CLAIRVOYANT

design engineer deliver

Reactive Spring

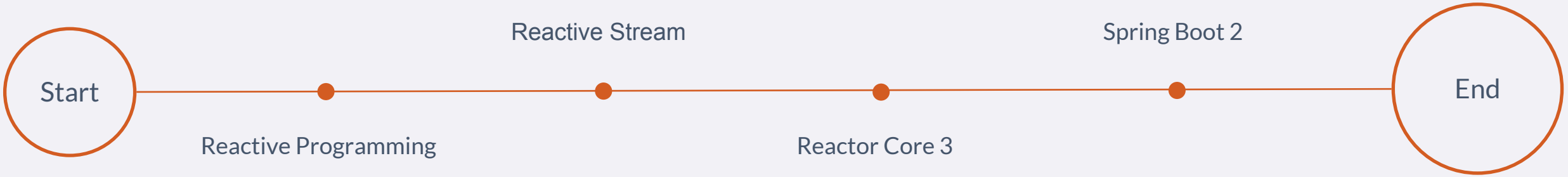
Team
Clairvoyant India Pvt. Ltd.

CLAIRVOYANT

Disclaimer

I am still learning.
— Michelangelo

Agenda

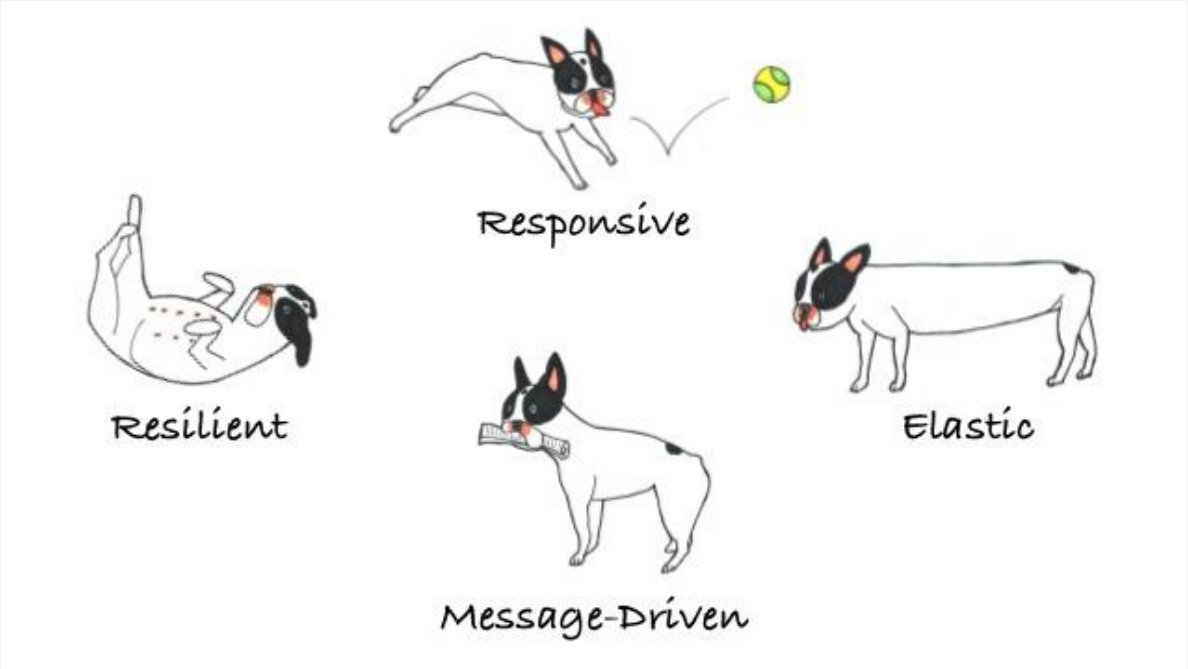

Start
Reactive Programming
Reactive Stream
Reactor Core 3
Spring Boot 2
End

Introduction

Introduction

	~10 years Ago	Now
No of server nodes	10’s	1000s
Response Time	Seconds	Milliseconds
Maintenance Time	Hours	None
Data Volume	GBs	TBs -> PBs
Consumer	Mostly Web	Web, Mobile

