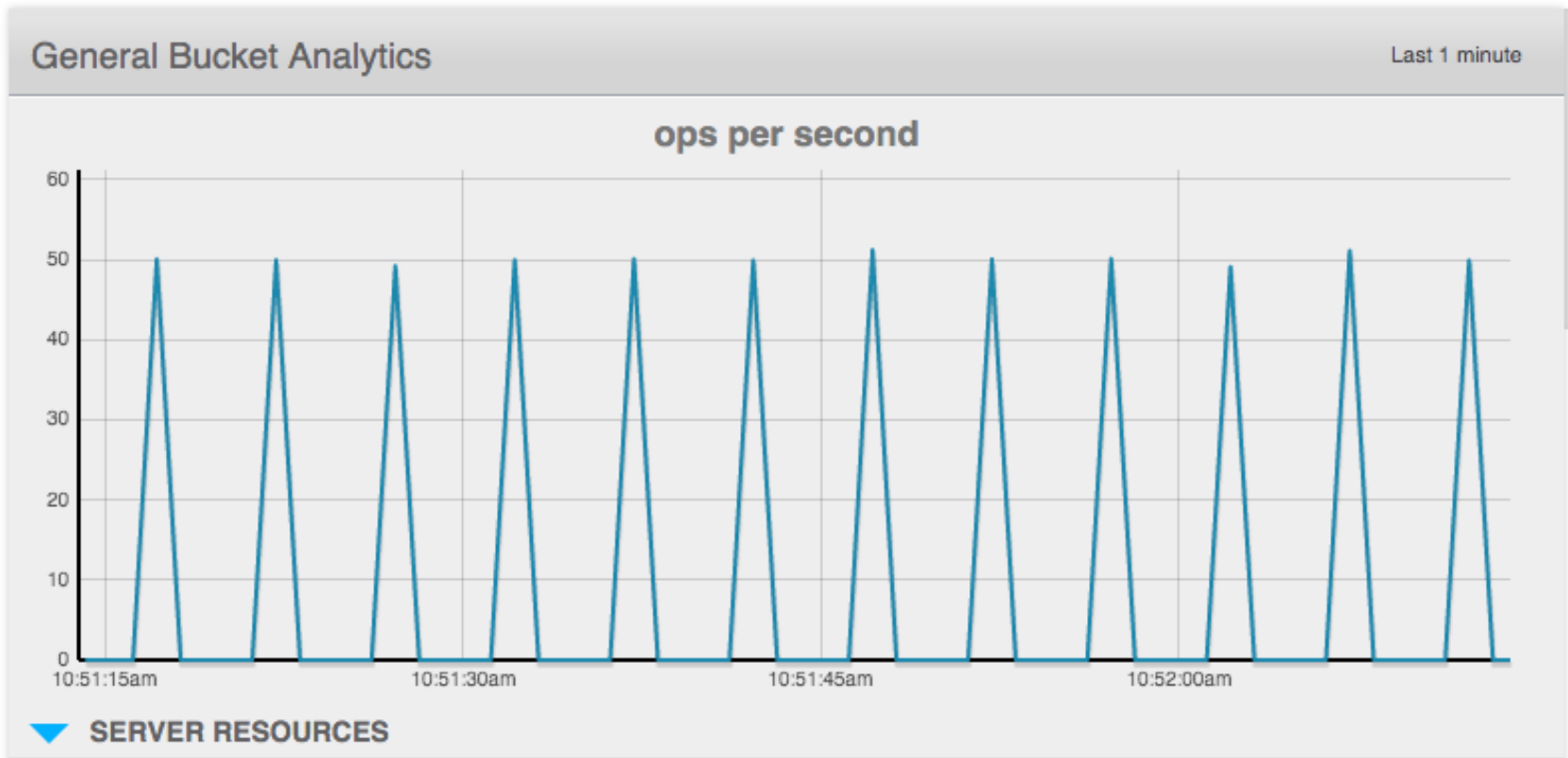
RxJava: Transforming Observables



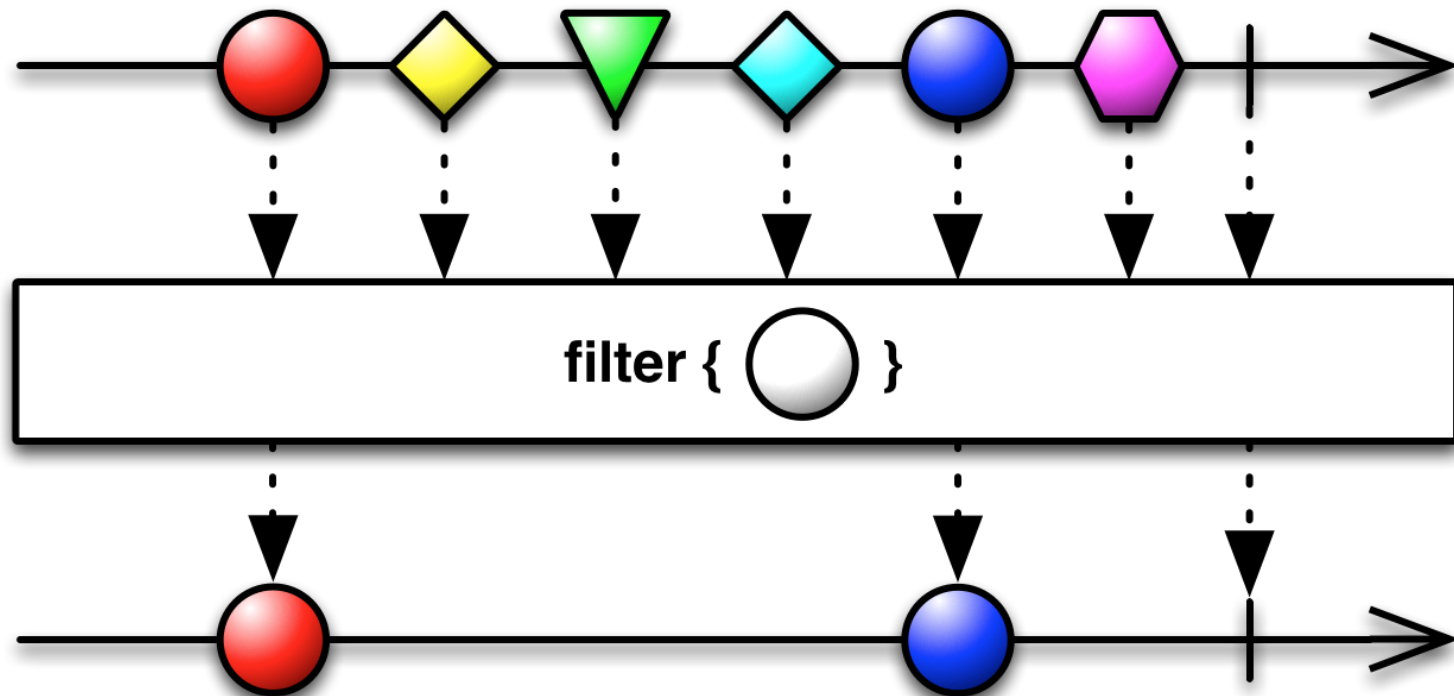


```
1 Observable
2     .interval(100, TimeUnit.MILLISECONDS)
3     .buffer(5, TimeUnit.SECONDS)
4     .flatMap(ids -> Observable.from(ids).map(i -> "id::" + i))
5     .flatMap(bucket.async()::get)
6     .toBlocking()
7     .last();
```


