

# Reactive Streams

because parallelism matters

@humbertostreb

# Humberto Streb

- software developer
- falling in love with distributed systems
- half-assed goalkeeper ;(

[fb.com/humbertostreb](https://fb.com/humbertostreb)

[twitter.com/humbertostreb](https://twitter.com/humbertostreb)

[github.com/hstreb](https://github.com/hstreb)



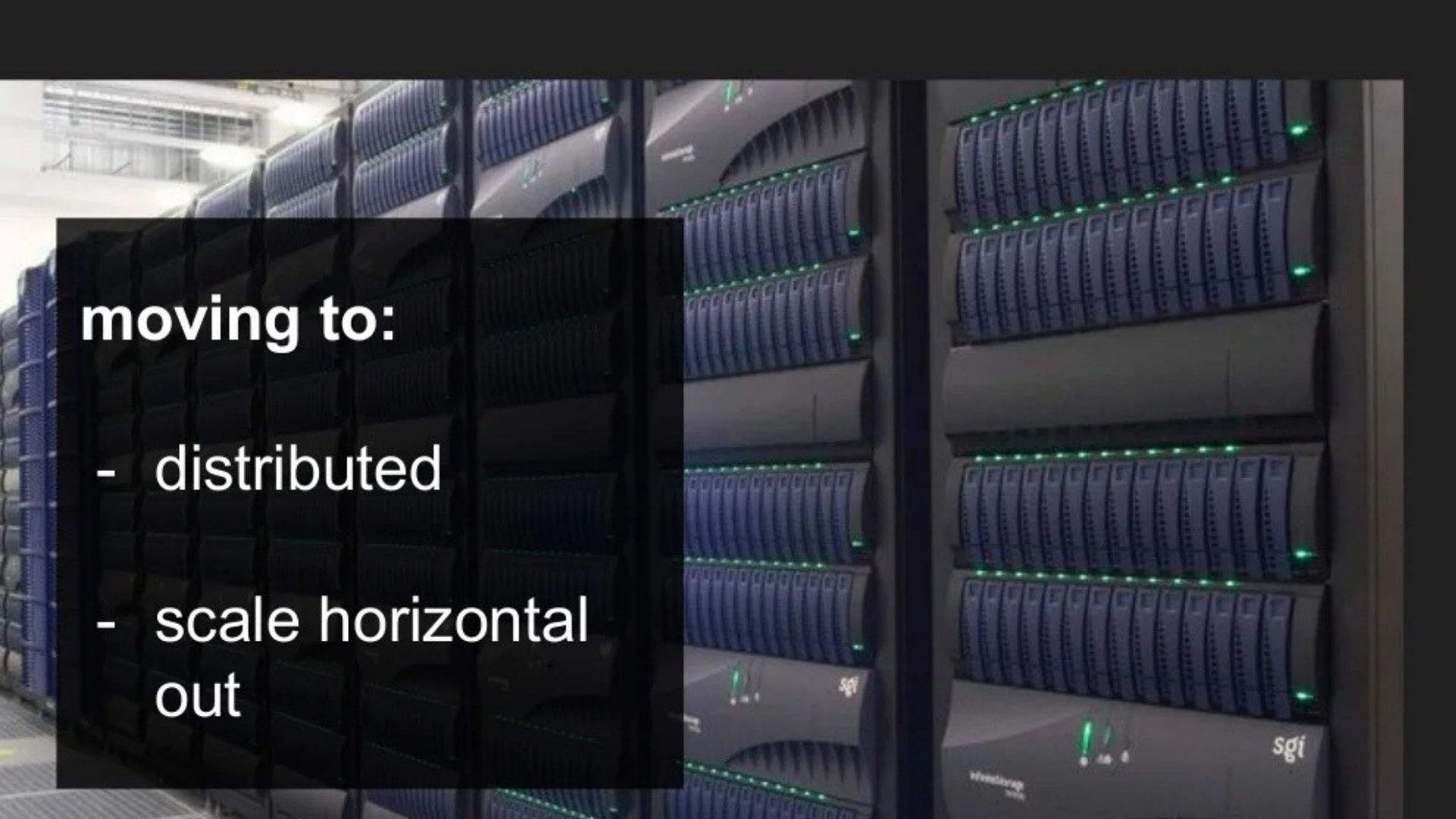
The background of the image is a light gray, textured surface featuring faint, repeating patterns of ancient Egyptian hieroglyphs. A prominent rectangular frame with a thin black border is centered on the image. Inside this frame, the word "history" is written in a bold, black, sans-serif font.