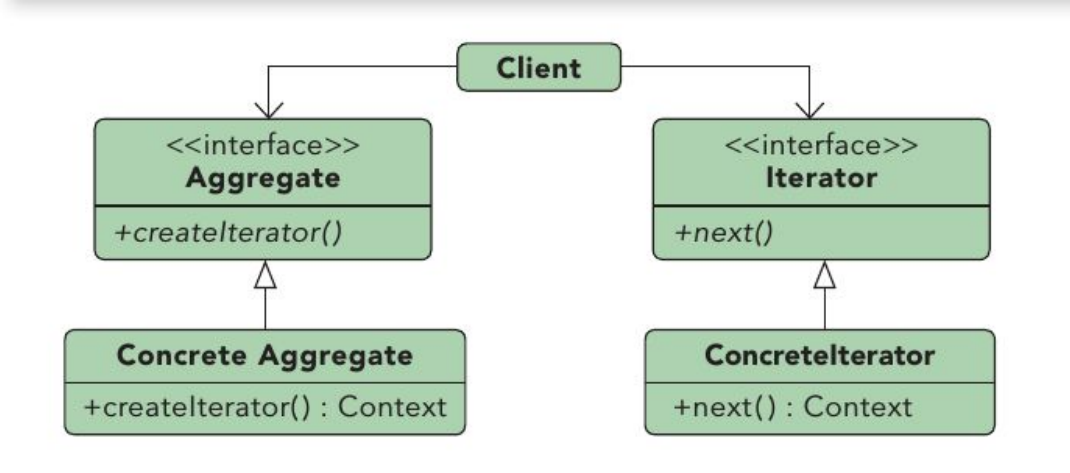
New architecture



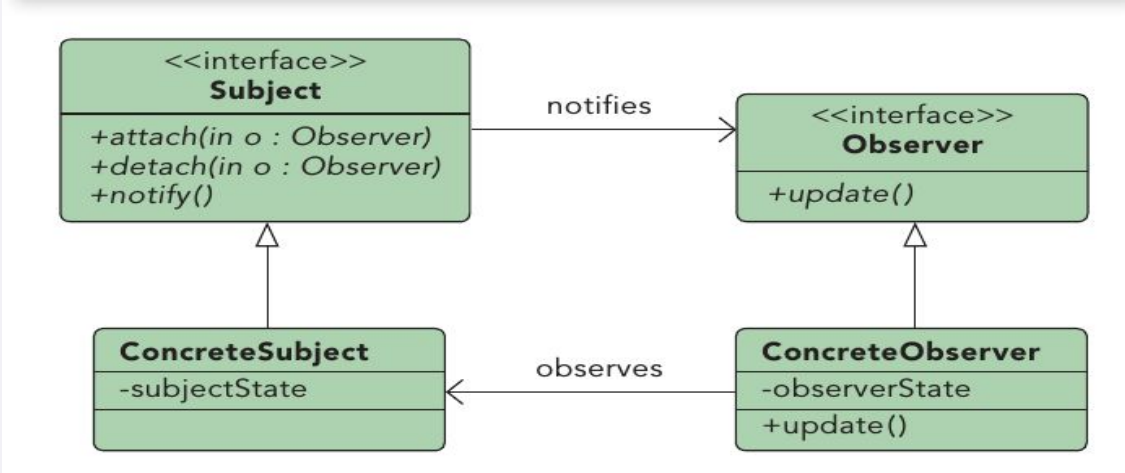
What is Reactive Programming ?

In computing, reactive programming is an asynchronous programming paradigm oriented around data streams and the propagation of change. ---- Wikipedia

What is Reactive Programming ?

