AVOID THE CALLBACK HELL WHEN USING NON-BLOCKING I/O

Use Reactive frameworks for scalable and resilient server applications

**MAGNUS LARSSON, PÄR WENÅKER, ANDERS ASPLUND**

CALLISTA
— ENTERPRISE —

- Recap - *Don't block your Mobiles and Internet of Things*

- Theory - *FRP, Functional Reactive Programming*

- Examples
    - Java 8 – *Lambdas and Streams*

    - RXJava – *Observables and Observers*

    - Next level – *Scala, Akka and Play*
- Summary & next step

CALLISTA
— ENTERPRISE —

Recap - *Don't block your Mobiles and Internet of Things*

CALLISTA
— ENTERPRISE —

THE SCALABILITY CHALLENGE...

Source: http://www.theconnectivist.com/2014/05/
infographic-the-growth-of-the-internet-of-things/
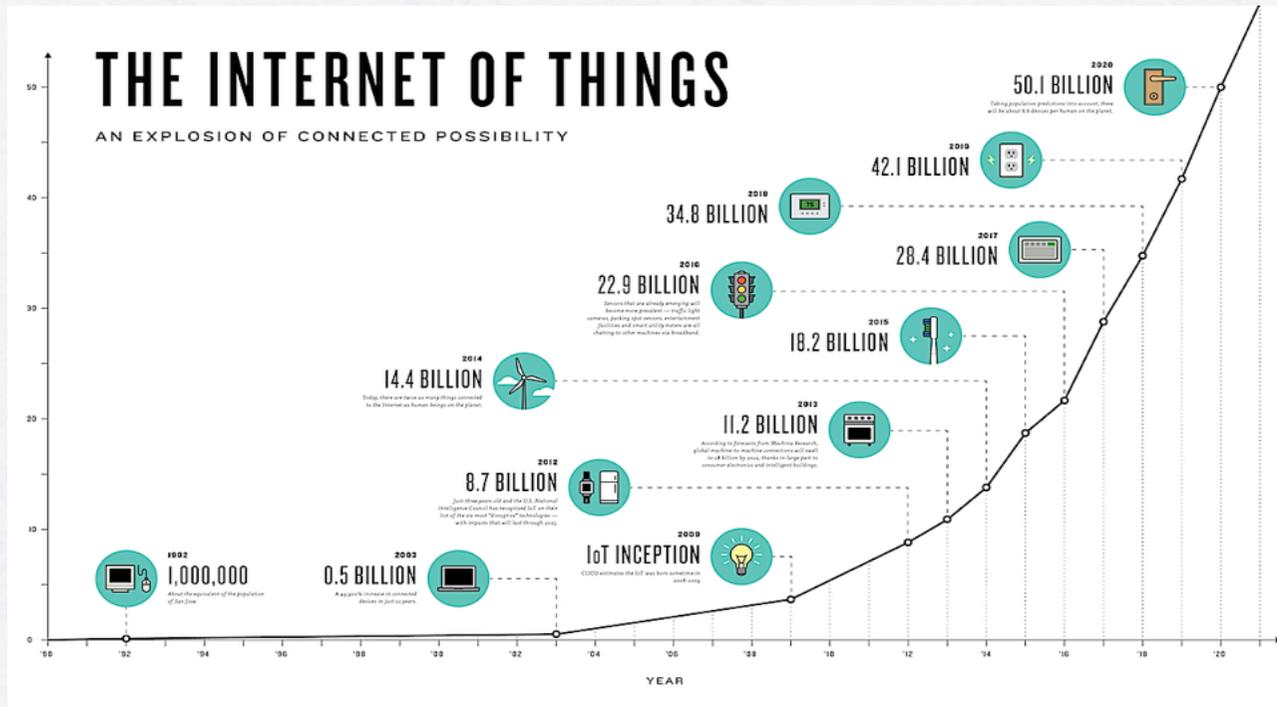
WATCH OUT FOR THE DOMINO EFFECT!

Resilience is as important as scalability

Source: http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html

5

CALLISTA
— ENTERPRISE —

THE REACTIVE MANIFESTO

- http://www.reactivemanifesto.org

CALLISTA
— ENTERPRISE —

Asynchronous & Non Blocking I/O

CALLISTA
— ENTERPRISE —
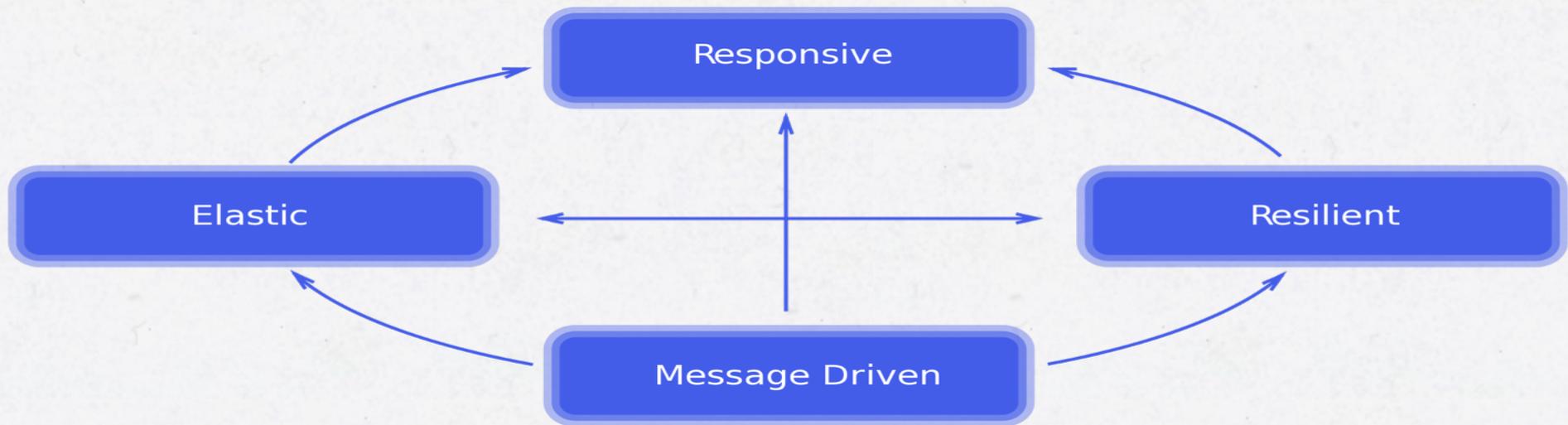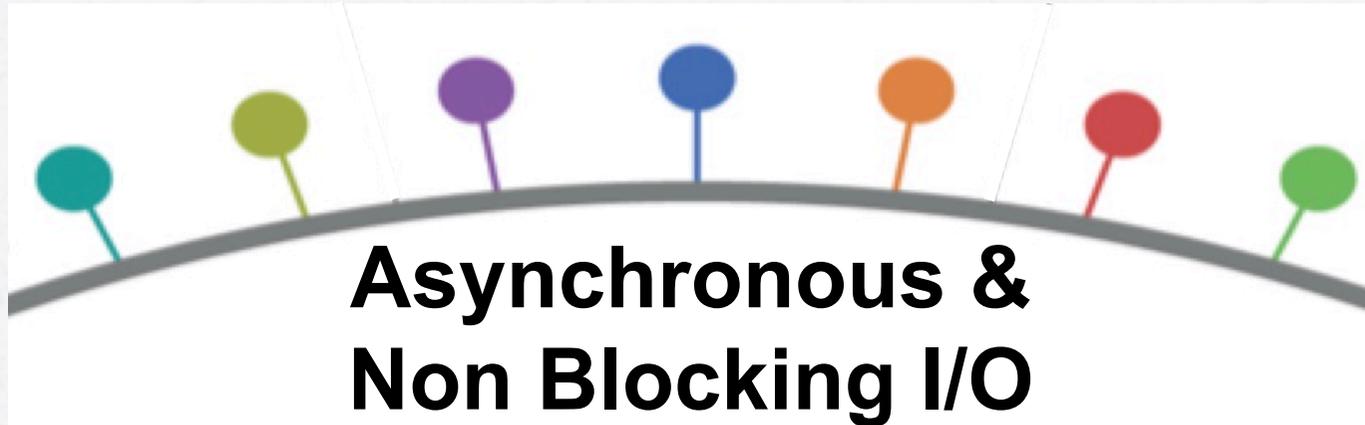
TRADITIONAL BLOCKING I/O

Slow external resource, e.g. a database or another service

The precious thread is busy doing nothing at all...
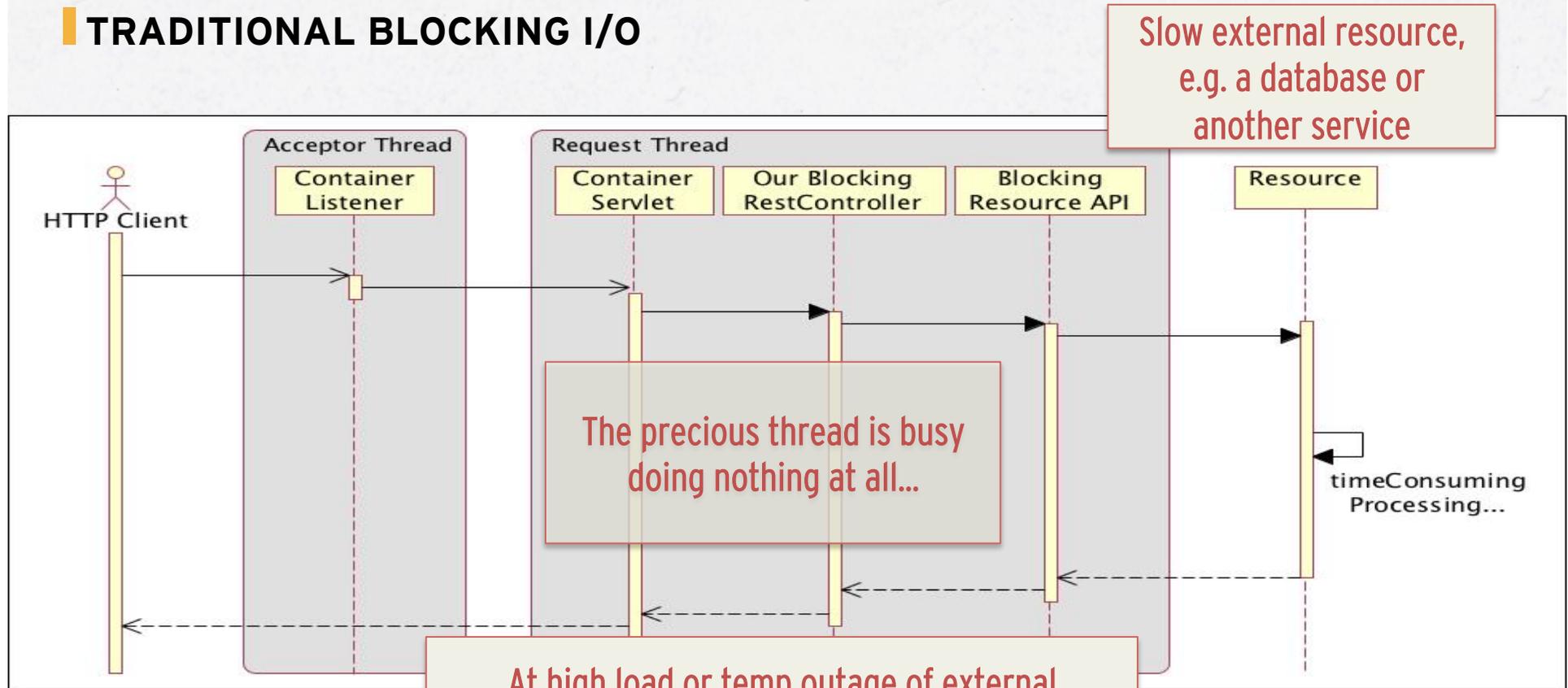
At high load or temp outage of external resources thread-pools are quickly filled up resulting in all kinds of problems...