RxJava: Transforming Observables



RxJava: Filtering Observables

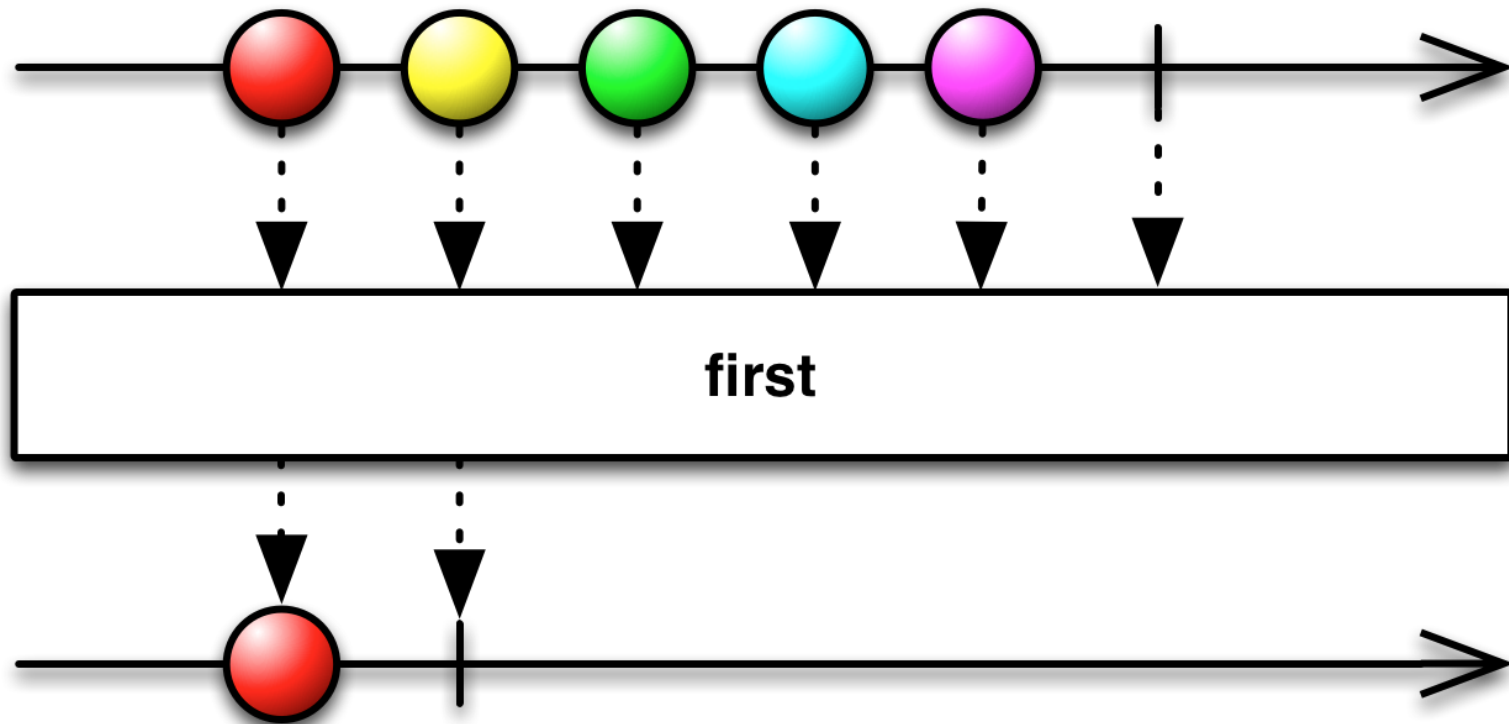


RxJava: Filtering Observables



```
1 Observable
2     .just("doc1", "doc2", "doc3")
3     .flatMap(new Func1<String, Observable<JsonDocument>>() {
4         @Override
5         public Observable<JsonDocument> call(String s) {
6             return bucket.async().get(s);
7         }
8     })
9     .filter(new Func1<JsonDocument, Boolean>() {
10        @Override
11        public Boolean call(JsonDocument document) {
12            return document.content().containsKey("foo");
13        }
14    })
15    .subscribe();
```

RxJava: Filtering Observables





```
1 bucket
2   .async()
3   .getFromReplica("doc", ReplicaMode.ALL)
4   .first()
5   .subscribe();
```





Example

Store, index and search files

The Application



- Using Ratpack as web framework
- Using Couchbase as database
- User can Upload a file
- User can list all the files
- User can search files
 - Using N1QL
 - Using fulltext search

Store and Search Files

You successfully uploaded /img/ZeppFirstChart.png!

Name:	Full Text Query:	N1QL Query:
<input type="text" value="Full Name"/>	<input type="text" value="Full Text Term"/>	<input type="text" value="Full Text Term"/>
File to upload:	<input type="button" value="Fulltext Search"/>	<input type="button" value="N1QL Ssearch"/>
<input type="button" value="Browse..."/> No file selected.		
<input type="button" value="Upload"/>		