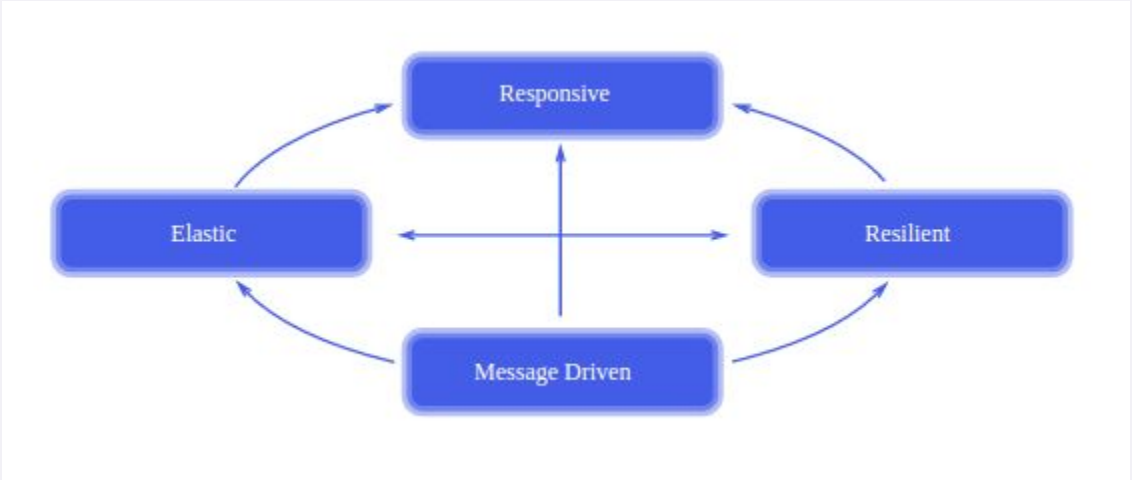


Iterator Pattern

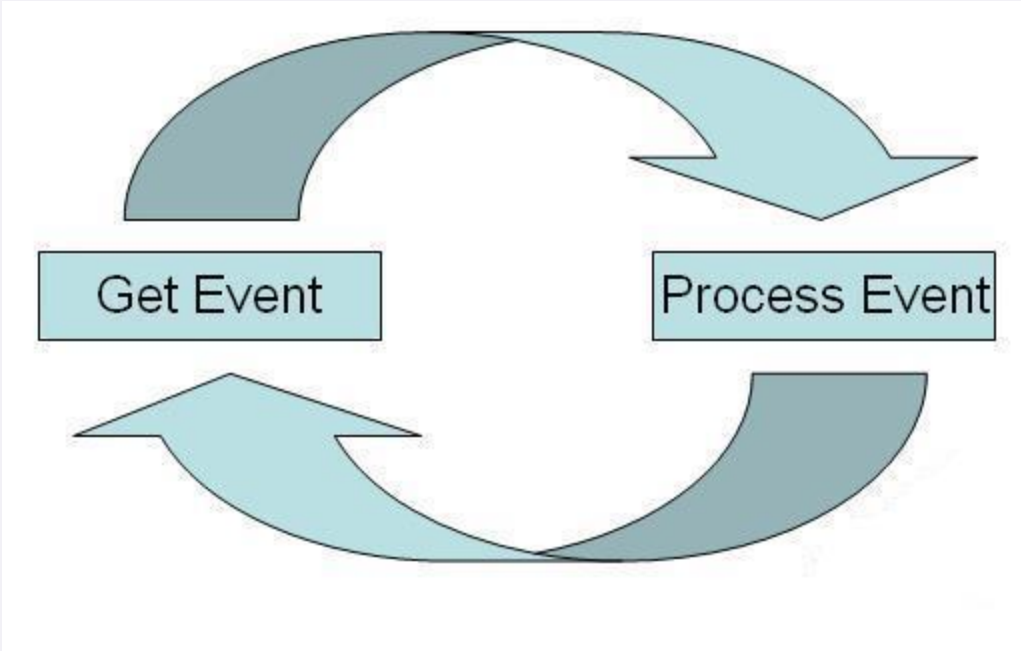


Observer Pattern

Reactive Manifesto



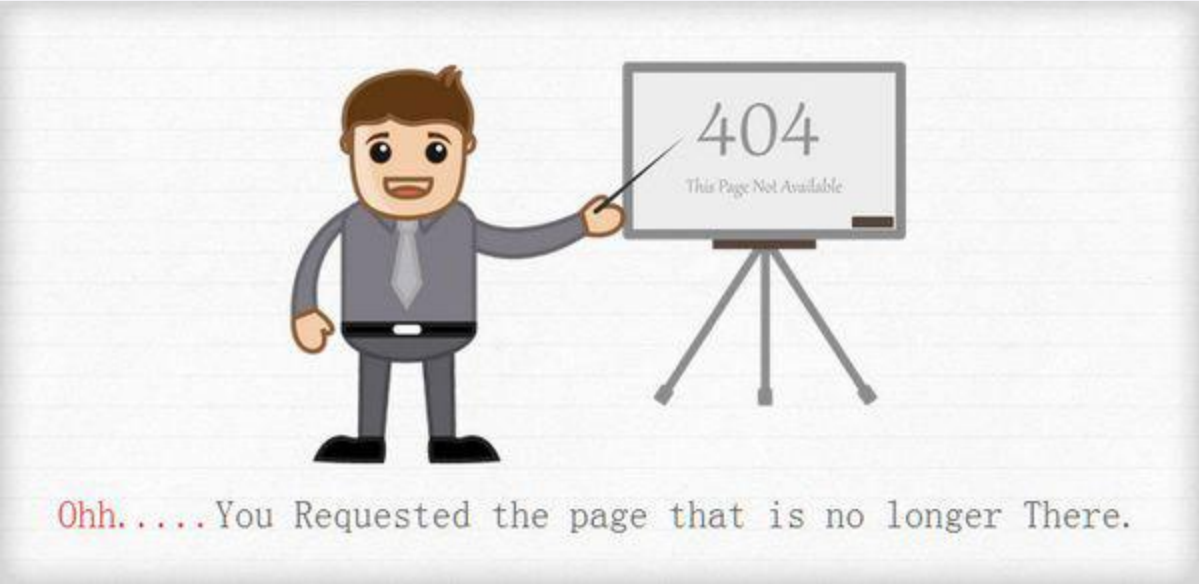
Event Driven



Scalable

Resilient

Responsive


404