CALLISTA
— ENTERPRISE —

NON-BLOCKING I/O

CALLISTA
— ENTERPRISE —
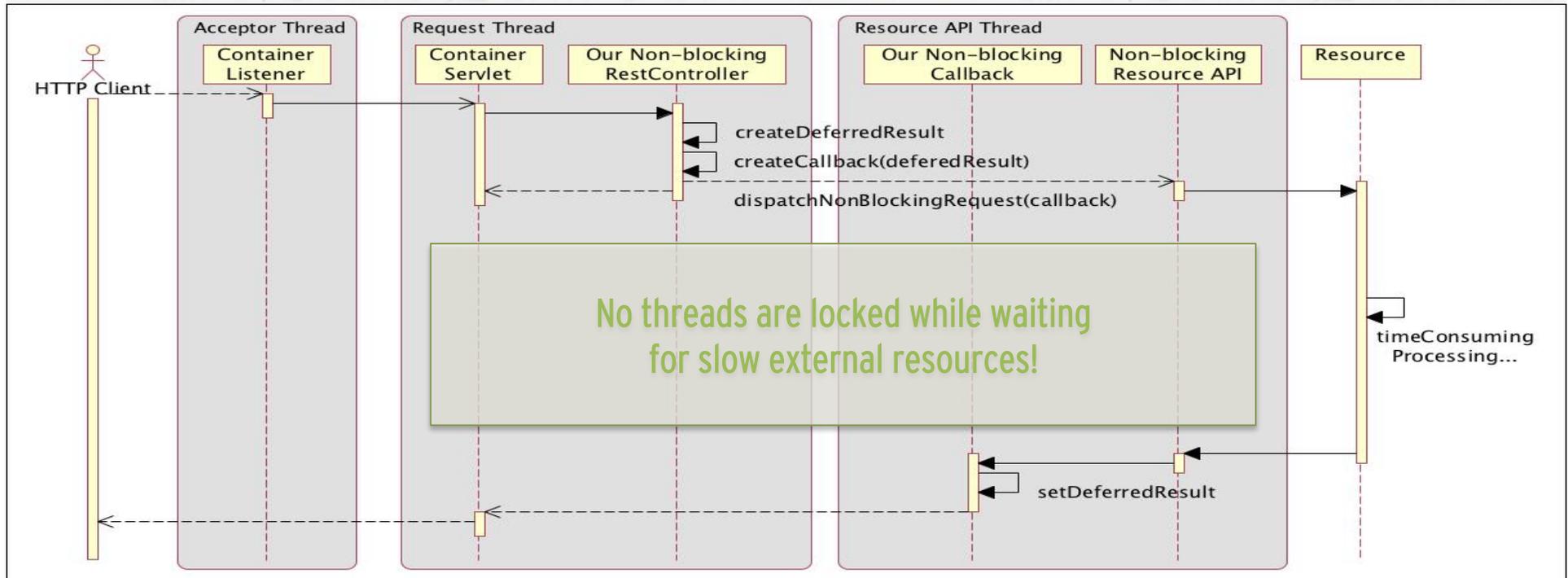
SAMPLE OUTPUT FROM A LOAD TEST

PATTERNS - ROUTER

CALLISTA
— ENTERPRISE —

PATTERNS - AGGREGATOR

Consumer

Consumer

Consumer

Consumer

Consumer

Consumer

Calls to the service
are made in parallel

Aggregator

Lookup
Database

Service

Service

Service

Service

Service

Service

CALLISTA
— ENTERPRISE —
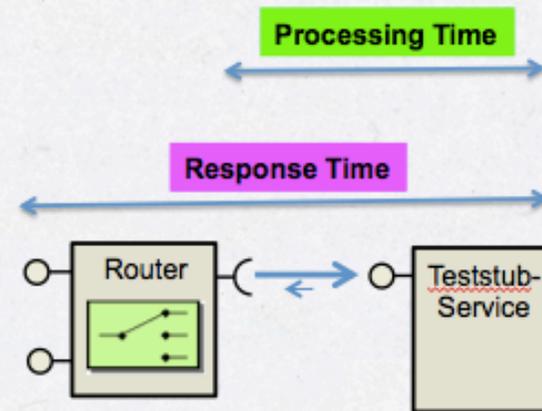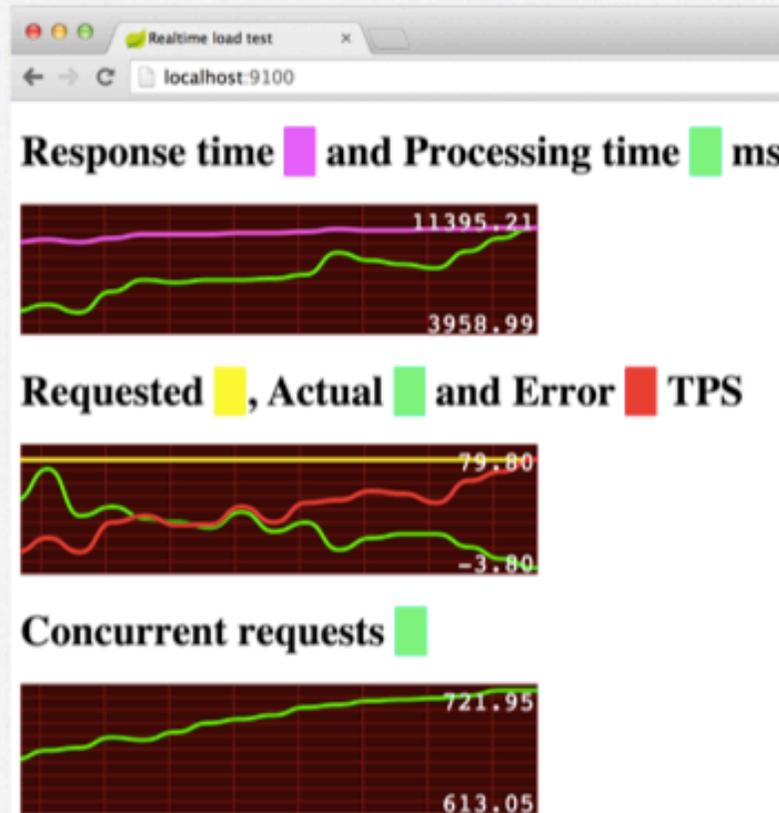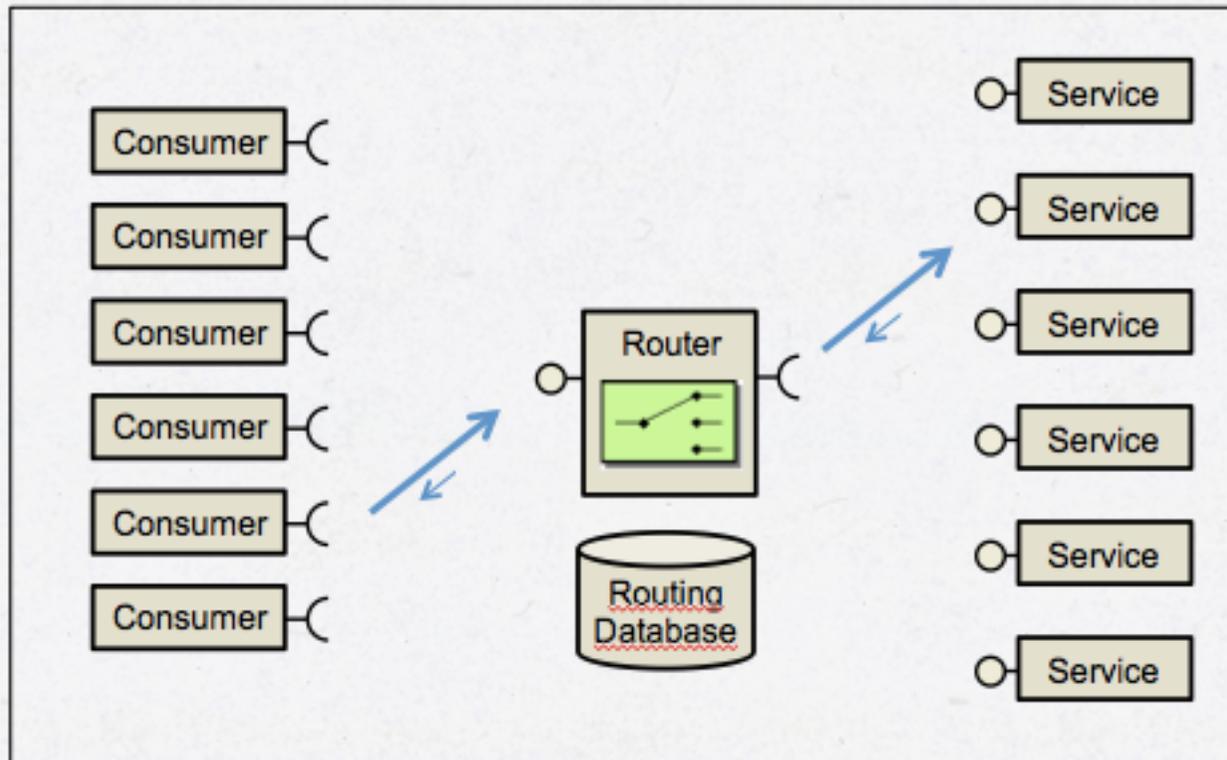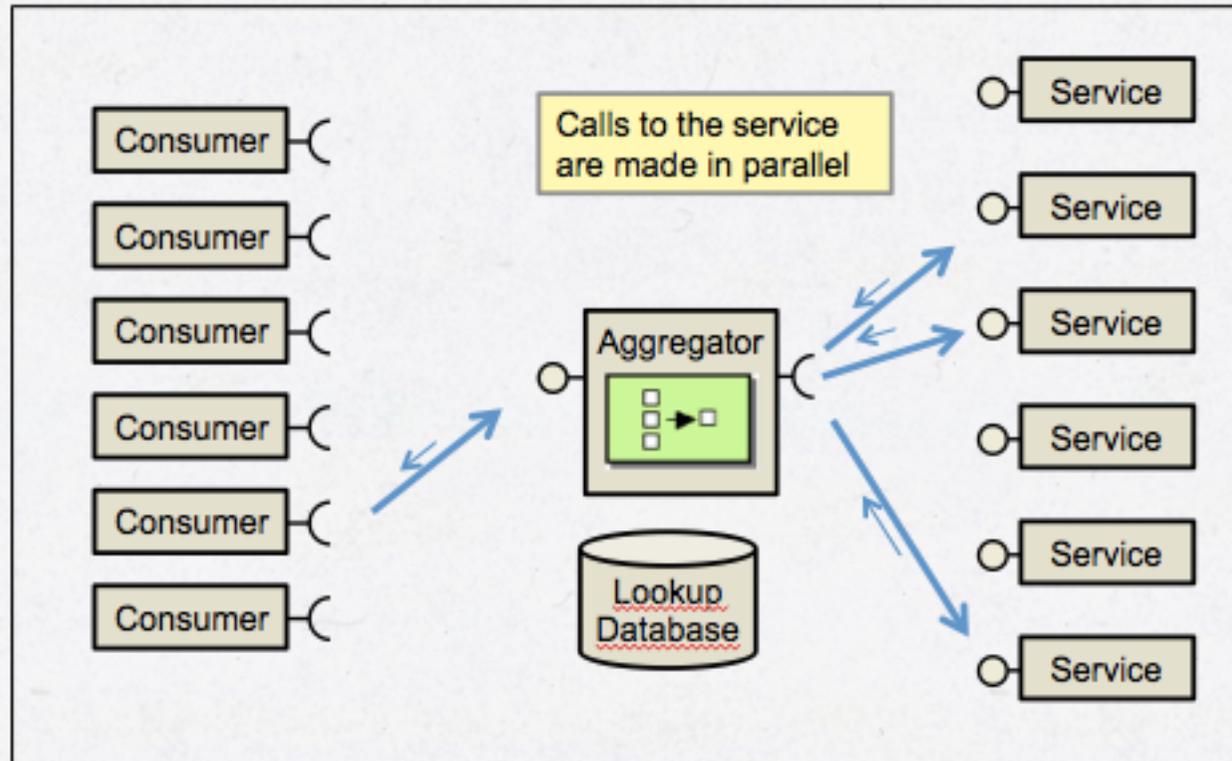
PATTERNS - ROUTING SLIP
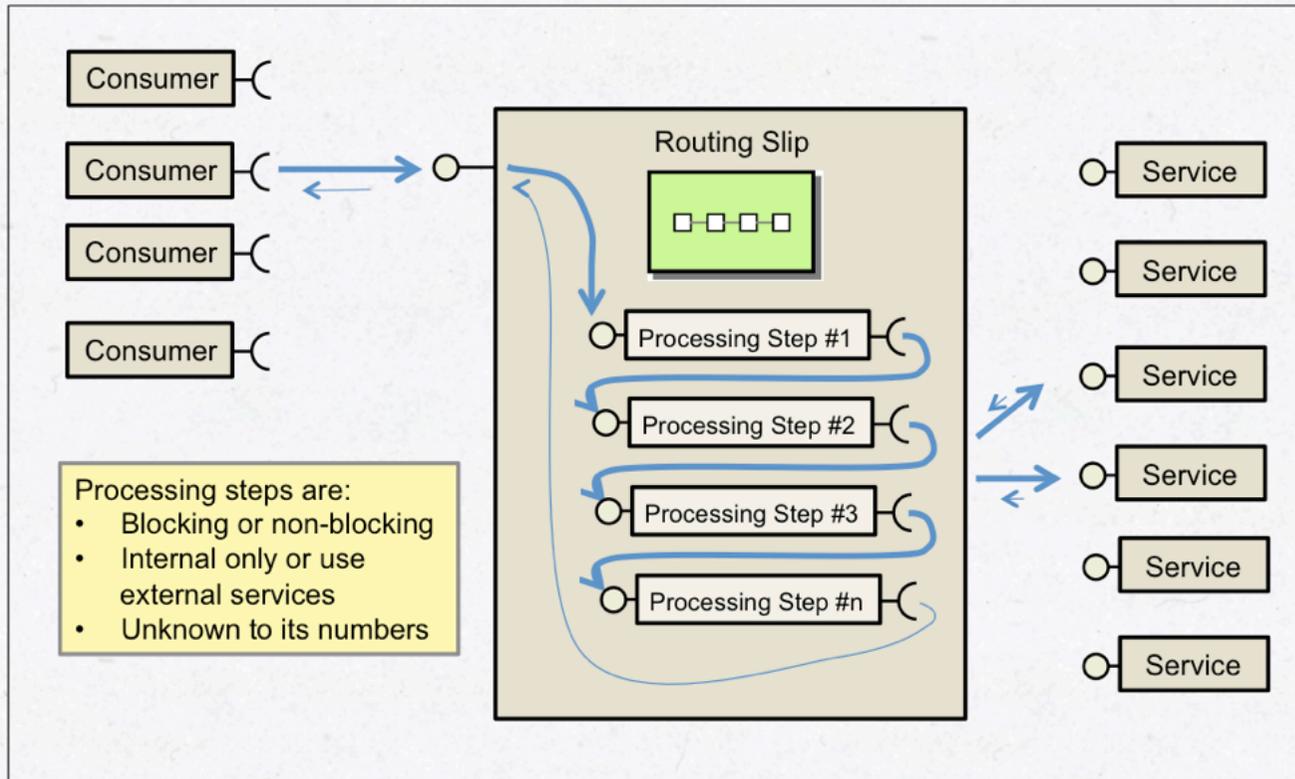
1. SPRING MVC - BLOCKING I/O VS. NON BLOCKING I/O

BLOCKING I/O

```java
@RestController
public class MyController {

  @RequestMapping("/block")
  public R block(...) {
    ...
    return new R(...);
  }
}
```

DeferredResult **IS THE KEY SPRING MVC ABSTRACTION!**

CALLBACK MODEL

NON BLOCKING I/O

```java
@RestController
public class ProcessingController {

  @RequestMapping("/non-block")
  public DeferredResult<R> nonBlock(...) {

    DeferredResult<R> dr =
      new DeferredResult<>();
    dispatch(new MyTask(dr, ...));
    return dr;
```

```java
public class MyTask implements MyCallback {

  private DeferredResult<R> deferredResult;
  public MyTask(DeferredResult<R> dr, ...) {
    this.df = df;
  }
  public void done() {
    df.setResult(new R(...));
```

14

2.C JAVA 8 AND LAMBDAS

```java
@RestController
public class MyController {

  @RequestMapping("/block")
  public String block(...) {
    ...
    resp = callUrl(...);
    return resp.getBody();
  }
}
```

```java
@RestController
public class MyController {

  @RequestMapping("/non-block")
  public DeferredResult<String>
    nonBlock (...) {

    final DeferredResult<String> dr =
      new DeferredResult<>();

    asyncHttpClient.execute(getUrl(...),
      (response) ->  {
        dr.setResult(
          response.getResponseBody());
      },
      (throwable) -> {...}
    );
```

ROUTING SLIP - EXAMPLE OF "THE CALLBACK HELL"

- Perform 5 sequential Non Blocking I/O calls…

```java
@RequestMapping("/routing-slip-non-blocking-lambda")
public DeferredResult<String> nonBlockingRoutingSlip() throws IOException {

    final DeferredResult<String> dr = new DeferredResult<>();

    // Send request #1
    ListenableFuture<Respons                    ent.execute(getUrl(1),
        (Response r1) -> {
            processResult(r1.get                    ss response #1
            asyncHttpClient.exec                    request #2
                (Response r2) -> {
                    processResult(r2.getResponseBody()); // Process response #2
                    asyncHttpClient.execute(getUrl(3),   // Send request #3
                        (Response r3) -> {
                            processResult(r3.getResponseBody()); // Process response #3
                            asyncHttpClient.execute(getUrl(4),   // Send request #4
                                (Response r4) -> {
                                    processResult(r4.getResponseBody()); // Process response #4
                                    asyncHttpClient.execute(getUrl(5),   // Send request #5
```

This is not OK!

PREVIEW - "*THE WAY OUT OF CALLBACK HELL...*"

```java
final DeferredResult<String> dr = new DeferredResult<>();

ListenableFuture<Response> execute = asyncHttpClient.execute(getUrl(1),
  (Response r1) -> {
    processResult(r1.getResponseBody());
    asyncHttpClient.execute(getUrl(2),
      (Response r2) -> {
        processResult(r2.getResponseBody());
        asyncHttpClient.execute(getUrl(3),
          ...
              (Response r5) -> {
                processResult(r5.getResponseBody());
                dr.setResult(getTotalResult());
```

```java
final DeferredResult<String> deferredResult = new DeferredResult<>();

Subscription subscription = Observable.<List<String>>just(new ArrayList<>())
  .flatMap(result -> doAsyncCall(result, 1, this::processResult))
  .flatMap(result -> doAsyncCall(result, 2, this::processResult))
  .flatMap(result -> doAsyncCall(result, 3, this::processResult))
  .flatMap(result -> doAsyncCall(result, 4, this::processResult))
  .flatMap(result -> doAsyncCall(result, 5, this::processResult))
```

PROGRAMMING
You're Doing It Completely Wrong.
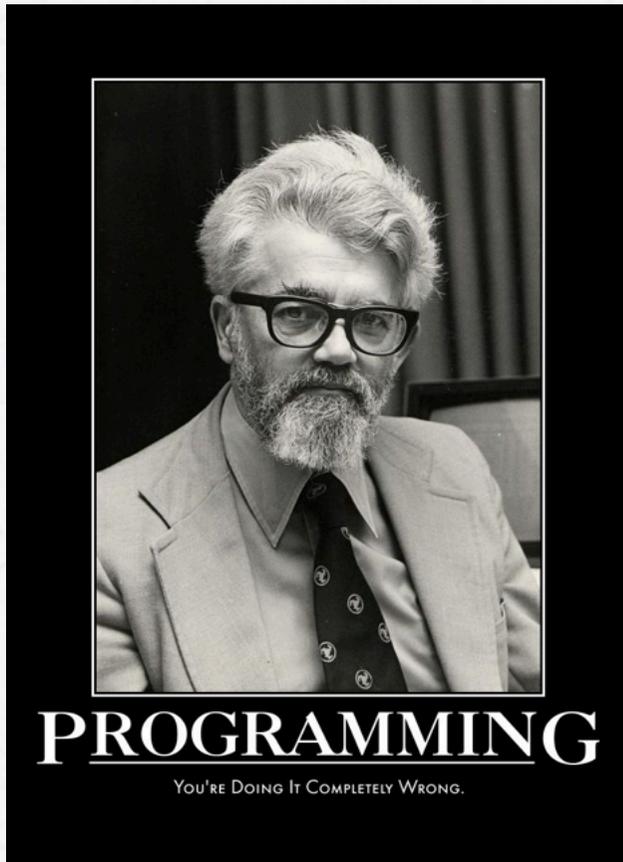
Asynchronous programming, are we doing it wrong?