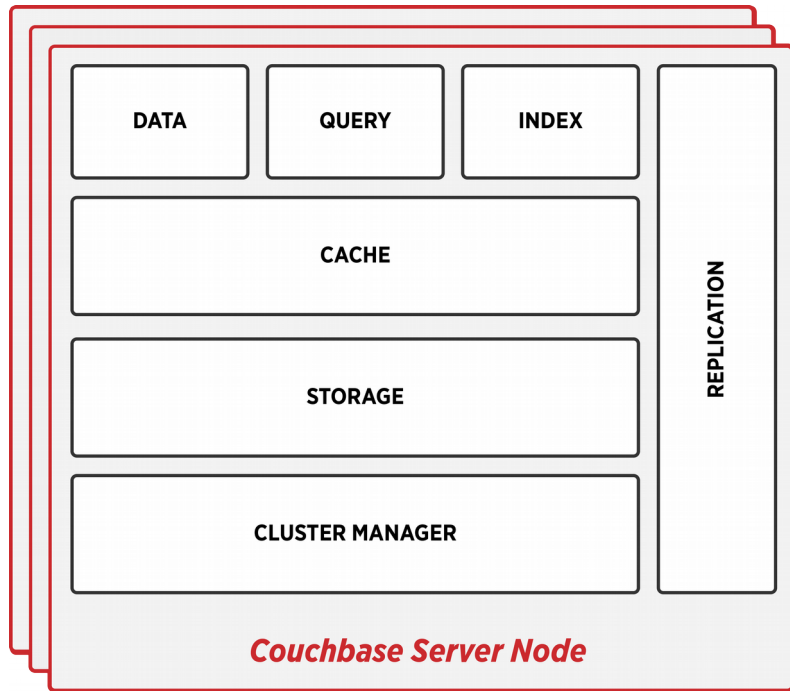
/img/ZeppFirstChart.png

/pdf/woot.pdf



About Couchbase

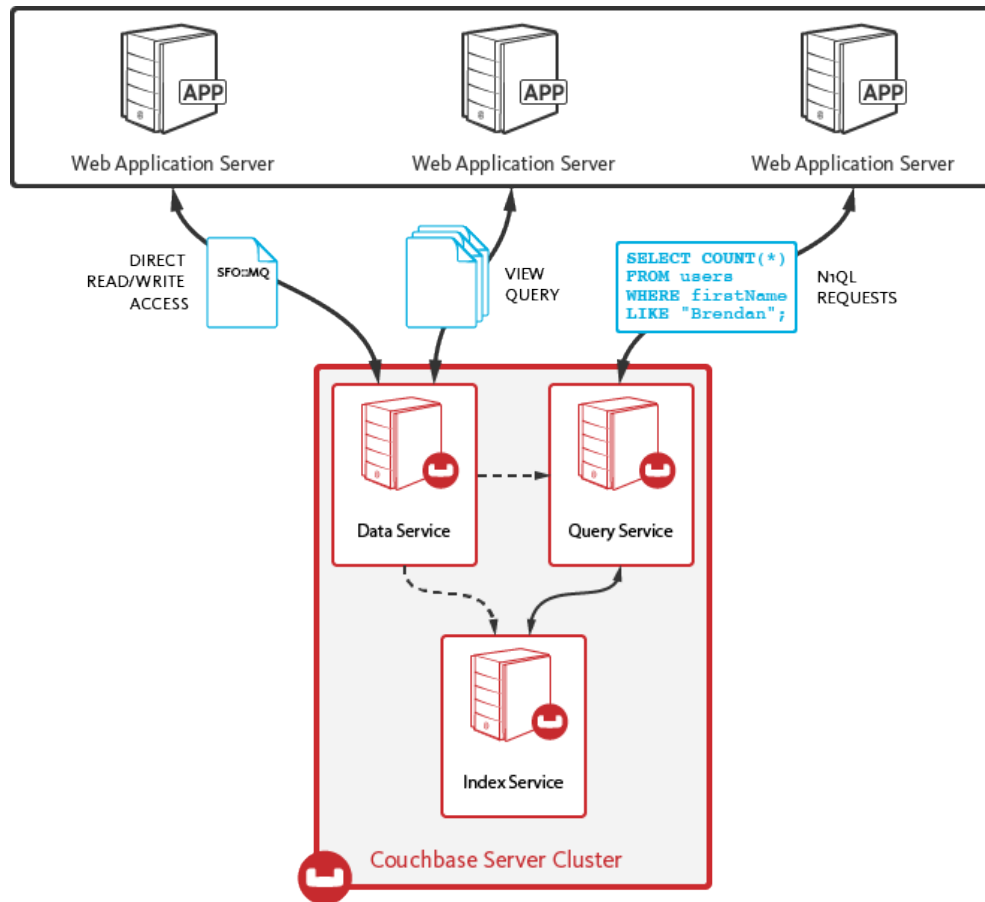
Couchbase Server - Single Node Architecture