history

**From:**

- single node
- vertical scale up



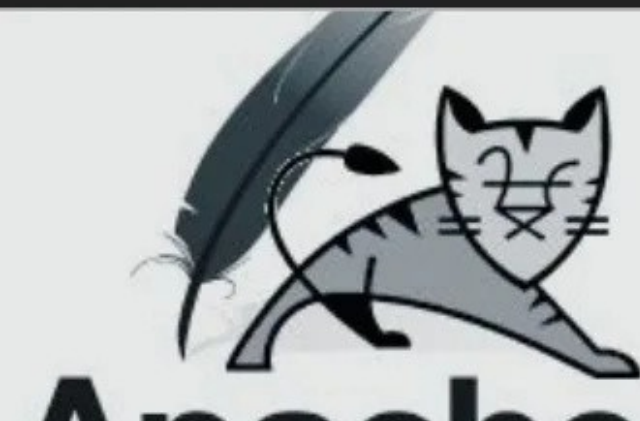


**moving to:**

- distributed
- scale horizontal  
out

# From:

## Applications Servers





# moving to: Microservices

O'REILLY

www.it-ebooks.info

## Building Microservices

DESIGNING FINE-GRAINED SYSTEMS




Sam Newman

Copyrighted Material

What I need to  
change in my  
codebase?







# **Async - Threads**

## Threads are powerful, but:

- low level abstraction
- imperative style
- demand more knowledge from the java API (lock, synchronized, ...)

# lock

```
final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();  
lock.writeLock().lock();  
try {  
    variable.complexAlgorithm();  
} finally {  
    lock.writeLock().unlock();  
}
```

# synchronized

```
synchronized(variable) {  
    variable.complexAlgorithm();  
}
```

...

```
synchronized(a.variable) {  
    consume(a.variable);  
}
```

A dramatic rescue scene on a body of water. A large red ship, possibly a ferry or tugboat, is partially submerged and tilted at a steep angle. A rescuer in a red jacket and black pants is carrying a person on their back, suspended in the air by a rope or cable. The water is choppy and grey. The text "The rescue" is overlaid in white.

**The rescue**

# Reactive Streams

PUBLISH

**SUBSCRIBE**





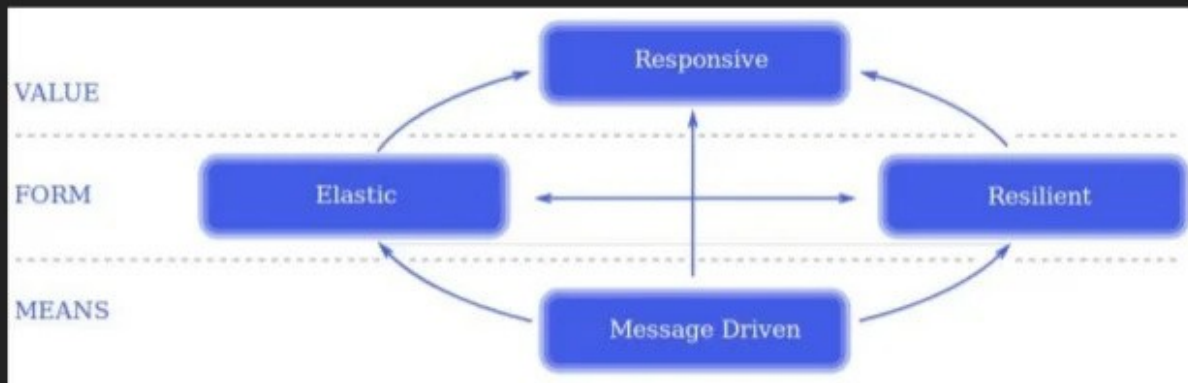
# Reactive Streams

- asynchronous stream processing
- non-blocking IO
- back pressure

<http://www.reactive-streams.org/>

# Reactive Manifesto

- responsive
- resilient
- elastic
- message driven



# From Imperative to Reactive Programming

- Composability and readability
- Data as a flow manipulated with a rich vocabulary of operators
- Nothing happens until you subscribe
- Backpressure
- High level but high value abstraction that is concurrency-agnostic