CALLISTA
— ENTERPRISE —

SYNCHRONOUS VS ASYNCHRONOUS

	Single item	Multiple items
Synchronous	T get ()	Iterable<T> get()
Asynchronous	Future<T> get()	Observable<T> get()

CALLISTA
— ENTERPRISE —

SYNCHRONOUS VS ASYNCHRONOUS

	Single item	Multiple items
Synchronous	T get ()	Iterable<T> get()
Asynchronous	Future<T> get()	Observable<T> get()

FUTURES

- Future is a placeholder for a value that does not yet exist
- The value of a future is calculated concurrently
- Futures are non-blocking (not Java Futures!)
- Futures are only assigned once
- Futures are read only
- Futures are composable without blocking or waiting

CALLISTA
— ENTERPRISE —

FUTURE STATES

CALLISTA
— ENTERPRISE —
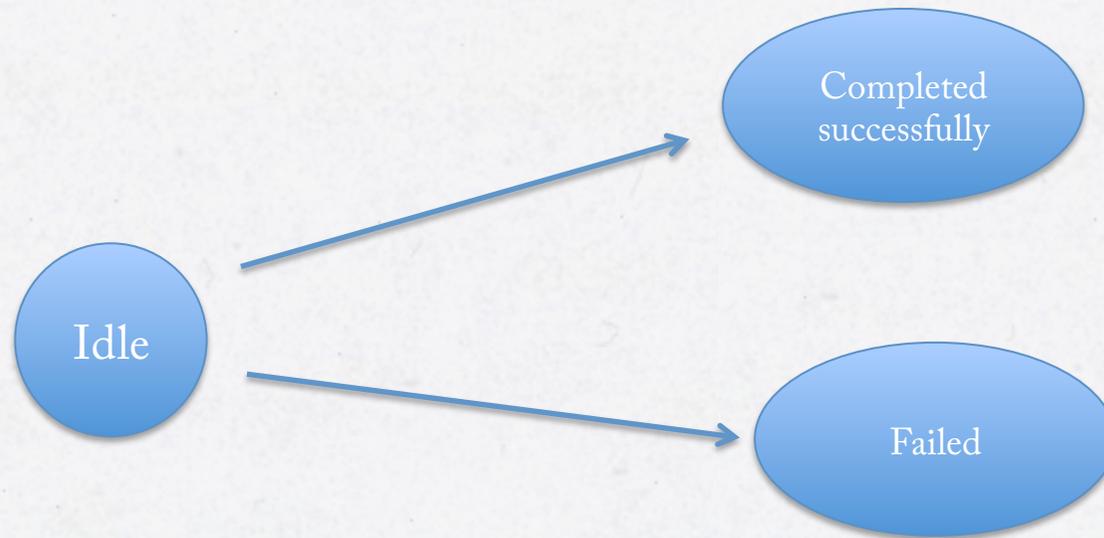
LAMBDAS

```
name = (param1, param2): T-> { ... }


buyStock = (stockName, amount): Boolean -> { ... }
```

CALLISTA
— ENTERPRISE —

ASYNCHRONOUS WITH CALLBACK

CALLISTA
— ENTERPRISE —
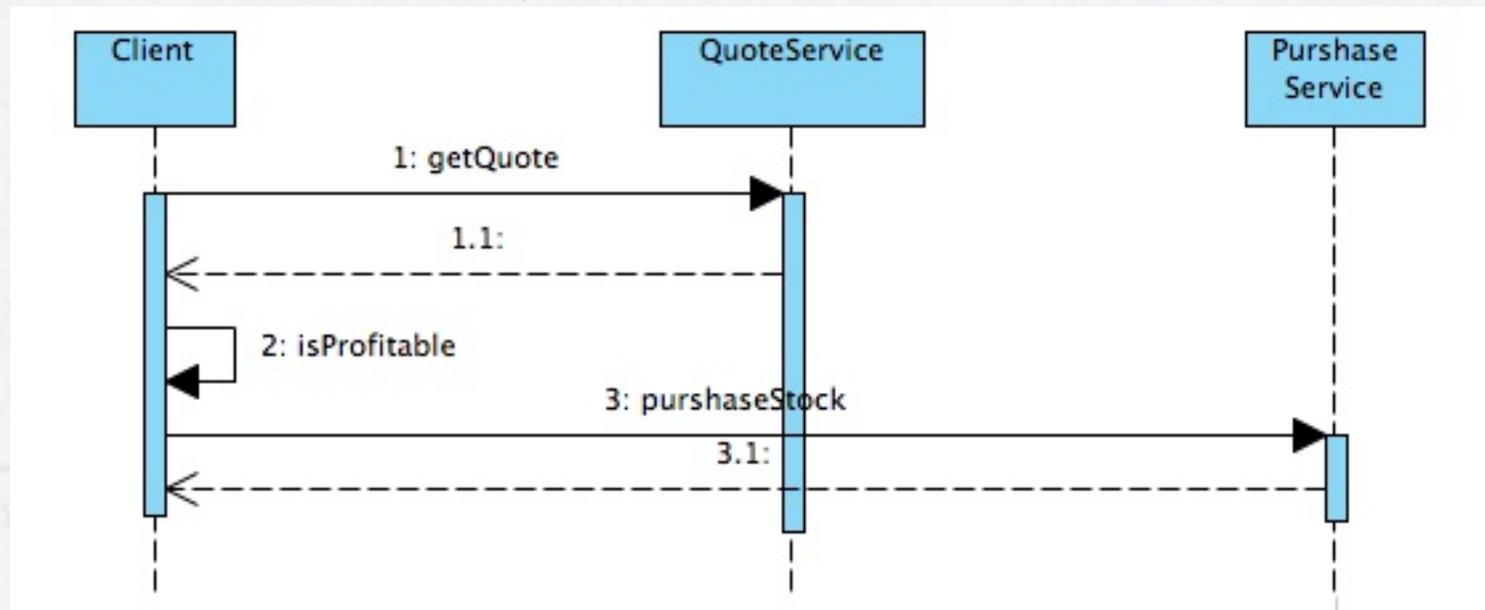
ASYNCHRONOUS WITH CALLBACK

```
getQuote = (stock, callback) -> { ... }
buyStock = (stock, callback) -> { ... }

getQuote( "ERICB", quote -> {

    if( isProfitable(quote) ) buyStock( "ERICB", result -> {

        if( result == OK )
            println( "Purshased $result" )
        }

    })

})
```
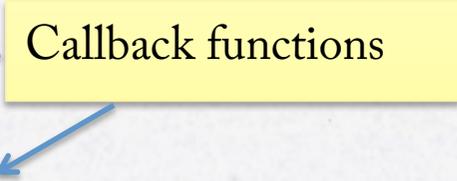
Callback functions

CALLISTA
— ENTERPRISE —

ASYNCHRONOUS WITH CALLBACK

CALLISTA
— ENTERPRISE —
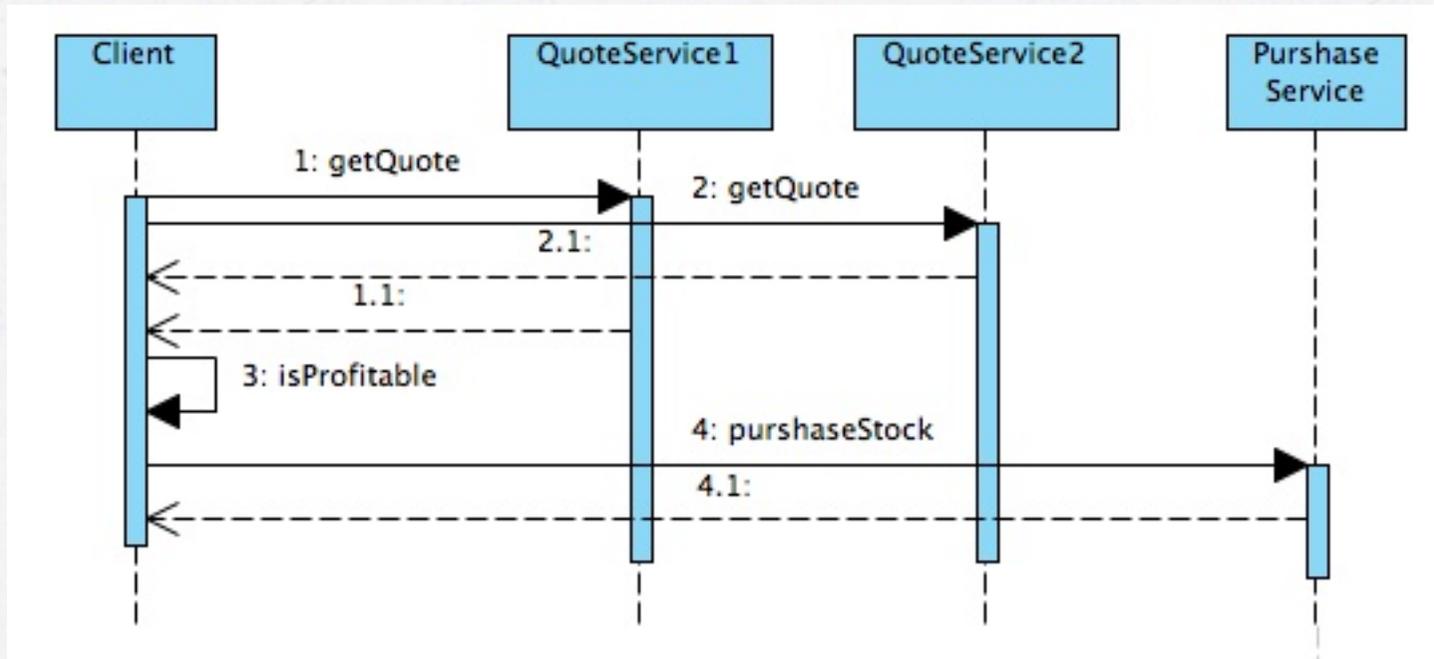
ASYNCHRONOUS WITH CALLBACK

```
getQuote (stock, callback) -> { ... }

var ericQuote, volvoQuote
getQuote( "ERICB", quote -> {
    ericQuote = quote
})
getQuote( "VOLVOB", quote -> {
    volvoQuote = quote
})


wait…


if( isProfitable(ericQuote, volvoQuote) )
    buyStock("ERICB" result -> …)
```

Parallel executions

Another non blocking call

Wait for both to complete BLOCKING!

CALLISTA
— ENTERPRISE —

SOLUTION?
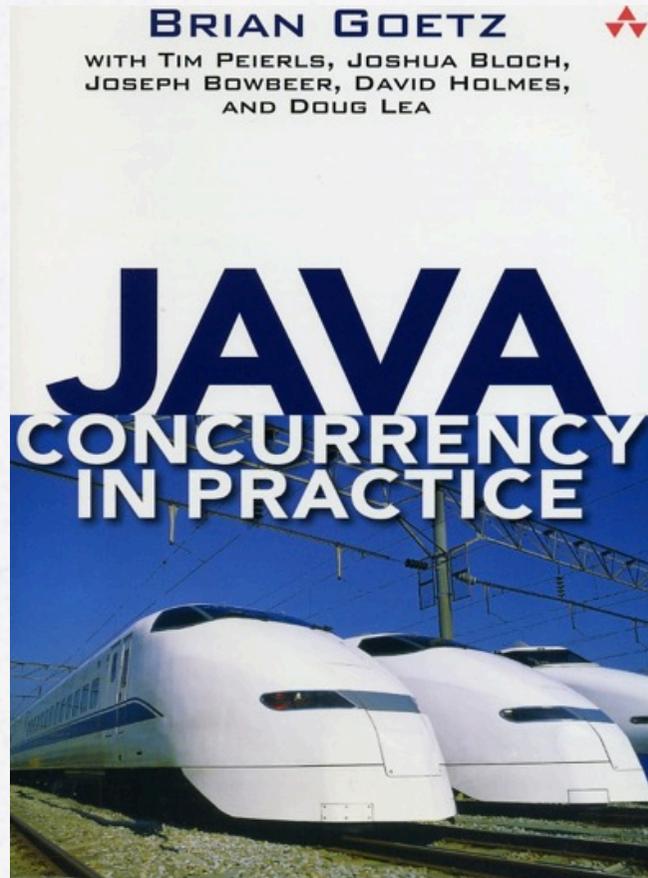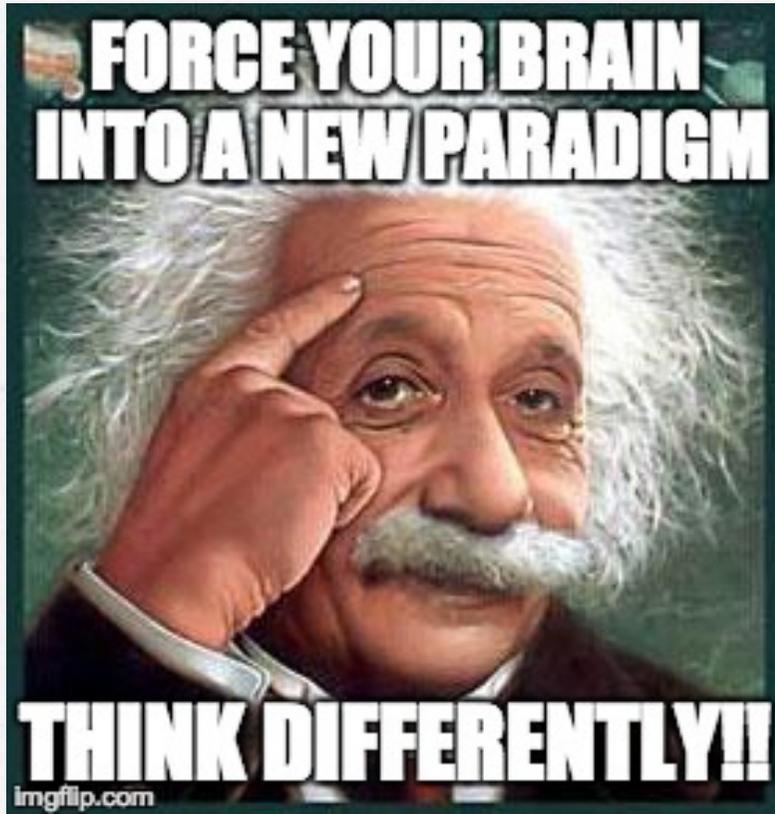
Let go of state and imperative programming…

CALLISTA
— ENTERPRISE —

29

ASYNCHRONOUS WITH FUTURES

```
getQuote = (stock): Future -> { ... }
buyStock = (stock): Future -> { ... }

var eQuoteF = getQuote("ERICB")
var vQuoteF = getQuote("VOLVOB")

when {
  eQuote <- eQuoteF
  vQuote <- vQuoteF
  if(isProfitable(eQuote, vQuote)
  result <- buyStock(stockToBuy)
} then {
  r -> print("Purshased $r"),
  err -> print("Did not purshase!")
}
```

Start two parallel calls

Extract value from futures

Check condition

"callbacks"

CALLISTA
— ENTERPRISE —

FUTURES

- Future is a placeholder for a value that does not yet exist
- The value of a future is calculated concurrently
- Futures are non-blocking (not Java Futures!)
- Futures are only assigned once
- Futures are read only
- **Futures are composable without blocking or waiting**

CALLISTA
— ENTERPRISE —

FUNCTION COMPOSITION

```
f: X ➔ Y
g: Y ➔ Z

f ο g: X ➔ Z      "f composed with g"

(f ο g)(x): g(f(x))


z:Z = g(f(x))
```

CALLISTA
— ENTERPRISE —

Create new futures by *transforming* and *combining* existing futures using the Future API.

CALLISTA
— ENTERPRISE —

How do we transform a Future?

CALLISTA
— ENTERPRISE —

FUTURE COMPOSITION

Future.map

f : String -> Int

Future<String>

Future["Hello"]

map(s -> s.length)
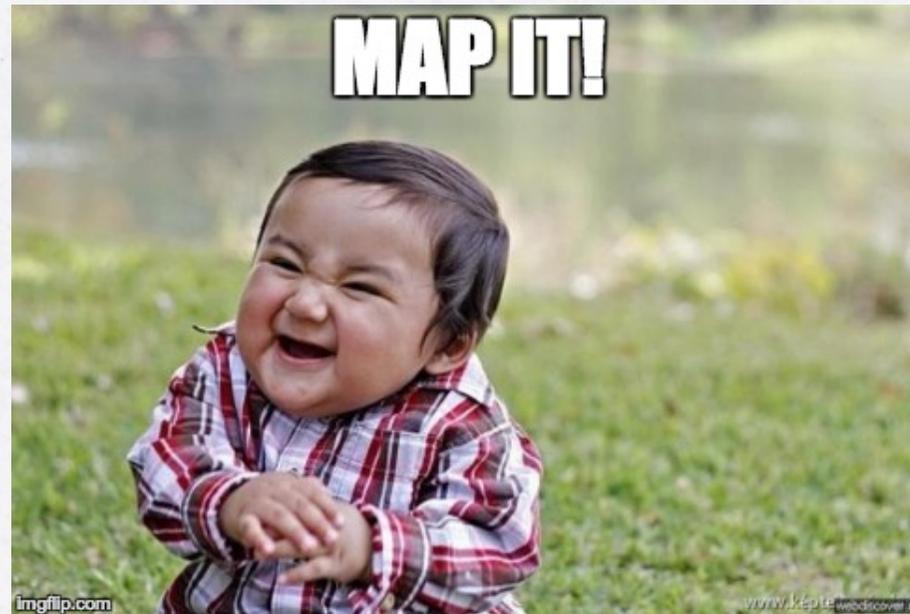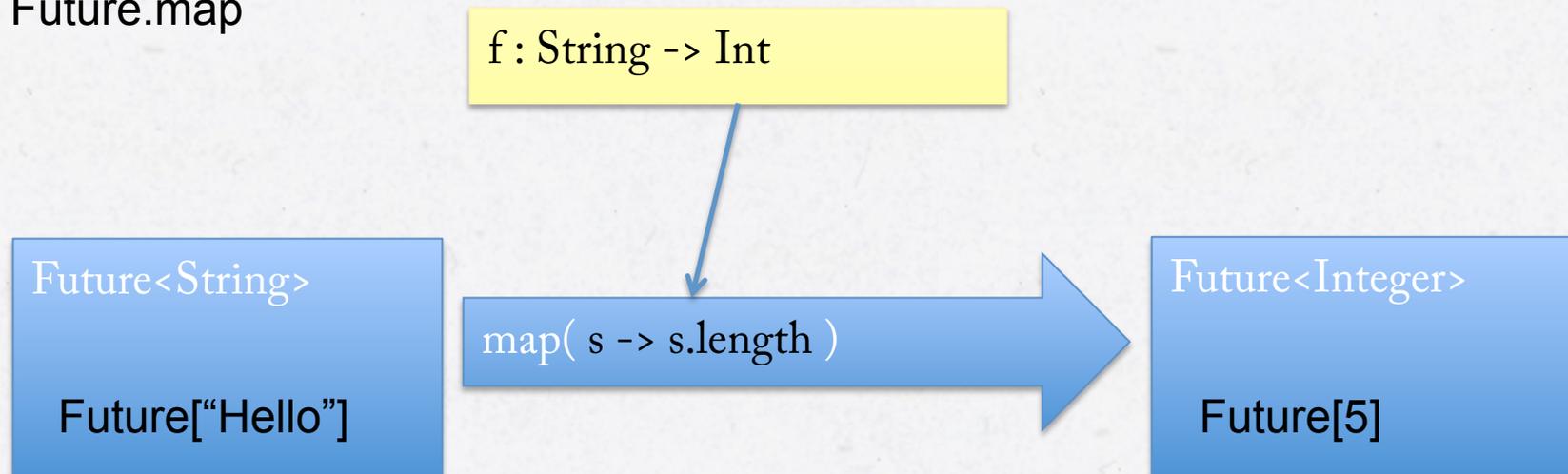
Future<Integer>

Future[5]

```
Future.completed(1)
        .map((v) -> v + 10)
        .onSuccess((v) -> println(v))



> 11
```

CALLISTA
— ENTERPRISE —

How do we combine Futures?