This Page Not Available
Ohh.....You Requested the page that is no longer There.

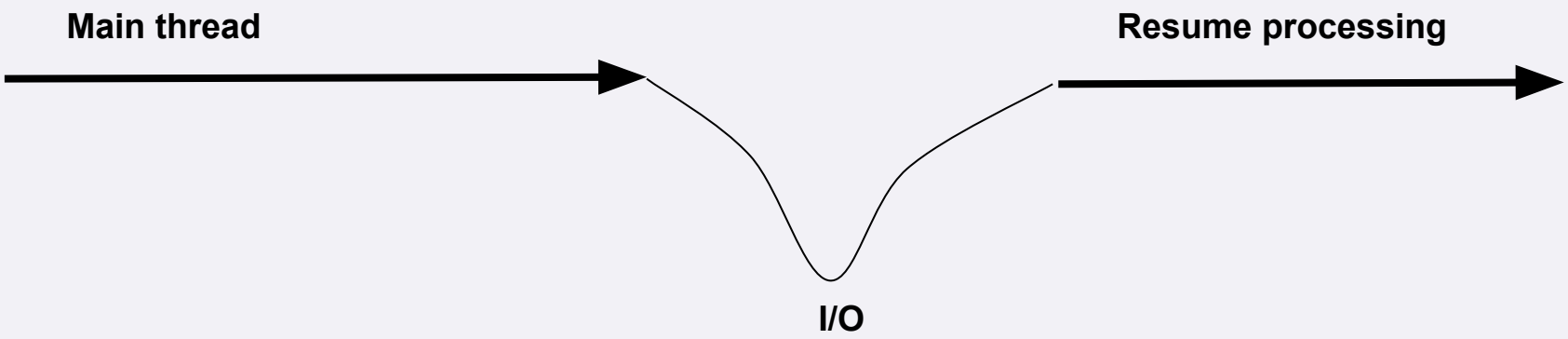
Reactive Libraries- Java Ecosystem

- Reactive Streams
- RxJava
- Reactor
- Spring Framework 5.0
- Ratpack
- Akka

Why ?