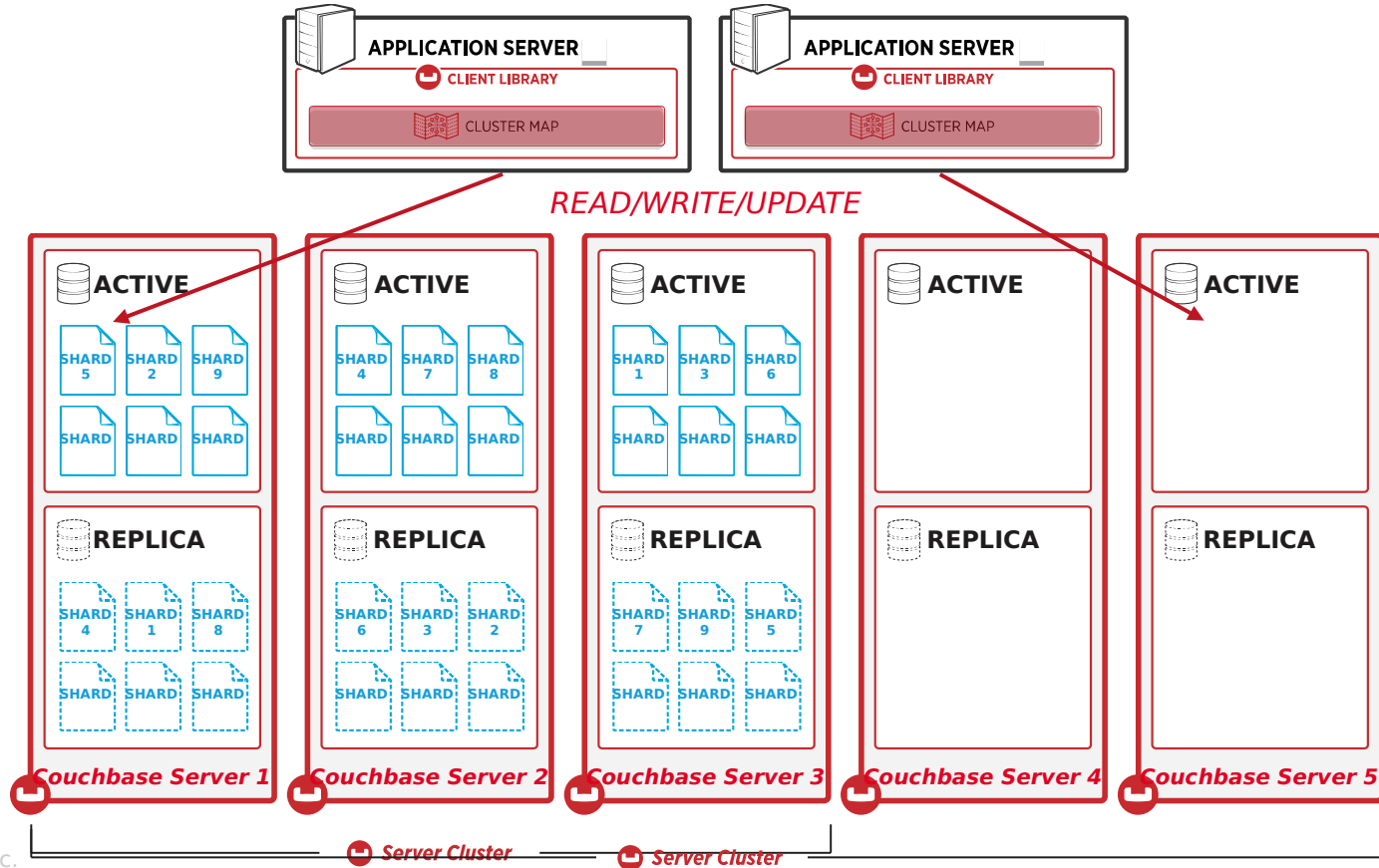
COUCHBASE SERVER CLUSTER

- **Data Service** – builds and maintains local view indexes
- **Indexing Engine** – builds and maintains Global Secondary Indexes
- **Query Engine** – plans, coordinates, and executes queries against either Global or Local view indexes
- **Cluster Manager** – configuration, heartbeat, statistics, RESTful Management interface