CALLISTA
— ENTERPRISE —
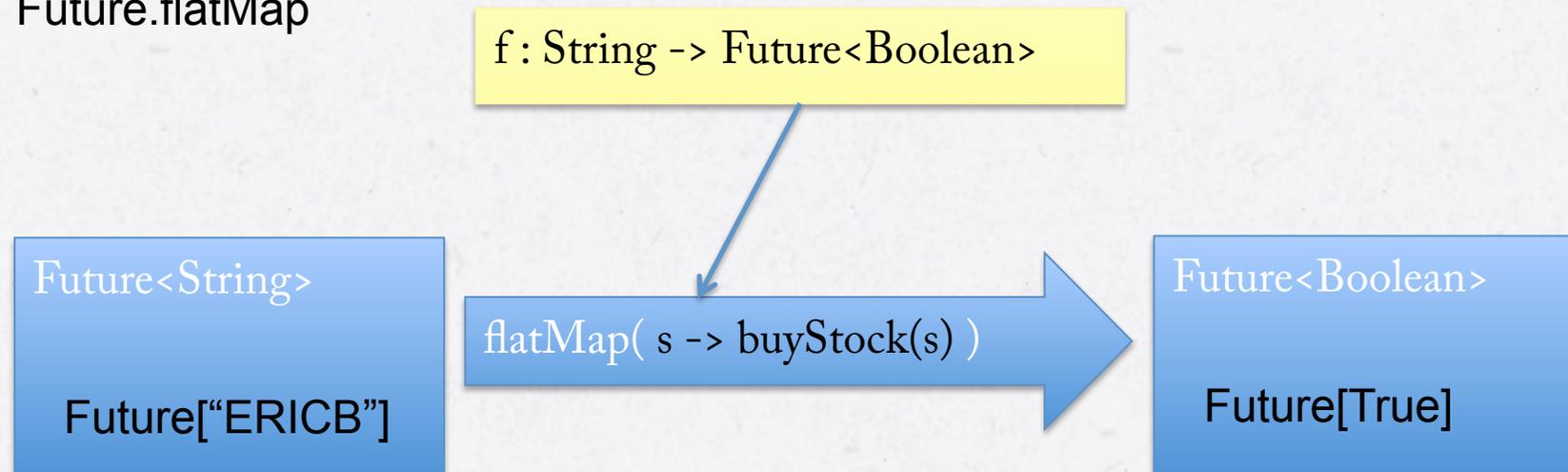
```
Future.completed(1)
        .flatMap((v) -> Future.completed(v + 10))
        .onSuccess((v) -> println(v))



> 11
```

CALLISTA
— ENTERPRISE —

ASYNCHRONOUS WITH FUTURES

```
getQuote (stock): Future<Int> -> { ... }
buyStock (stock): Future<Int> -> { ... }

var eQuoteF = getQuote("ERICB")
var vQuoteF = getQuote("VOLVOB")


val resultF = eQuoteF.flatMap{ eQuote =>
    vQuoteF
        .filter(vQuote -> isProfitable(eQuote, vQuote))
        .flatMap(() -> buyStock("ERICB"))
}

resultF.onSuccess(() -> println("Purshased ERICB"))
```

f : Int-> Future<Int>

f : ()-> Future<Int>

40

CALLISTA
— ENTERPRISE —

STREAMS

CALLISTA
— ENTERPRISE —
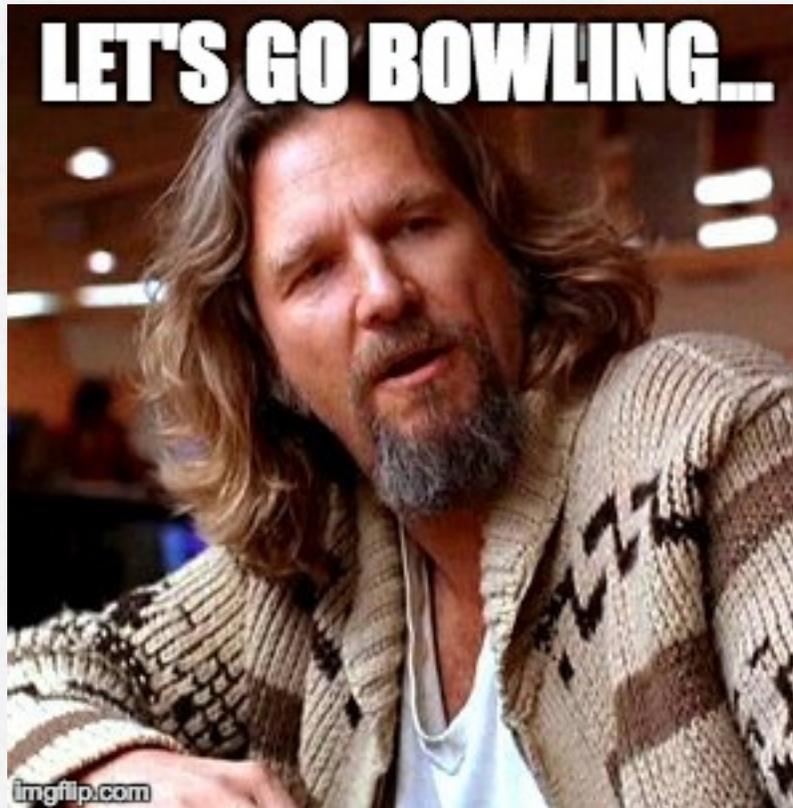
SYNCHRONOUS VS ASYNCHRONOUS

	Single items	Multiple items
Synchronous	T get ()	Iterable<T> get()
Asynchronous	Future<T> get()	Observable<T> get()

OBSERVABLE

Design Patterns
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Observable

Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()

Observer
+notify()

onNext()
onError()
onCompleted()

notifyObservers()
  for observer in observerCollection
  call observer.notify()

ConcreteObserverA
+notify()

ConcreteObserverB
+notify()

CALLISTA
— ENTERPRISE —

COLLECTIONS WITH OBSERVABLES
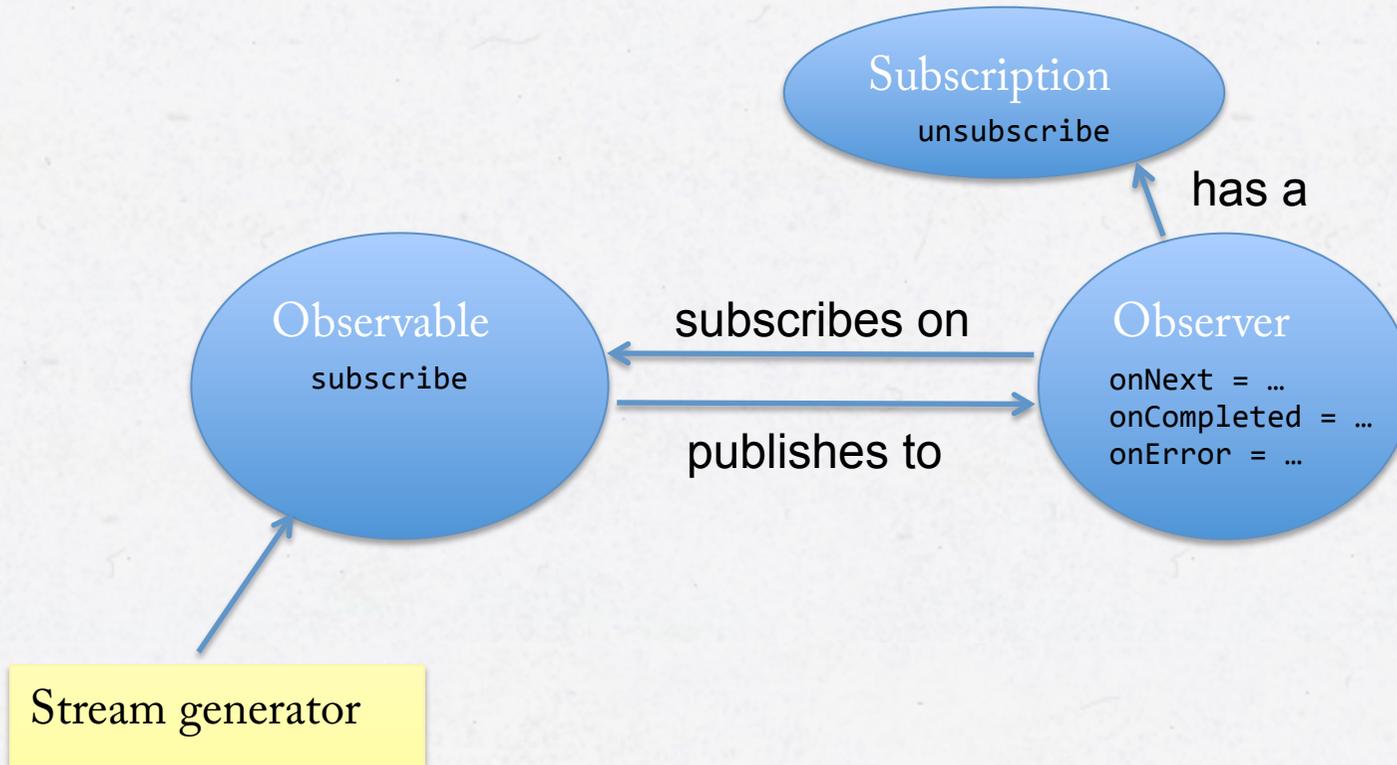
MARBLE DIAGRAMS

This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

This vertical line indicates that the Observable has completed successfully.

These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

flip

This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

CALLISTA
— ENTERPRISE —

RX MAP

CALLISTA
— ENTERPRISE —
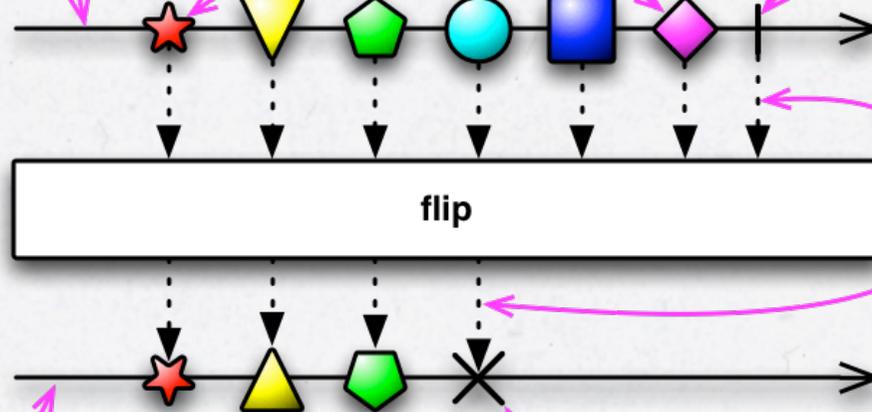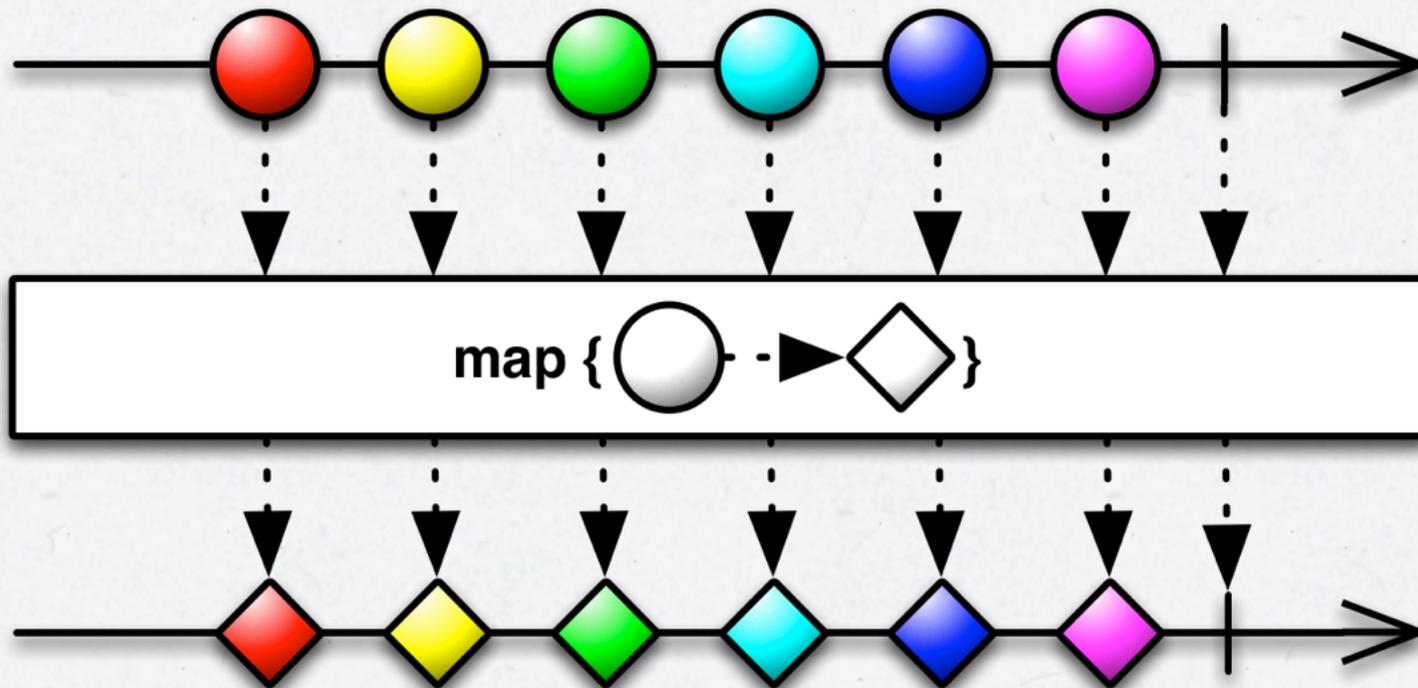
RX MAP

```
Observable<Integer> o = Observable.range(1,5);

o.map(v -> v * 10).subscribe(System.out::println);
```

```
10
20
30
40
50
```