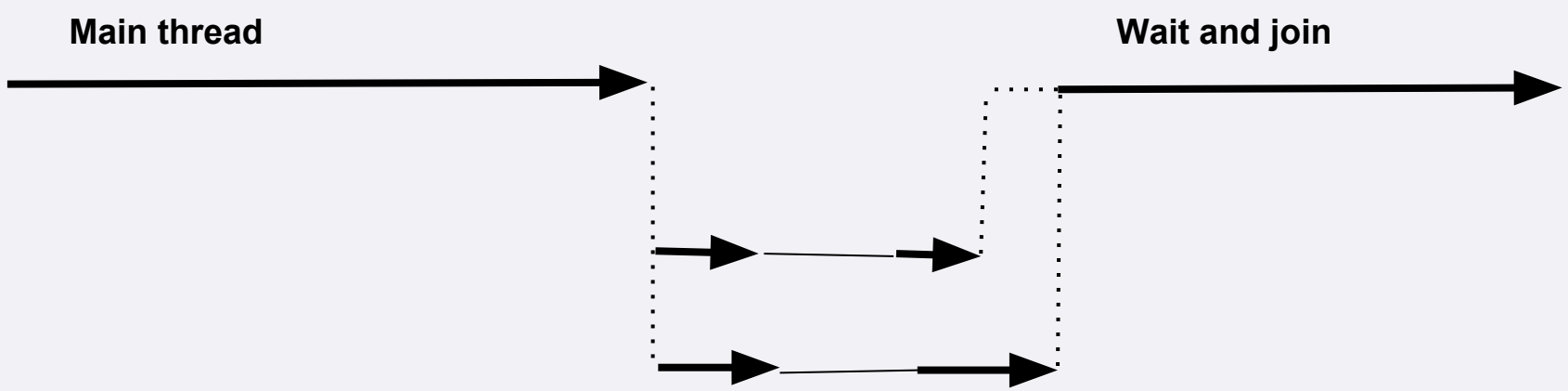
Blocking Can Be Wasteful

Synchronous and blocking



Parallelize

Asynchronous and blocking



Seeking more efficiency

- Callbacks
  - Non readable code (Callback hell)
  - Hard to compose
- Futures and CompletableFuture
  - Easy to block
  - Hard to compose
  - Lack support for multiple values
  - Lack support for error handling

```
userService.getFavorites(userId, new Callback<List<String>>() { //1
  public void onSuccess(List<String> list) { //2
    if (list.isEmpty()) { //3
      suggestionService.getSuggestions(new Callback<List<Favorite>>() {
        public void onSuccess(List<Favorite> list) { //4
          UiUtils.submitOnUiThread(() -> { //5
            list.stream()
                .limit(5)
                .forEach(uiList::show); //6
            });
        }
        public void onError(Throwable error) { //7
          UiUtils.errorPopup(error);
        }
      });
    } else {
      list.stream() //8
          .limit(5)
          .forEach(favId -> favoriteService.getDetails(favId, //9
            new Callback<Favorite>() {
              public void onSuccess(Favorite details) {
                UiUtils.submitOnUiThread(() -> uiList.show(details));
              }

              public void onError(Throwable error) {
                UiUtils.errorPopup(error);
              }
            }
          ));
    }
  }
});
```


Example of Reactor code equivalent to callback code

```
userService.getFavorites(userId)
           .flatMap(favId -> favoriteService.getDetails(favId))
           .switchIfEmpty(() -> suggestionService.getSuggestions())
           .take(5)
           .publishOn(UiUtils.uiThreadScheduler())
           .subscribe((details) -> uiList.show(details),
                      (error) -> UiUtils.errorPopup());
```


Example of Reactor code equivalent to callback code

```
userService.getFavorites(userId)
           .timeout(Duration.ofMillis(1000))
           .onErrorResume(error -> cacheService.cachedFavoritesFor(userId))
           .flatMap(favId -> favoriteService.getDetails(favId))
           .switchIfEmpty(() -> suggestionService.getSuggestions())
           .take(5)
           .publishOn(UiUtils.uiThreadScheduler())
           .subscribe((details) -> uiList.show(details),
                      (error) -> UiUtils.errorPopup());
```