Application to Database Integration



Smart Connectivity with Built in Sharding and Replication



Accessing Data From Couchbase

Key access using Document ID

- Operations are extremely fast with consistent low latency
- Reads and writes are evenly distributed across Data Service nodes
- Data is cached in built-in Managed Caching layer and stored in persistent storage layer

Views using static queries

- Pre-computed complex Map-Reduce queries
- Incrementally updated to power analytics, reporting and dashboards
- Strong for complex custom aggregations

Queries using N1QL

- SQL-like : `SELECT * FROM WHERE, LIKE, GROUP, etc.,`
- `JOINS`
- Powerful Extensions (nest, unnest) for JSON to support nested and hierarchical data structures.
- Multiple access paths - Views and global secondary indexes
- ODBC/JDBC drivers available

Complex N1QL Query



```
public static List<Map<String, Object>> getAll(final Bucket bucket, String from,
String to) {
    String queryStr = "SELECT faa AS fromAirport, geo " +
        "FROM `" + bucket.name() + "` r" +
        "WHERE airportname = " + from +
        "UNION SELECT faa AS toAirport, geo " +
        "FROM `" + bucket.name() + "` r" +
        "WHERE airportname = " + to;
    ParameterizedN1qlQuery query = ParameterizedN1qlQuery.parameterized(queryStr,
        JSONArray.create().add(from).add(to));
    N1qlQueryResult queryResult = bucket.query(query);
    return extractResultOrThrow(queryResult);
}
```



Error Handling

with RxJava





- The network is unreliable
- Servers fail
- The SDK contains bugs

- Always specify timeouts and deal with them!
- The synchronous wrapper defines them for you all the time.



```
bucket.get("doc", Integer.MAX_VALUE, TimeUnit.SECONDS);
```

Timeouts: Complex Example



```
1 bucket
2     .query(ViewQuery.from("beers", "by_name").limit(100)) // query
3     .flatMap(AsyncViewResult::rows) // stream each row
4     .flatMap(AsyncViewRow::document) // grab doc for each row
5     .filter(doc -> doc.content().getDouble("abv") > 5.0) // filter beer by abv
6     .count() // count all filtered beers
7     .timeout(10, TimeUnit.SECONDS) // overall timeout
8     .subscribe(System.out::println); // print
```

Circuit Breakers



- monitor traffic
- open if errors happen
 - Latency
 - Throughput
 - Wrong results

close in a controlled

- fashion
- expose metrics



Backpressure



- Allows for coordinated flow control under stress conditions





- Things will go wrong, so better plan for it
- Do not trust integration points (including the SDK)
- Synchronous retry & fallback is too damn hard

- RxJava provides (almost) everything you need to
 - fallback
 - retry
 - fail fast



- Fail fast
 - Don't let your system get stuck
 - Shed load with backpressure
- Retry
 - immediately won't help
 - either linear or exponential back-off necessary
 - have a strategy if retry also doesn't work (Fallbacks!)



```
1 bucket
2     .get("beer") // fetch doc
3     .onErrorResumeNext(bucket.getFromReplica("beer", ReplicaMode.ALL)) // fallback to replica
4     .first() // grab the first doc to arrive
5     .map(doc -> doc.content().getString("name")) // extract the name
6     .timeout(2, TimeUnit.SECONDS) // only wait 2 secs
7     .doOnError(System.err::println) // error log if needed
8     .onErrorReturn(error -> "Not found!"); // if failure, return a default string
```




- Declarative API instead of complicated retryWhen

```
1 observable.retryWhen(  
2     anyOf(BackpressureException.class, TemporaryFailureException.class)  
3     .delay(Delay.exponential(TimeUnit.SECONDS), 5)  
4     .max(100)  
5     .build()  
6 );
```



Code

Where do you find us?



- developer.couchbase.com
- blog.couchbase.com
- @couchbase or @couchbasedev
- forums.couchbase.com
- stackoverflow.com/questions/tagged/couchbase



Couchbase
DEVELOPER COMMUNITY