CALLISTA
— ENTERPRISE —
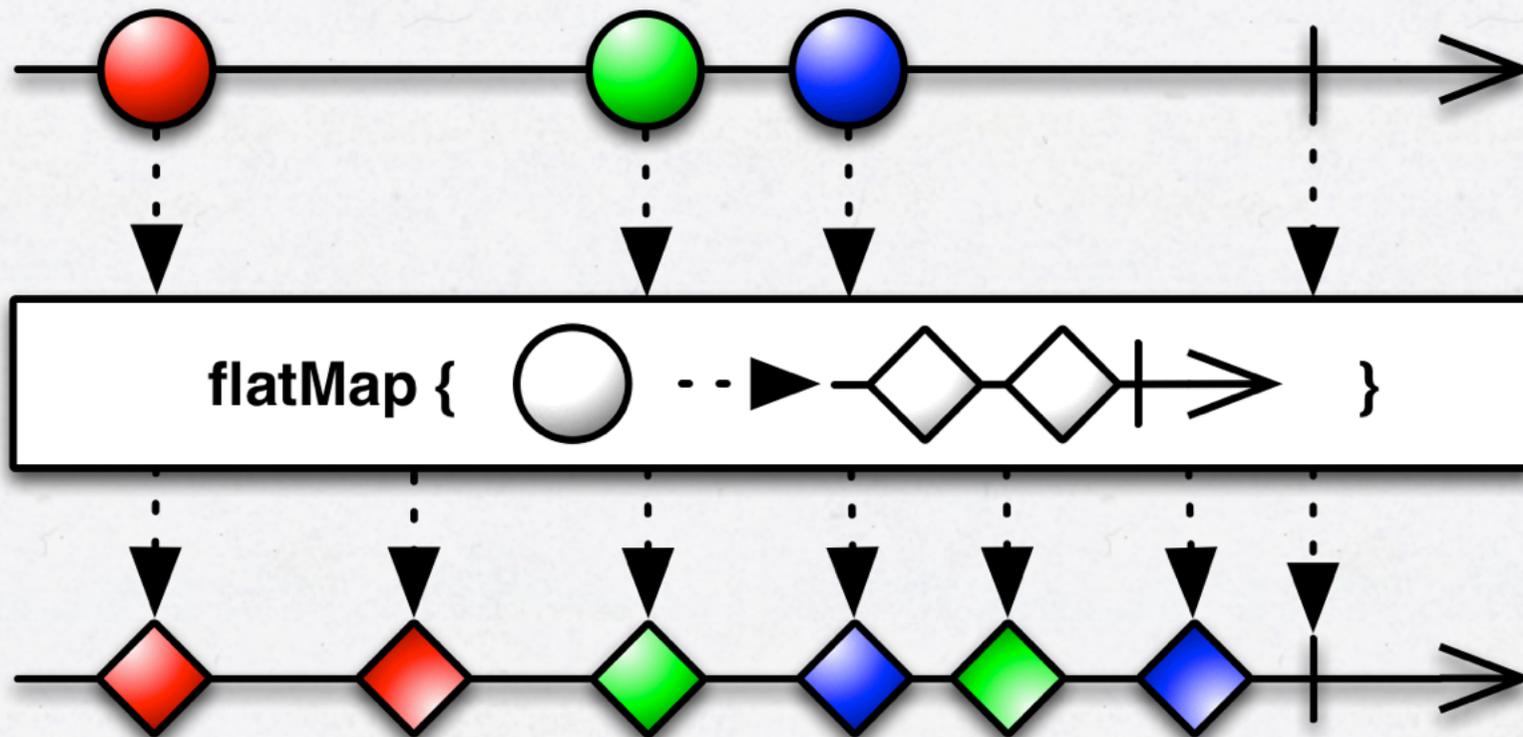
RX FLATMAP

```
Observable<Integer> o = Observable.range(1,5);

o.flatMap(v -> Observable.range(v*10, 2)).subscribe(System.out::println);
```

```
 10
 11
 20
 21
 30
 31
 40
 41
 50
 51
```

CALLISTA
— ENTERPRISE —

RX FILTER

CALLISTA
— ENTERPRISE —
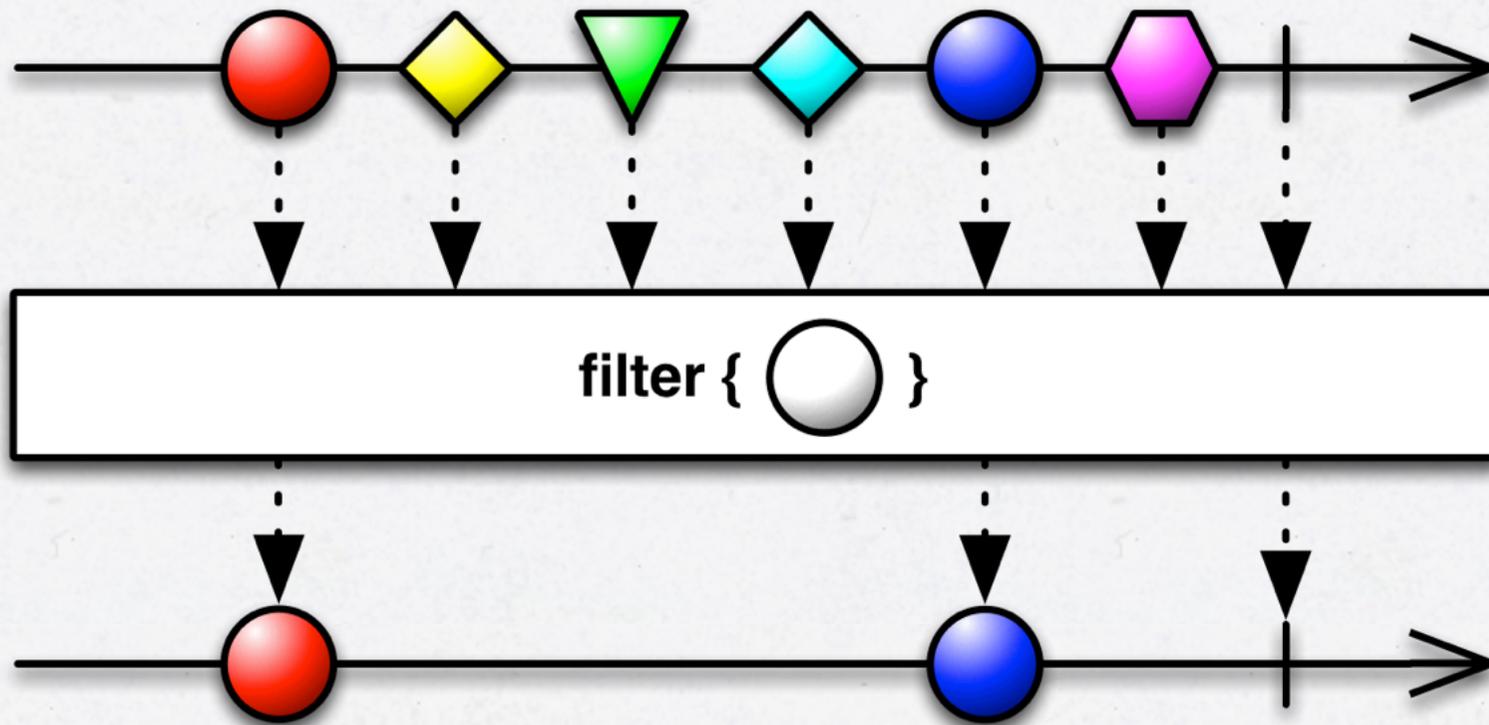
RX FILTER

```
Observable<Integer> o = Observable.range(1,10);

o.filter(s -> s % 3 == 0).subscribe(System.out::println);
```

```
3
6
9
```

NEXT STEP

CALLISTA
— ENTERPRISE —

SYNCHRONOUS VS ASYNCHRONOUS

	Single item	Multiple items
Synchronous	T get ()	Iterable<T> get()
Asynchronous	Future<T> get()	Observable<T> get()

- Since 1.5
- Future – Blocking api

- Since 1.8
- CompletableFuture – Non-Blocking

Future

- Create
  - new CompletableFuture<T>()

- Complete
  - Successfully

    Value

    » complete(T value)
  - With Error

    Throwable

    » completeExceptionally(Throwable t)

CALLISTA
— ENTERPRISE —

EXAMPLE - ASYNC HTTP CLIENT

```java
public class AsyncHttpClientJava8 {
    public CompletableFuture<Response> execute(String url, int id) {

        final CompletableFuture<Response> result = new CompletableFuture<>();

        asyncHttpClient.prepareGet(url).execute(new AsyncCompletionHandler<Response>() {

            @Override
            public Response onCompleted(Response response) throws Exception {
                result.complete(response);
                return response;
            }

            @Override
            public void onThrowable(Throwable t) {
                result.completeExceptionally(t);
            }

        });

        return result;
    }
}
```

CALLISTA
— ENTERPRISE —

FUTURE COMPOSITION

Future.map

f : String -> Int

Future<String>

map(s -> s.length)

Future<Integer>

CALLISTA
— ENTERPRISE —
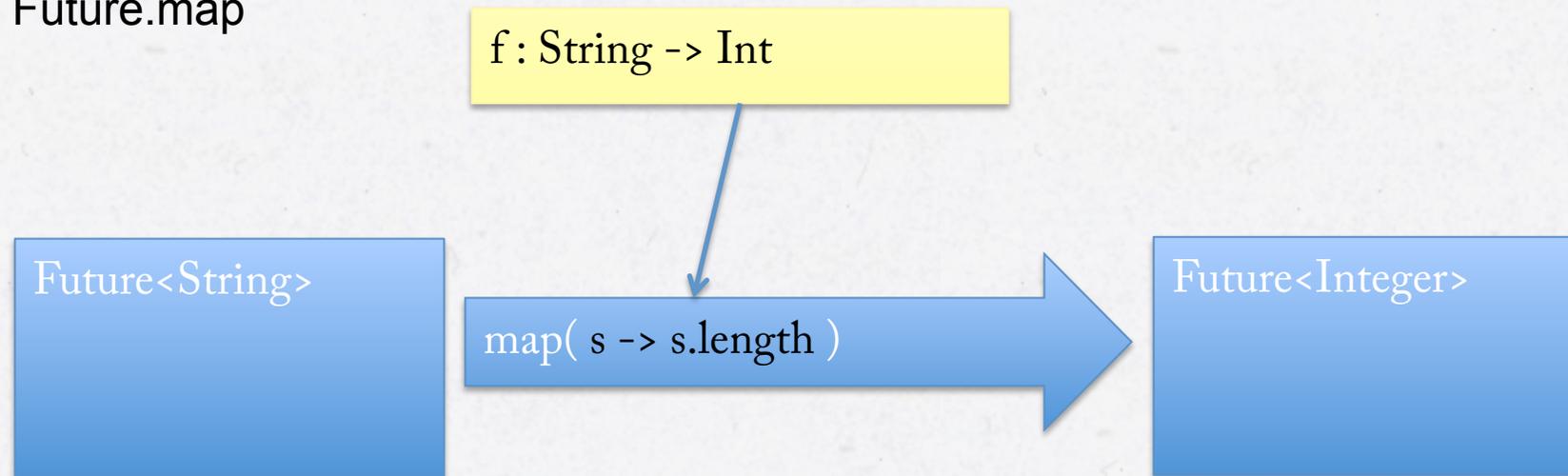
COMPLETABLEFUTURE - MAP

- Transforming
  - thenApply / thenApplyAsync ⇔ Map
- Completion (Callback)
  - thenAccept / thenAcceptAsync

```java
CompletableFuture.completedFuture(3)
    .thenApply(i -> i * Math.PI)
    .thenAccept(i -> System.out.println(i));

//Prints out 9,424776...
```

CALLISTA
— ENTERPRISE —

COMPLETABLEFUTURE - MAP

```java
//Returns coordinates for an address
CompletableFuture<String> downloadPage(Url url);

//Parse to html document
Document parse(String page);

CompletableFuture<String> pageF = downloadPage(url)
CompletableFuture<Document> docF = pageF.thenApply(page -> parse(page));

docF.thenAccept(doc -> System.out.println(doc));

//Prints the parsed document
```

CALLISTA
— ENTERPRISE —

Future.flatMap

f : String -> Future<Int>

Future<String>

flatMap(s -> buyStock(s))
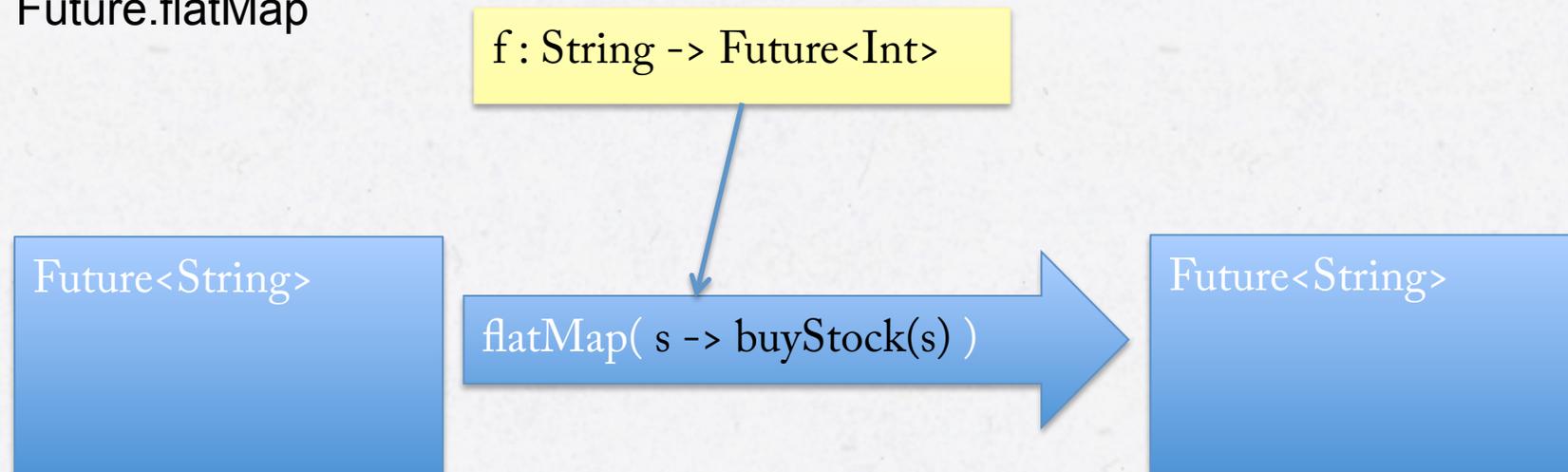
Future<String>

COMPLETABLEFUTURE - FLATMAP

thenCompose(…) / thenComposeAsync(…)   ⇔   flatMap

```java
CompletableFuture.completedFuture(3)
    .thenCompose(i -> CompletableFuture.completedFuture(i * Math.PI))
    .thenAccept(i -> System.out.println(i));

//Prints out 9,424776...
```

COMPLETABLEFUTURE - FLATMAP

```java
//Returns coordinates for an address
CompletableFuture<Coordinates> getCoordinates(Address address);

//Returns weather forecast for the provided coordinates
CompletableFuture<WeatherForecast> getWeatherForecast(Coordinates coords);


CompletableFuture<Coordinates> coordsF = getCoordinates(address)

CompletableFuture<WeatherForecast> coordsF.thenCompose(coords -> getWeatherForecast(coords))

weatherForecastF.thenAccept(weatherForecast -> System.out.println(weatherForecast));

//Prints weather forecast for a given address
```

CALLISTA
— ENTERPRISE —

AGGREGATOR EXAMPLE

CALLISTA
— ENTERPRISE —
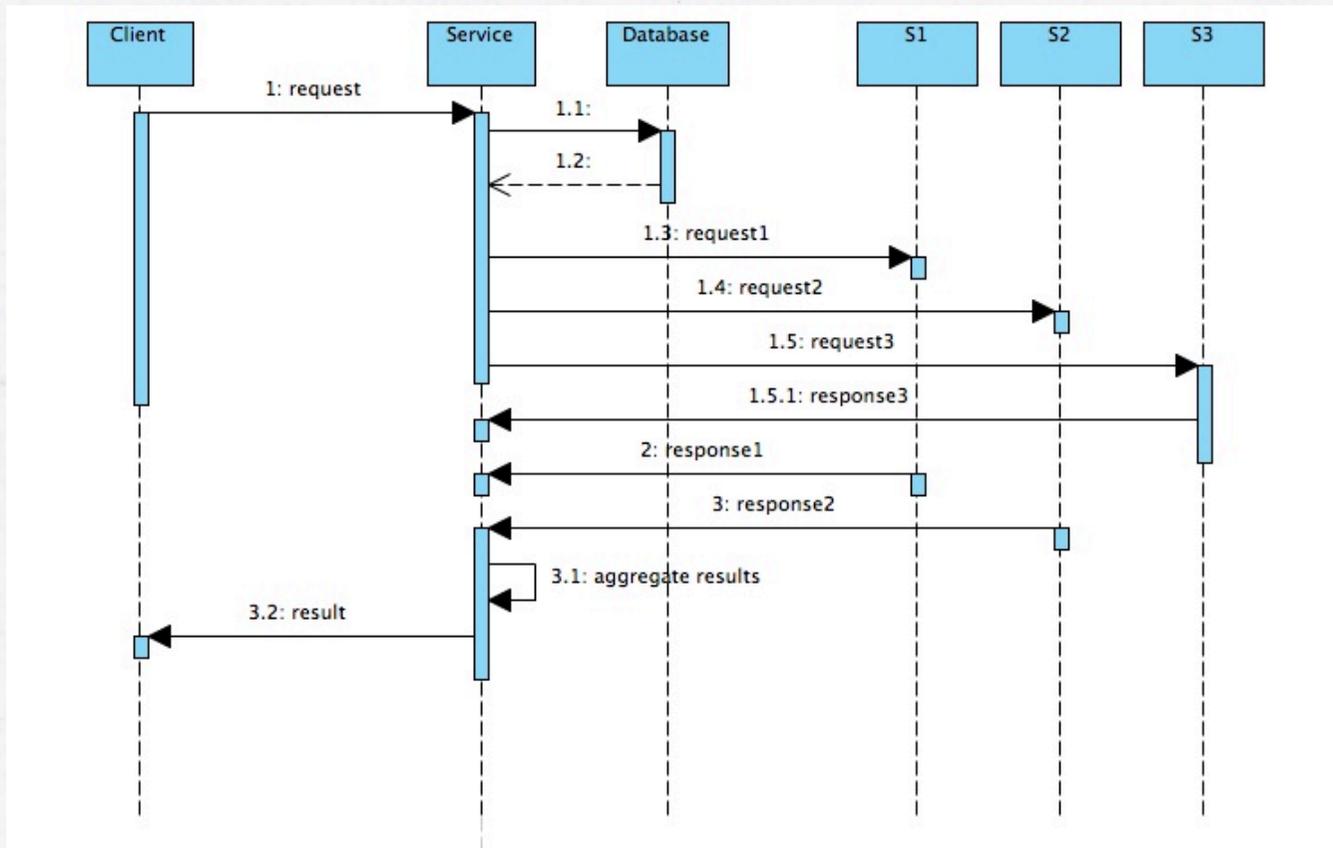
AGGREGATOR - JAVA8

```java
@RequestMapping("/aggregate-non-blocking-java8")
public DeferredResult<String> nonBlockingAggregator(...) {

    final DeferredResult<String> deferredResult = new DeferredResult<>();
    final DbLookup dbLookup = new DbLookup(dbLookupMs, dbHits);

    final CompletableFuture<List<String>> urlsF =
            supplyAsync(() -> dbLookup.lookupUrlsInDb(SP_NON_BLOCKING_URL, minMs, maxMs), dbThreadPoolExecutor);

    urlsF
        .thenComposeAsync(urls -> executeHttpRequests(urls))
        .thenAccept(result -> populateDeferredResult(deferredResult, result));

    return deferredResult;
}

private CompletableFuture<List<String>> executeHttpRequests(List<String> urls) {
    CompletableFuture<List<String>> futureResponses =
            sequence(IntStream.rangeClosed(1, urls.size()).mapToObj(i ->
                doAsyncCall(urls.get(i - 1), i))
            .collect(Collectors.toList()));

    return futureResponses;
}

private CompletableFuture<String> doAsyncCall(String url, int id) {
    return asyncHttpClientJava8.execute(url, id)
            .thenApply(response -> extractResponseBody(response));
}

private void populateDeferredResult(DeferredResult<String> deferredResult, List<String> result) {
    String collectedResponse = result.stream().collect(Collectors.joining("\n"));
    deferredResult.setResult(collectedResponse);
}
```