Example of CompletableFuture combination

```
CompletableFuture<List<String>> ids = ifhIds();

CompletableFuture<List<String>> result = ids.thenComposeAsync(l -> {
        Stream<CompletableFuture<String>> zip =l.stream().map(i ->
         {
              CompletableFuture<String> nameTask = ifhName(i);
              CompletableFuture<Integer> statTask = ifhStat(i);

              return nameTask.thenCombineAsync(statTask, (name, stat) -> "Name " + name + " has stats
" + stat);
         });
        List<CompletableFuture<String>> combinationList = zip.collect(Collectors.toList());
        CompletableFuture<String>[] combinationArray = combinationList.toArray(new
CompletableFuture[combinationList.size()]);

        CompletableFuture<Void> allDone = CompletableFuture.allOf(combinationArray);
        return allDone.thenApply(v -> combinationList.stream()
                                                     .map(CompletableFuture::join)
                                                     .collect(Collectors.toList()));
});
List<String> results = result.join();
```


Example of Reactor code equivalent to future code

```
Flux<String> ids = ifhrIds();
Flux<String> combinations = ids.flatMap(id ->
        {
                Mono<String> nameTask = ifhrName(id);
                Mono<Integer> statTask = ifhrStat(id);

                return nameTask.zipWith(statTask, (name, stat) -> "Name " + name +
" has stats " + stat);
        });

combinations
    .collectList()
    .subscribe(strings -> strings.stream().forEach(s -> System.out.println(s)));
```

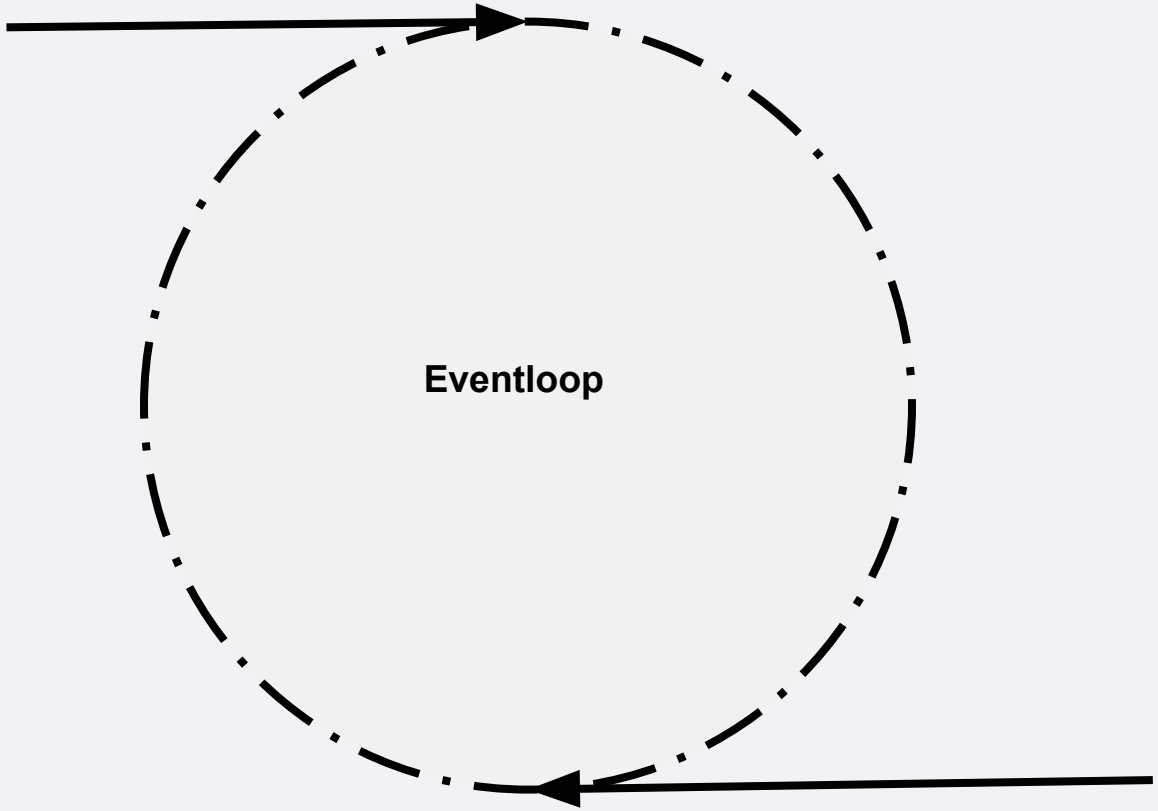

Can’t we just use Java 8 types

Type	Non-blocking	Streaming
Future<T>	No	No
CompletableFuture<T>	Yes	No
Stream<T>	No	Yes
InputStream/OutputStream	No	Yes