# Implementations

- [akka streams](#)
  - based in actors
  - rich environment (akka)
    - cluster
    - TCP
    - akka persistence
    - Alpakka

# Implementations

- [RxJava](#)
  - netflix oss

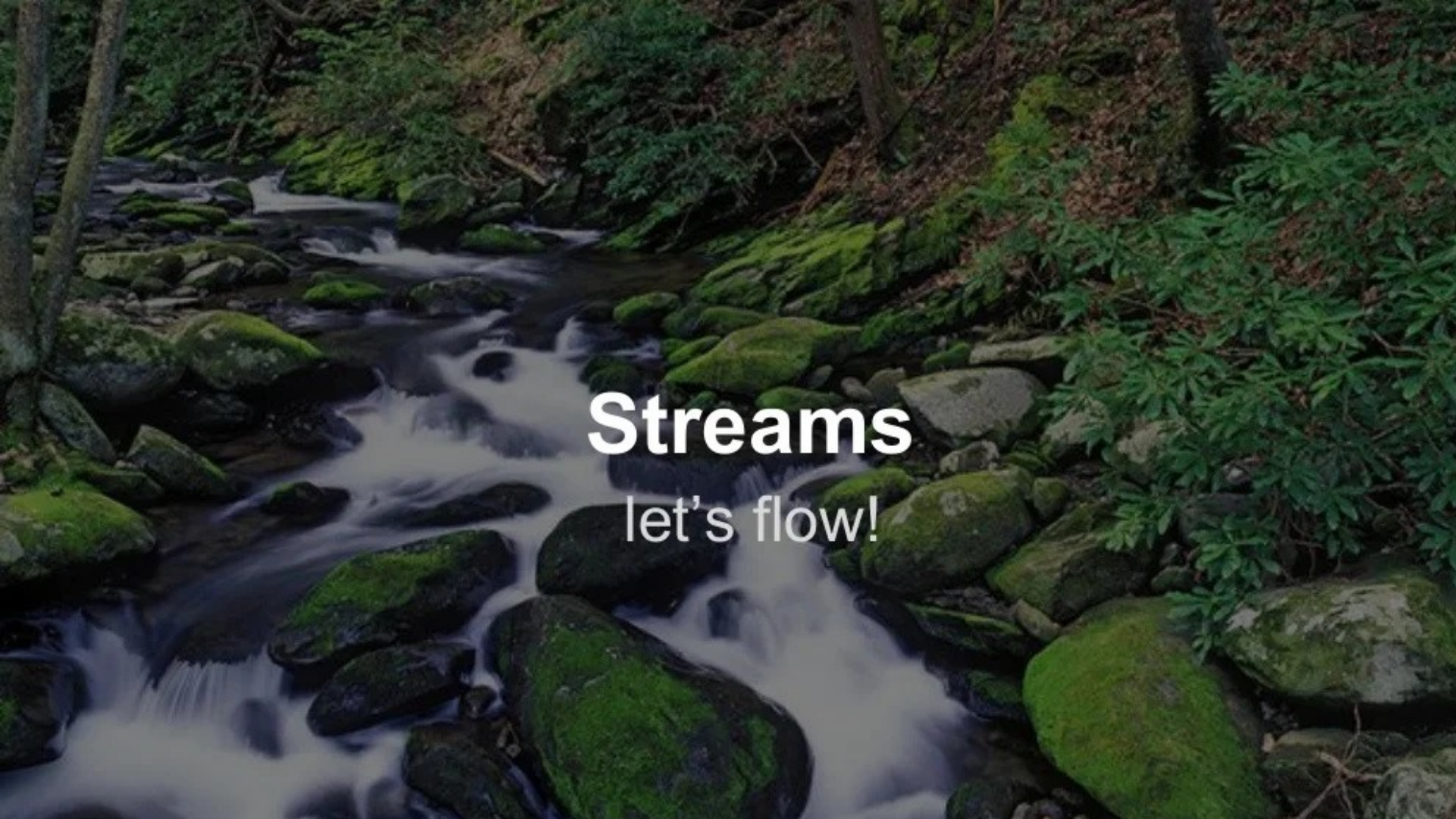
# Implementations

- [Reactor](#)
  - Spring support



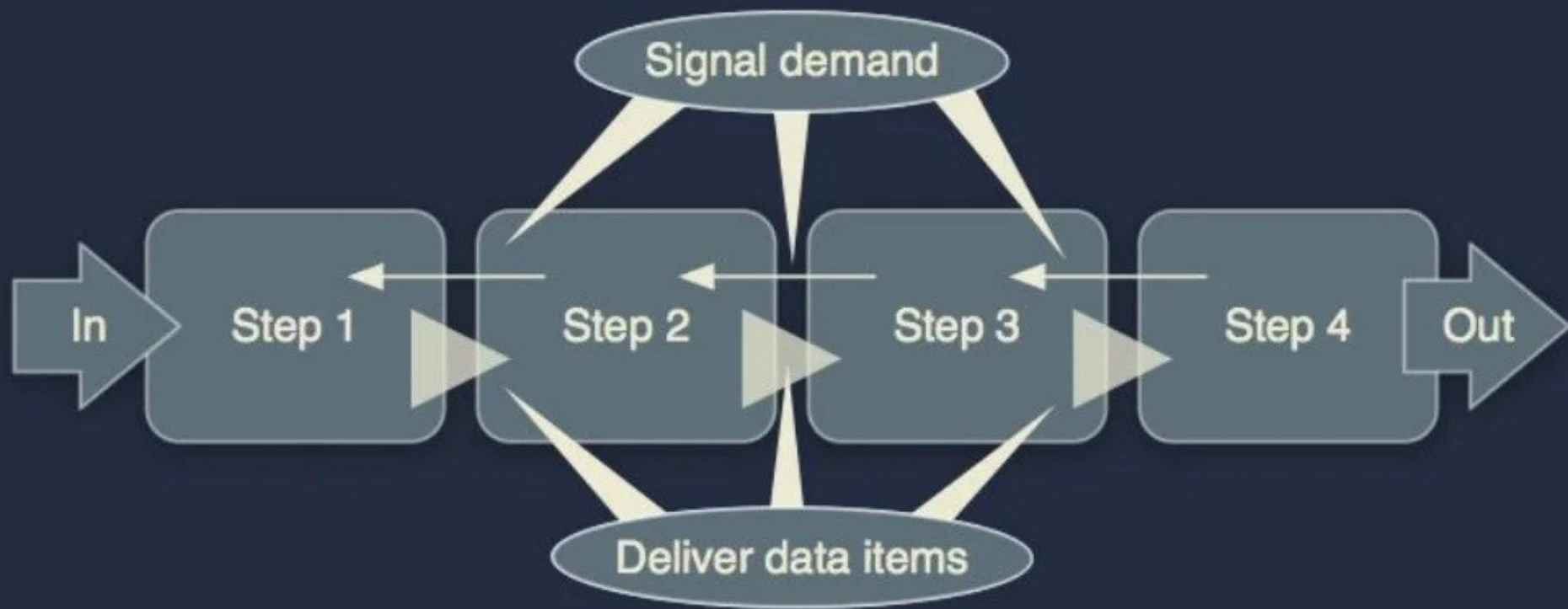
# Implementations

- [Java 9](#)

A long-exposure photograph of a stream flowing over mossy rocks in a forest. The water is blurred, creating a sense of motion. The rocks are covered in vibrant green moss. The surrounding forest is dense with green foliage and trees.

# Streams

let's flow!



# When use Reactive Streams

- it's not about be fast, it's about be efficient
- racionalize resources consumption
- support high demand

# Project Reactor



# Reactor

- **Mono<T>** an Asynchronous 0-1 Result
- **Flux<T>** an Asynchronous Sequence of 0-n Items



# Creating

```
Mono<Integer> mono = Mono.just(1);
```

```
Mono<Integer> monoEmpty = Mono.empty();
```

```
Flux<Integer> flux = Flux.range(0, 10);
```

```
Flux<String> foo = Flux.just("foo", "bar");
```

```
Flux<String> fromIterable = Flux.fromIterable(iterable);
```

# Extensive number of functions

- `map()`
- `filter()`
- `repeat()`
- `replay()`
- `retry()`
- `skip()`, `skipUntil()`, `skipWhile()` ...
- `switchIfEmpty()`
- `take()`, `takeUntil()`, `takeWhile()` ...
- `zip()`

...