CALLISTA
— ENTERPRISE —

ROUTING SLIP EXAMPLE
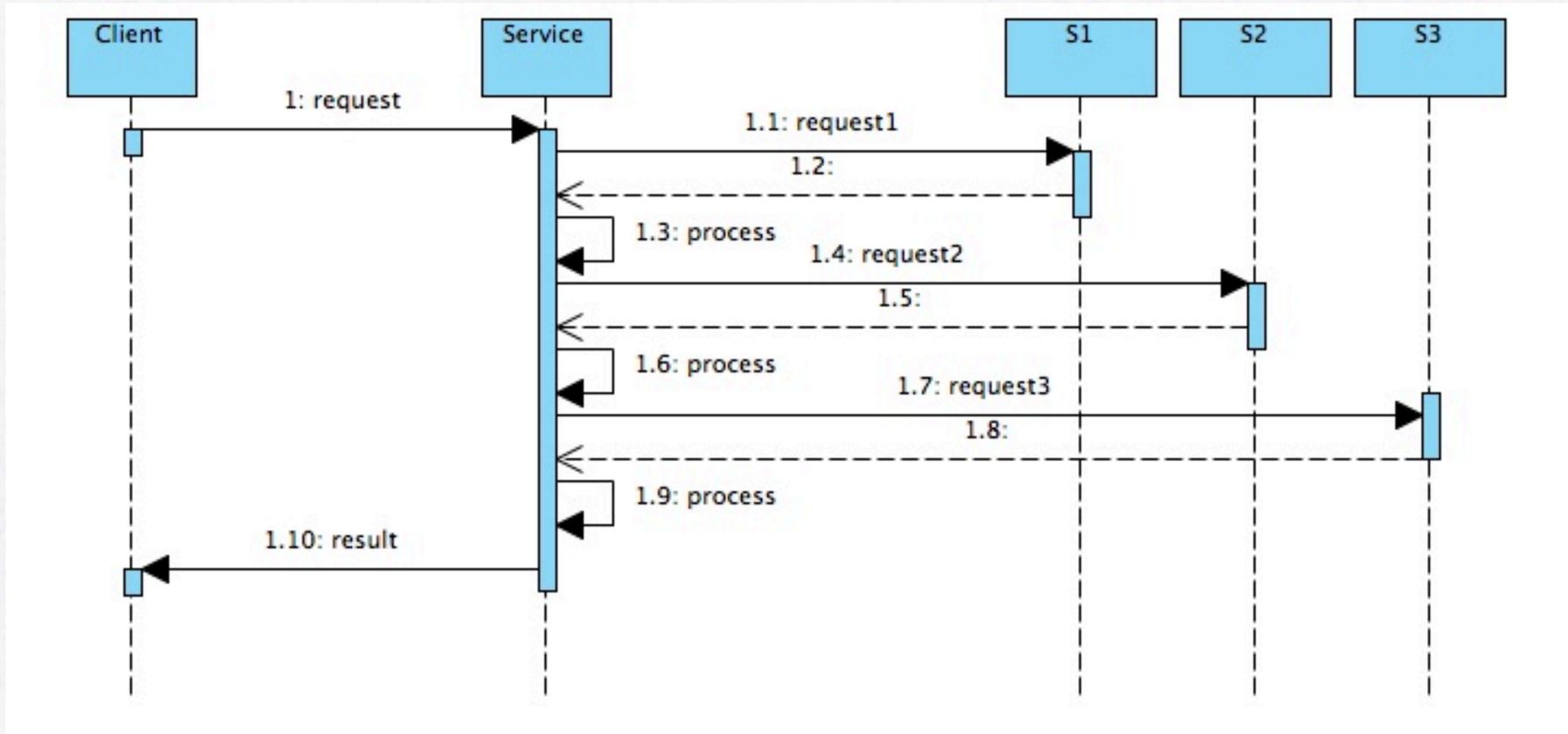
ROUTING SLIP - DO YOU REMEMBER?

```java
@RequestMapping("/routing-slip-non-blocking-lambda")
public DeferredResult<String> nonBlockingRoutingSlip() throws IOException {

  final DeferredResult<String> dr = new DeferredResult<>();

  // Send request #1
  ListenableFuture<Response> execute = asyncHttpClient.execute(getUrl(1),
    (Response r1) -> {
      processResult(r1.getResponseBody()); // Process response #1
      asyncHttpClient.execute(getUrl(2),   // Send request #2
        (Response r2) -> {
          processResult(r2.getResponseBody()); // Process response #2
          asyncHttpClient.execute(getUrl(3),   // Send request #3
            (Response r3) -> {
              processResult(r3.getResponseBody()); // Process response #3
              asyncHttpClient.execute(getUrl(4),   // Send request #4
                (Response r4) -> {
                  processResult(r4.getResponseBody()); // Process response #4
                  asyncHttpClient.execute(getUrl(5),   // Send request #5
                    (Response r5) -> {
                      processResult(r5.getResponseBody()); // Process response #5
                      // Get the total result and set it on the deferred result
                      dr.setResult(getTotalResult());
                      ...
```

CALLISTA
— ENTERPRISE —

ROUTING SLIP - JAVA8

```java
@RequestMapping("/routing-slip-non-blocking-java8")
public DeferredResult<String> nonBlockingRoutingSlip() throws IOException {

    final DeferredResult<String> deferredResult = new DeferredResult<>();

    doAsyncCall(new ArrayList<>(), 1)
            .thenCompose(result -> doAsyncCall(result, 2))
            .thenCompose(result -> doAsyncCall(result, routeCall(result)))
            .thenCompose(result -> doAsyncCall(result, 5))
            .thenAccept(result -> deferredResult.setResult(getTotalResult(result)));


    return deferredResult;
}

private CompletableFuture<List<String>> doAsyncCall(List<String> result, int num) {
    String url = getUrl(num);

    final CompletableFuture<List<String>> newResult = asyncHttpClientJava8
            .execute(url, num)
            .thenApply(response -> getResponseBody(response))
            .thenApply(body -> addBodyToResult(body, result));

    return newResult;
}
```

CALLISTA
— ENTERPRISE —

SUMMARY - JAVA 8

- Sequential programming model

- No framework needed

- Big api

- Stream and CompletableFuture api differs

CALLISTA
— ENTERPRISE —

SYNCHRONOUS VS ASYNCHRONOUS

	Single item	Multiple items
Synchronous	T get ()	Iterable<T> get()
Asynchronous	Future<T> get()	Observable<T> get()

REACTIVE EXTENSIONS

msdn.microsoft.com/en-us/data/gg577609

Data Developer Center

Search Data with Bing

bing

Home   Library   Learn   Downloads   Support   Community

Documentation   Videos   Articles   Books   Hands-on Labs   Webcasts

Data Developer Center > Learn > **Reactive Extensions**

The Reactive Extensions (Rx)...

...is a library to compose asynchronous and event-based programs using observable collections and LINQ-style query operators.

CALLISTA
— ENTERPRISE —

REACTIVE EXTENSIONS

http://reactivex.io/

The Observer pattern done right

ReactiveX is a combination of the best ideas from
the Observer pattern, the Iterator pattern, and functional programming

CALLISTA
— ENTERPRISE —

REACTIVE EXTENSIONS

Functional

Avoid intricate stateful programs, using clean input/output functions over observable streams.

Less is more

ReactiveX's operators often reduce what was once an elaborate challenge into a few lines of code.

Async error handling

Traditional try/catch is powerless for errors in asynchronous computations, but ReactiveX is equiped with proper mechanisms for handling errors.

Concurrency made easy

Observables and Schedulers in ReactiveX allow the programmer to abstract away low-level threading, synchronization, and concurrency issues.

POLYGLOT

ReactiveX | Getting started | Docs ▾ | Languages ▾ | Resources ▾ | Community ▾

Languages

- Java: RxJava
- JavaScript: RxJS
- C#: Rx.NET
- C#(Unity): UniRx
- Scala: RxScala
- Clojure: RxClojure
- C++: RxCpp
- Ruby: Rx.rb
- Python: RxPY
- Groovy: RxGroovy
- JRuby: RxJRuby
- Kotlin: RxKotlin

ReactiveX for platforms and frameworks

- RxNetty
- RxAndroid

CALLISTA
— ENTERPRISE —

REACTIVE EXTENSIONS

OBSERVABLE

Observable

onNext()
onError()
onCompleted()

Observer
+notify()

Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()

ConcreteObserverA
+notify()

ConcreteObserverB
+notify()

notifyObservers()
 for observer in observerCollection
  call observer.notify()

CALLISTA
— ENTERPRISE —
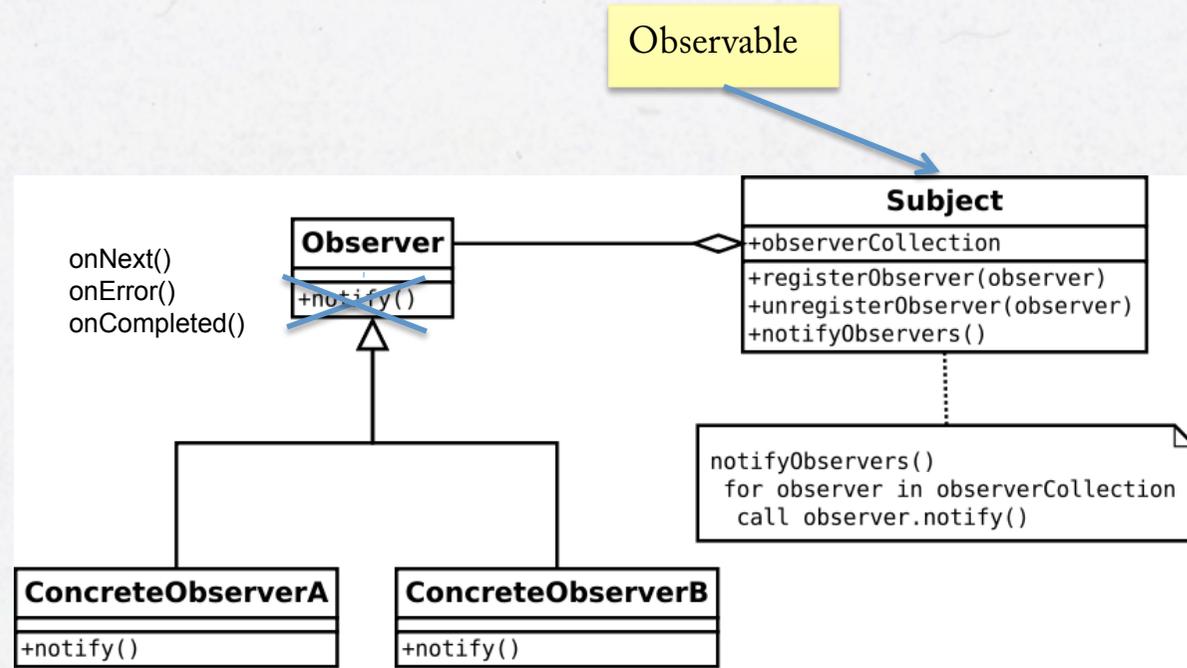
ITERABLE VS. OBSERVABLE

event	Iterable (pull)	Observable (push)
retrieve data	T next()	onNext(T)
discover error	throws Exception	onError(Exception)
complete	!hasNext()	onCompleted()

CALLISTA
— ENTERPRISE —

ITERABLE VS. OBSERVABLE

Observable "Push based"	Iterable (Java Stream) "Pull based"
| ```
getData()
 .skip(10)
 .take(5)
 .map(s -> s + "transformed")
 .forEach(System.out::println);
``` | ```
getData()
   .skip(10)
   .limit(5)
   .map(s -> s + "transformed")
   .forEach(System.out::println);
``` |

CALLISTA
— ENTERPRISE —

REACTIVE EXTENSIONS

- Lightweight (only one jar-file).
- Flexible in implementation of Observables.
- Changing implementation does not break Observers.
- Polyglot

MARBLE DIAGRAMS

This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

This vertical line indicates that the Observable has completed successfully.

These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

flip

This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

BORING...

CALLISTA
— ENTERPRISE —

CREATE AN OBSERVABLE

```java
public Observable<Response> observable(String url, String acceptHeader) {
    return Observable.create(observer -> {

        try {
            asyncHttpClient.prepareGet(url).execute(new AsyncCompletionHandler<Response>() {
                @Override
                public Response onCompleted(Response response) throws Exception {
                    observer.onNext(response);
                    observer.onCompleted();
                    return response;
                }


                @Override
                public void onThrowable(Throwable t) {
                    observer.onError(t);
                }
            });
        } catch (Exception e) {
            observer.onError(e);
        }
    });
}
```
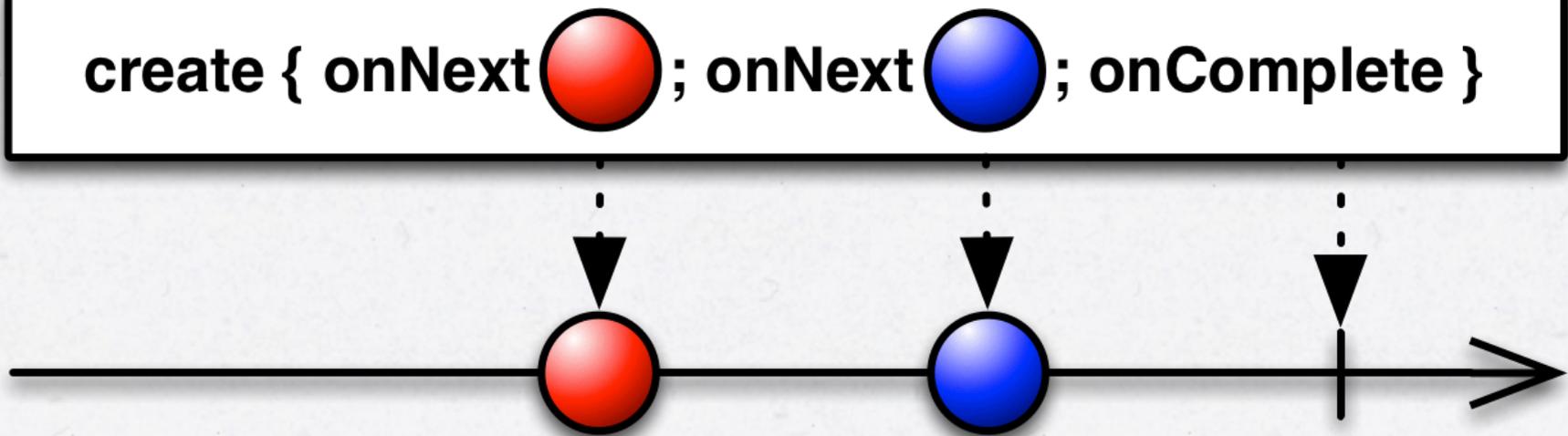
CALLISTA
— ENTERPRISE —

STILL BORING

CALLISTA
— ENTERPRISE —

AGGREGATOR EXAMPLE

CALLISTA
— ENTERPRISE —
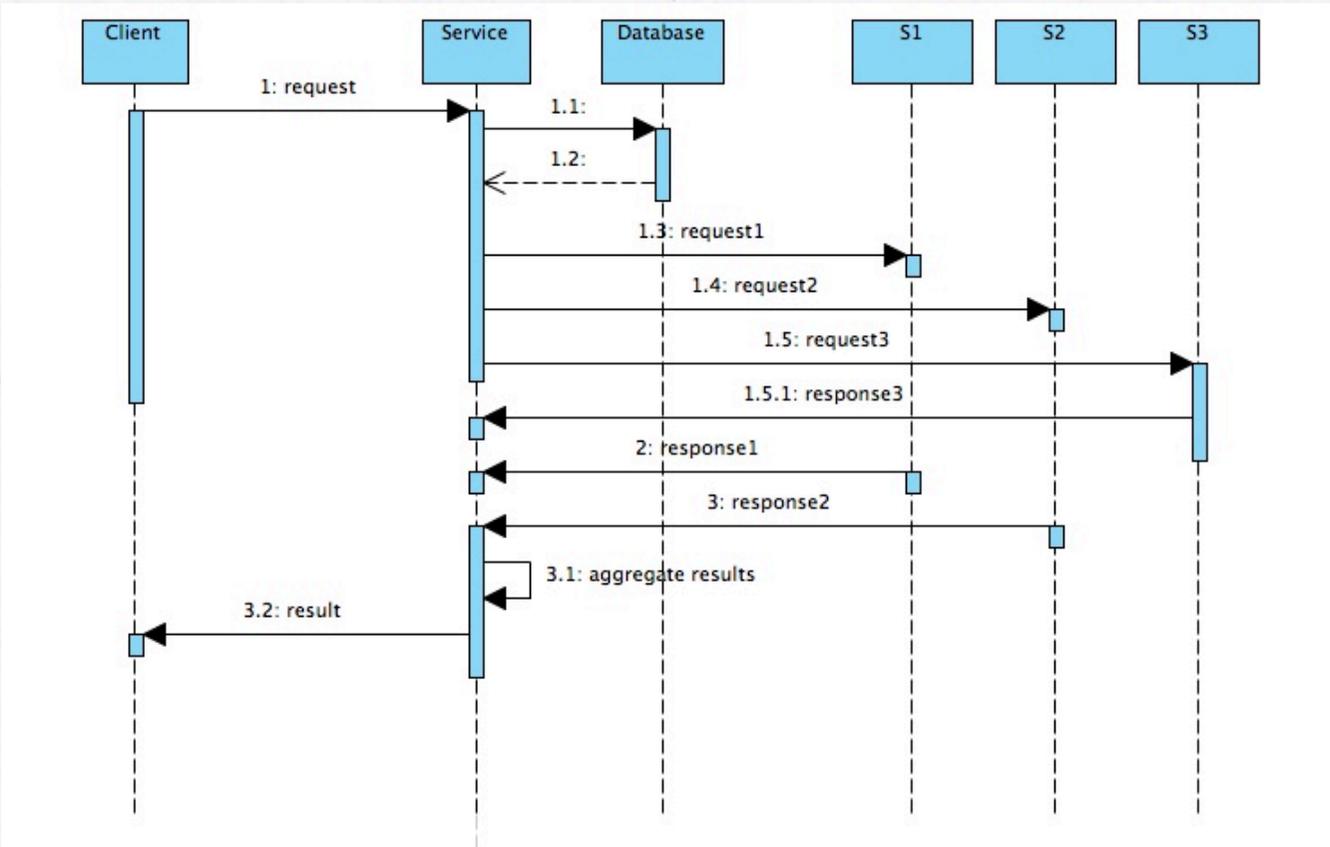
AGGREGATOR EXAMPLE

```java
DbLookup dbLookup = new DbLookup(dbLookupMs, dbHits);
DeferredResult<String> deferredResult = new DeferredResult<>();

Subscription subscription =
    Observable.from(dbLookup.lookupUrlsInDb(SP_NON_BLOCKING_URL, minMs, maxMs))
        .subscribeOn(Schedulers.from(dbThreadPoolExecutor))
        .observeOn(Schedulers.io())
        .flatMap(request ->
            asyncHttpClientRx
                .observable(request.url, accept)
                .map(this::getResponseBody)
        )
        .observeOn(Schedulers.computation())
        .buffer(dbHits)
        .subscribe(v -> deferredResult.setResult(getTotalResult(v)));

deferredResult.onCompletion(subscription::unsubscribe);
return deferredResult;
```

CALLISTA
— ENTERPRISE —

CALLISTA
— ENTERPRISE —

CALLISTA
— ENTERPRISE —
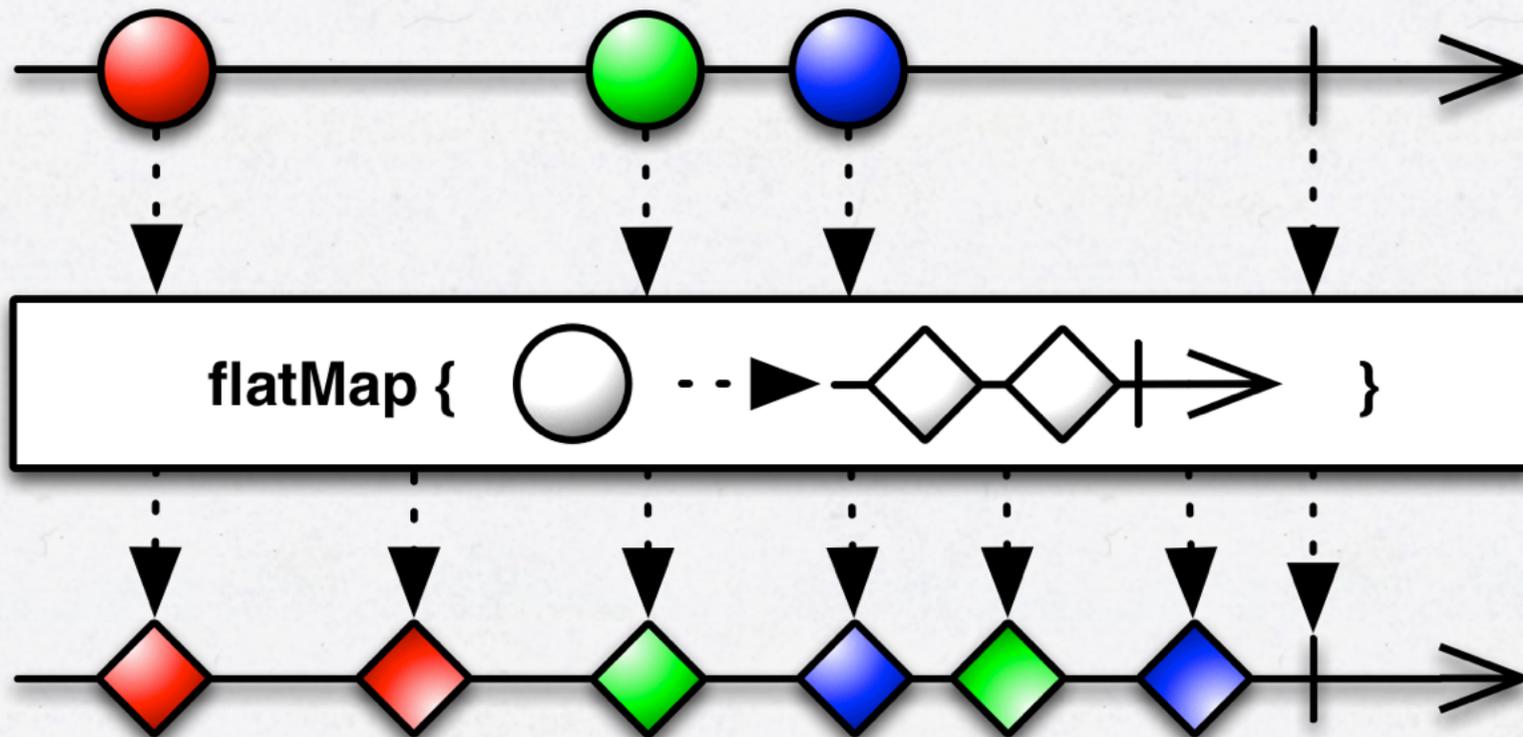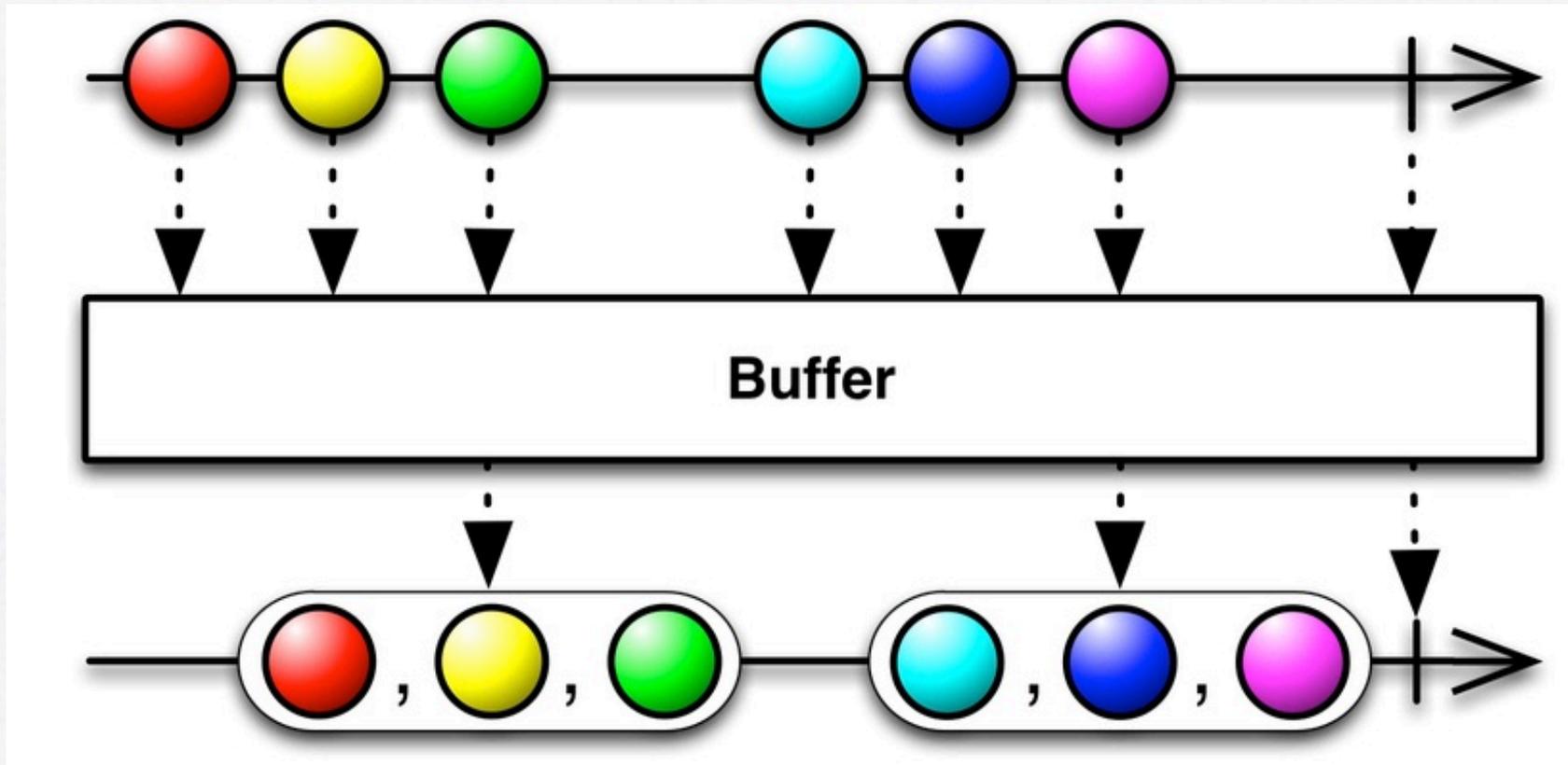
JUST AWESOME

ROUTING SLIP EXAMPLE
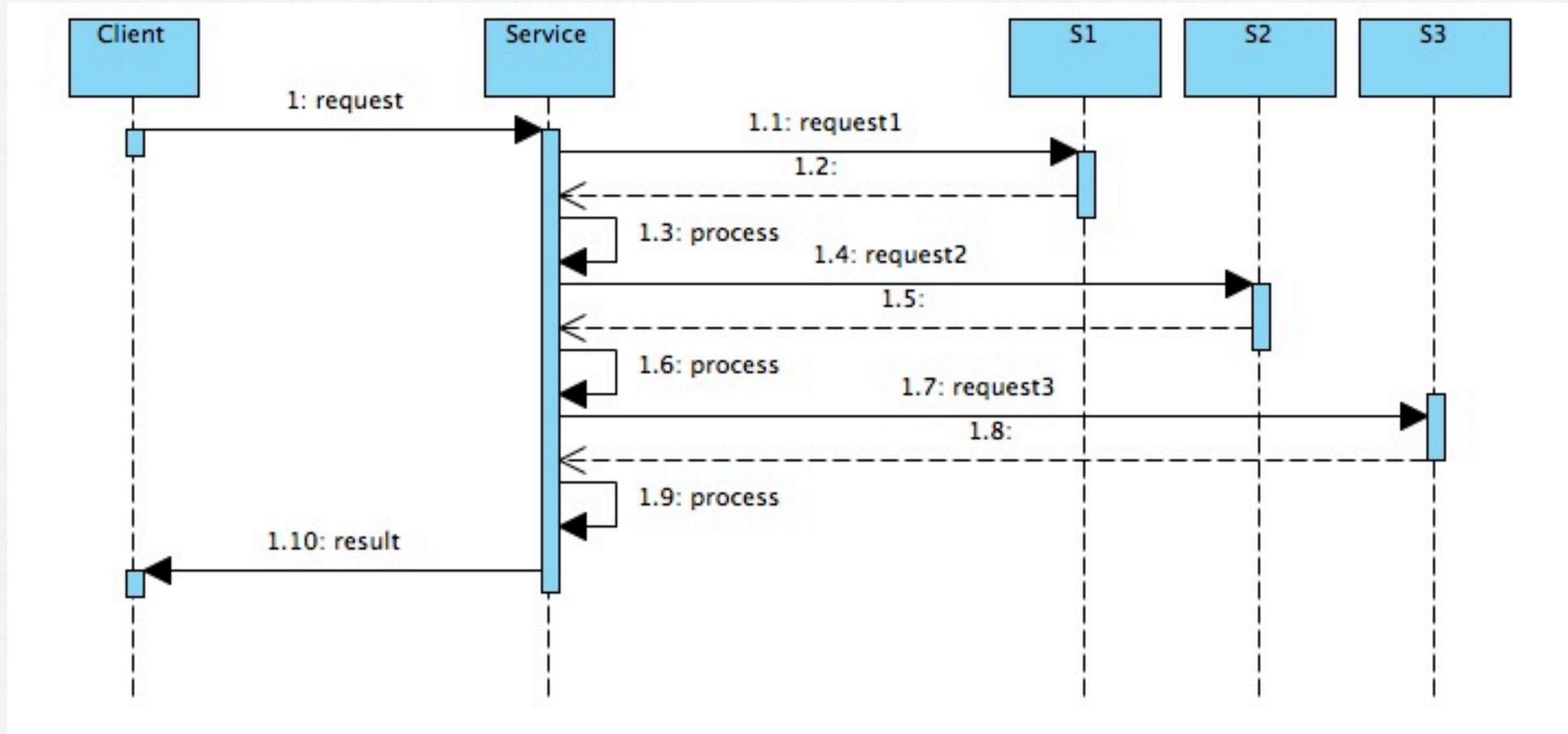
ROUTING SLIP EXAMPLE

```java
public DeferredResult<ResponseEntity<String>> nonBlockingRoutingSlip(...) {
    DeferredResult<ResponseEntity<String>> deferredResult = new DeferredResult<>();

    Subscription subscription = Observable.just(new Result())
            .flatMap(result -> doAsyncCall(result, 1))
            .flatMap(result -> doAsyncCall(result, 2))
            .flatMap(result -> doAsyncCall(result, routeCall(result)))
            .flatMap(result -> doAsyncCall(result, 5))
            .subscribe(
                    result -> deferredResult.setResult(result.getTotalResult()),
            );
    deferredResult.onCompletion(subscription::unsubscribe);
    return deferredResult;
}


private Observable<Result> doAsyncCall(Result result, int num) {
    return asyncHttpClientRx
            .observable(getUrl(num))
            .map(resp -> result.processResponse(resp))
}
```