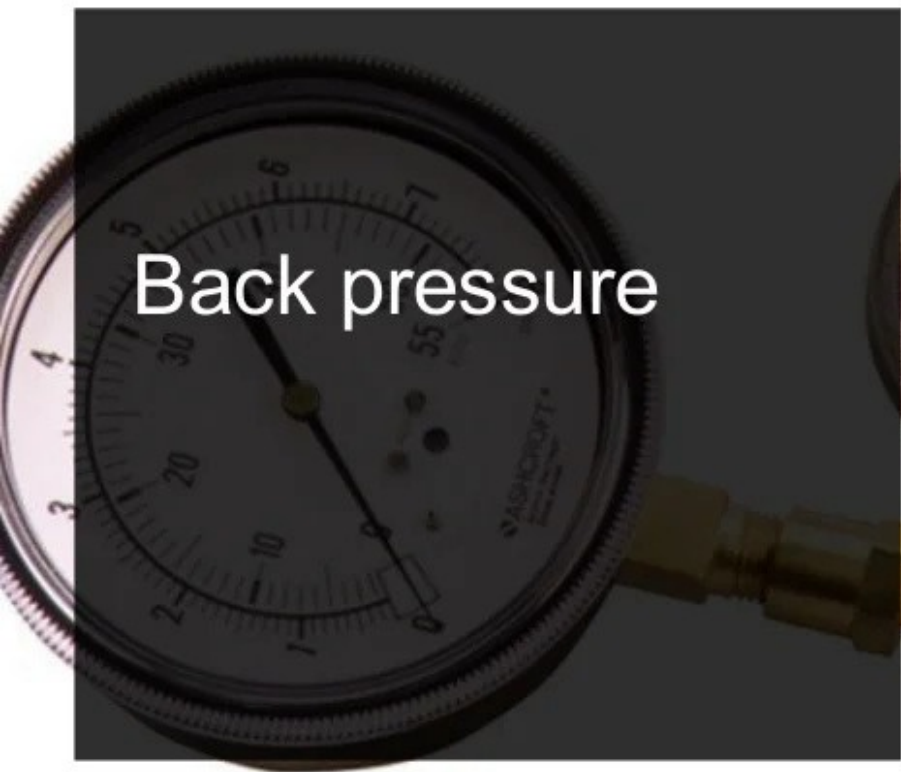
# Async

```
Flux.defer(() -> Flux.fromIterable(repository.findAll()))  
    .subscribeOn(Schedulers.elastic());
```

...

```
flux.publishOn(Schedulers.parallel())  
    .doOnNext(repository::save)  
    .then();
```

Back pressure



```
Flux<Long> flux = Flux.interval(Duration.ofMillis(100))  
    .take(100);
```

flux

```
.buffer(Duration.ofMillis(10))  
.tolerable()  
.forEach(System.out::println);
```

flux

```
.buffer(Duration.ofMillis(200))  
.tolerable()  
.forEach(System.out::println);
```

// faster consumer

[0]

[1]

[2]

...

// slow consumer

[1, 2]

[3, 4]

[5, 6]

...



# Testing

@Test

```
public void testAppendBoomError() {  
    Flux<String> source = Flux.just("foo", "bar");
```

```
    StepVerifier.create(  
        appendBoomError(source))  
        .expectNext("foo")  
        .expectNext("bar")  
        .expectErrorMessage("boom")  
        .verify();  
}
```

# More

- Spring 5.0 with reactive support

<https://www.brighttalk.com/webcast/14893/277207>

- Deal with JDBC blocking

<https://dzone.com/articles/spring-5-webflux-and-jdbc-to-block-or-not-to-block>

# links

- <https://www.reactivemanifesto.org/>
- <https://blog.redelastic.com/a-journey-into-reactive-streams-5ee2a9cd7e29>
- <https://www.infoq.com/articles/reactor-by-example>
- <https://spring.io/blog/2016/07/28/reactive-programming-with-spring-5-0-m1>
- <http://musigma.org/java/2016/11/21/reactor.html>

# Thanks

## Reactive Streams

because parallelism matters

@humbertostreb