CALLISTA
— ENTERPRISE —

CONCLUSIONS

CALLISTA
— ENTERPRISE —
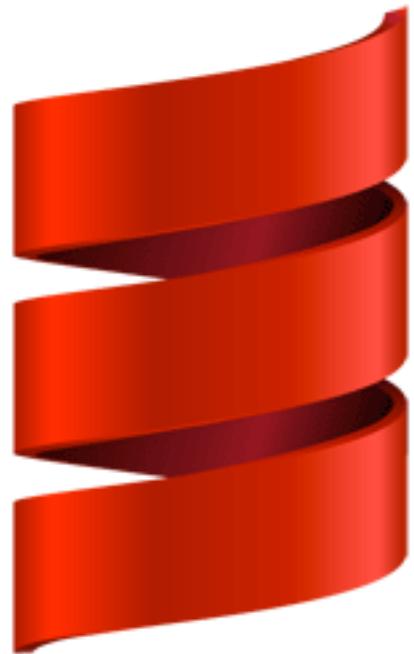
AGGREGATOR EXAMPLE
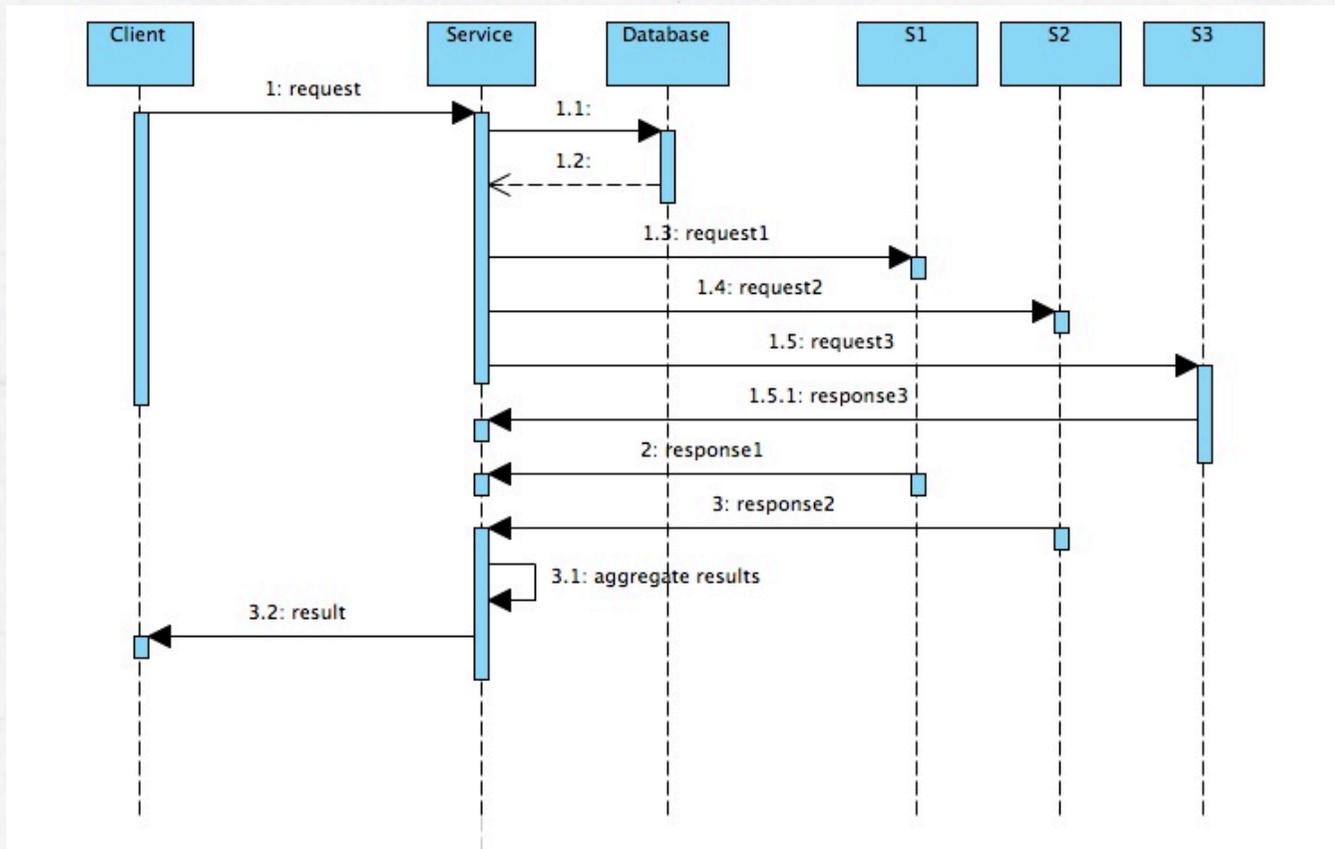
AGGREGATOR - SCALA

```scala
@RequestMapping(Array("/aggregate-non-blocking-scala"))
def nonBlockingAggregator(...) maxMs: Int): DeferredResult[String] = {

    val deferredResult = new DeferredResult[String]
    val urlsF: Future[List[String]] = doDbLookup(dbLookupMs, dbHits, minMs, maxMs)(ExecutionContext.fromExecutor(taskExecutor))

    val resultsF: Future[List[String]] = urlsF.flatMap { urls =>
        sequence(
            urls.map(url => asyncCall(url)) //List[Future[String]]
        )
    }

    resultsF.map(results => deferredResult.setResult(results.mkString("\n")))
    deferredResult
}

def asyncCall(url: String): Future[String] = AsyncHttpClientScala.get(url).map(response => response.getResponseBody)
```

ROUTING SLIP EXAMPLE
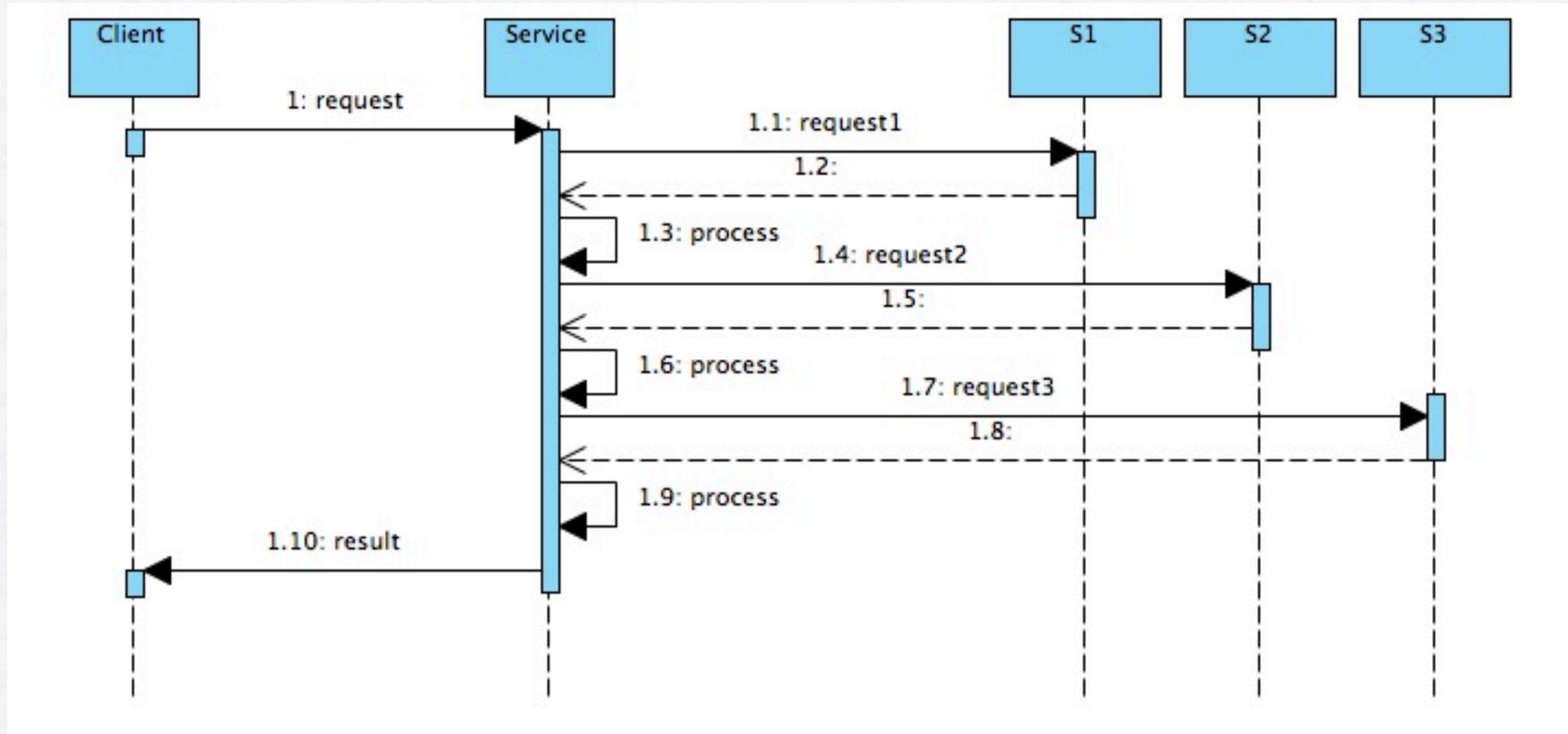
ROUTING SLIP - SCALA

```scala
@RequestMapping(Array("/routing-slip-non-blocking-scala"))
def nonBlockingRoutingSlip: DeferredResult[String] = {
    val deferredResult = new DeferredResult[String]()
    val result =
        for {
            r1 <- doAsyncCall(1)
            r2 <- doAsyncCall(2)
            r3 <- doAsyncCall(routeCall(r2))
            r4 <- doAsyncCall(5)
        } yield List(r1, r2, r3, r4)
    result.map(v => deferredResult.setResult(v.mkString("\n")))
    deferredResult
}

def doAsyncCall(num: Integer) = AsyncHttpClientScala.get(nonBlockingUrl(num)).map(r => r.getResponseBody)
```

CALLISTA
— ENTERPRISE —

PLAY FRAMEWORK

PLAY FRAMEWORK

- Developer friendly
- Stateless web tier
- Non-blocking I/O
- Build on Akka
- Real-time enabled
- Restful by default
- Websockets, Comet, EventSource

CALLISTA
— ENTERPRISE —

CALLISTA
— ENTERPRISE —

PLAY AGGREGATE

```scala
def aggregate(dbLookupMs: Int, dbHits: Int, minMs: Int, maxMs: Int) = Action.async {

  val dbLookup = new DbLookup(dbLookupMs, dbHits)

  val urlsF = Future{
    dbLookup.lookupUrlsInDb(SP_NON_BLOCKING_URL, minMs, maxMs).asScala
  }(Contexts.simpleDbLookups)

  urlsF.flatMap { urls => // List[String]
    sequence(
      urls.map { url =>
        WS.url(url).get().map(r => r.body) // List[Future[String]]
      }
    ).map(v => Ok(v.mkString("", "\n", "\n")))
  }
}
```

PLAY ROUTING SLIP

```scala
def routingSlip = Action.async {

  def processResult(s: String) = true

  def getUrl(processingStepNo: Int) = {
    val sleeptimeMs = 100 * processingStepNo
    s"$SP_NON_BLOCKING_URL?minMs=$sleeptimeMs&maxMs=$sleeptimeMs"
  }

  def doAsyncCall(num: Int) = WS.url(getUrl(num)).get().map(r => (r.body, processResult(r.body)))

  val result =
    for {
      (r1, next) <- doAsyncCall(1)
      (r2, next) <- doAsyncCall(2)
      (r3, next) <- doAsyncCall(if (next) 4 else 3)
      (r4, next) <- doAsyncCall(5)
    } yield List(r1, r2, r3, r4)

  result.map(v => Ok(v.mkString("", "\n", "\n")))
}
```

CALLISTA
— ENTERPRISE —
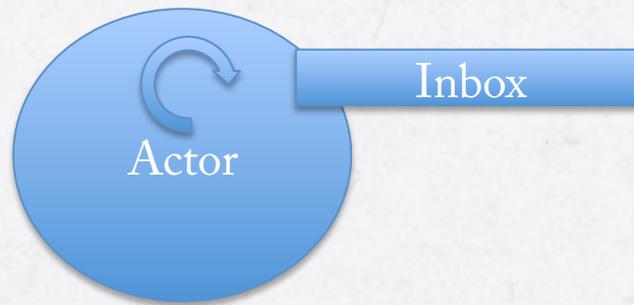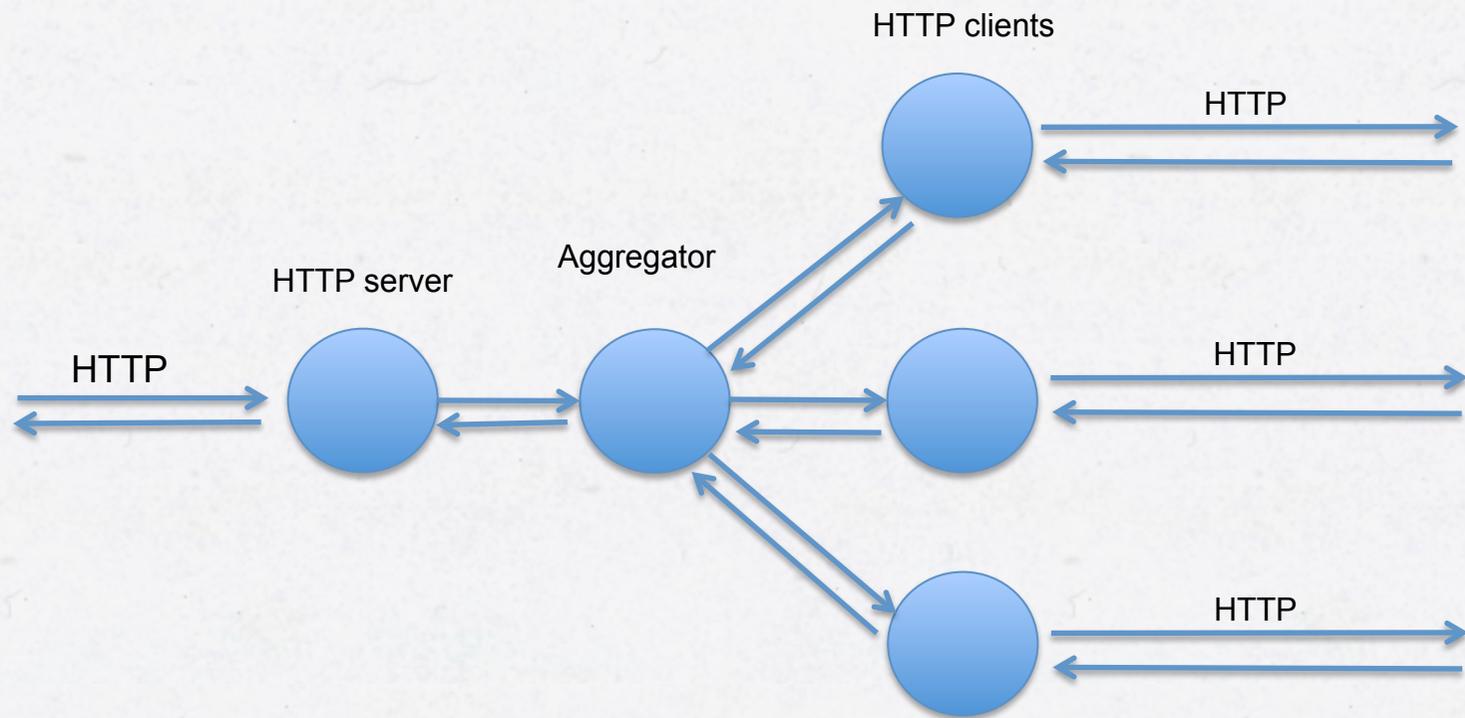
AKKA ROUTING SLIP
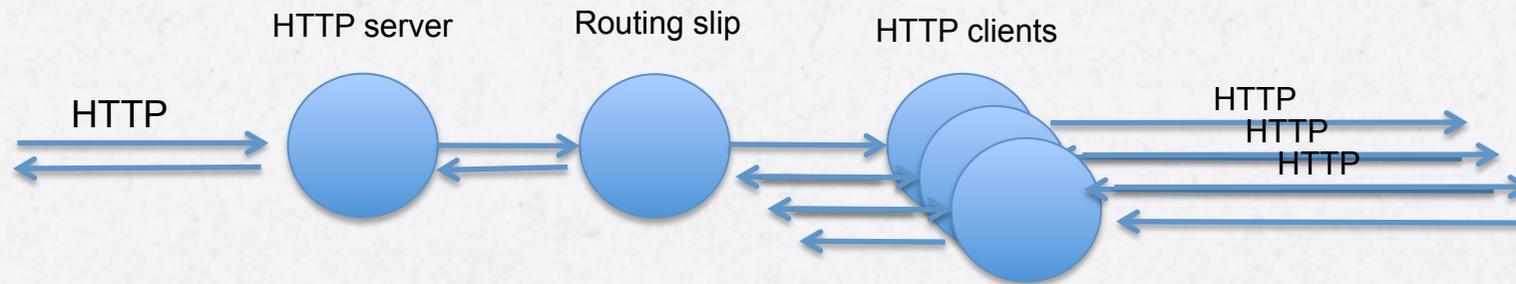
Summary and next step

CALLISTA
— ENTERPRISE —

SUMMARY

- We have seen…
  - If you need scalability and resilience its time to say goodbye to Blocking I/O!
  - The callback model is simple to get started with
    - » …but gets easily very complex…

  - Reactive and Functional programming to the rescue!

  - Makes it possible to program asynchronous logic in a sequential way
    - » Composable Futures is a key feature!

  - A lot of alternatives exists…
    - » We have looked at Java 8, RX Java and Scala/Akka (more exists…)
    - » What to choose?

CALLISTA
— ENTERPRISE —

SUMMARY

- What to choose?

  - Start with simple cases and the *callback* model to warm up…

  - When you need it and are ready for *reactive* and *functional* programming:

    » Choose Java 8 and Completable Futures if
      ‣ You must/want to avoid 3'rd party libraries
      ‣ Watch out for the bloated API!

    » Choose RX Java otherwise
      ‣ Clean abstraction and API
      ‣ Supports several languages (JS, .Net et al)
      ‣ Runs on Java 7 (but you have to live without lambdas then ☺)

    » If you are ready to switch language take a look at Scala…

CALLISTA
— ENTERPRISE —