Imperative to Reactive programming

- Composability and readability
- Data as a flow manipulated with a rich vocabulary of operators
- Nothing happens until you subscribe
- Backpressure or the ability for the consumer to signal the producer that the rate of emission is too high
- High level but high value abstraction that is concurrency-agnostic

Asynchronous and non-blocking



Reactive is for **scalability** and **stability**, but not for speed.

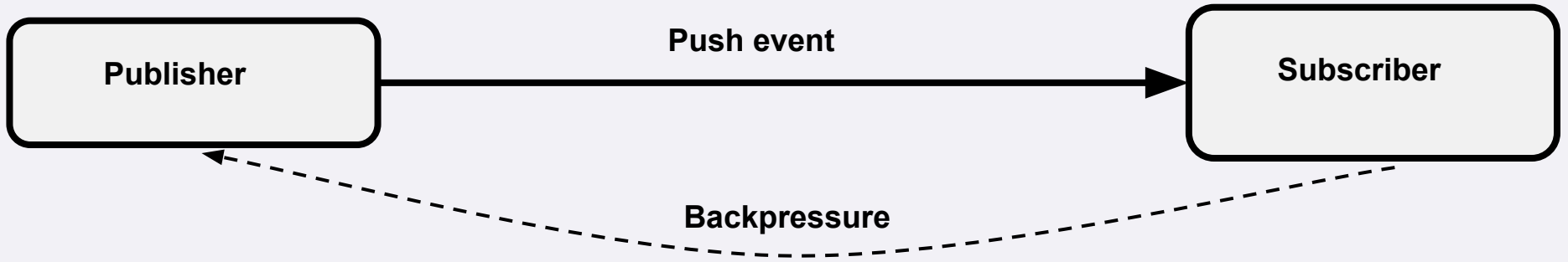
Should we convert everything to **Reactive**?

Use cases

- Remote call with latency
- Serve a lot of slow clients
- Push events to the client
- Real time analysis

Reactive Streams

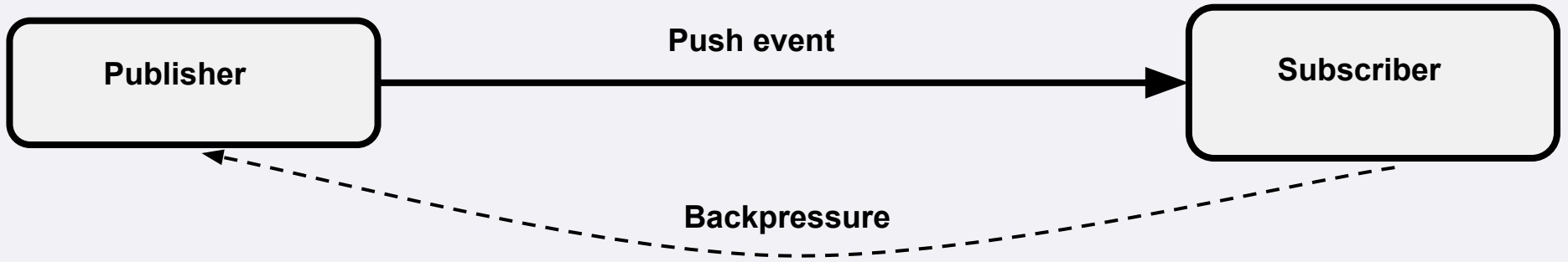
- Reactive stream is a contract for asynchronous stream processing with non-blocking back pressure handling
- De-facto standard for the behaviour of reactive libraries and for interoperability
- Co-designed by Netflix, Lightbend, Pivotal, RedHat, Kaazing, Twitter, and many others
- Implemented by RxJava, Reactor, Akka stream..



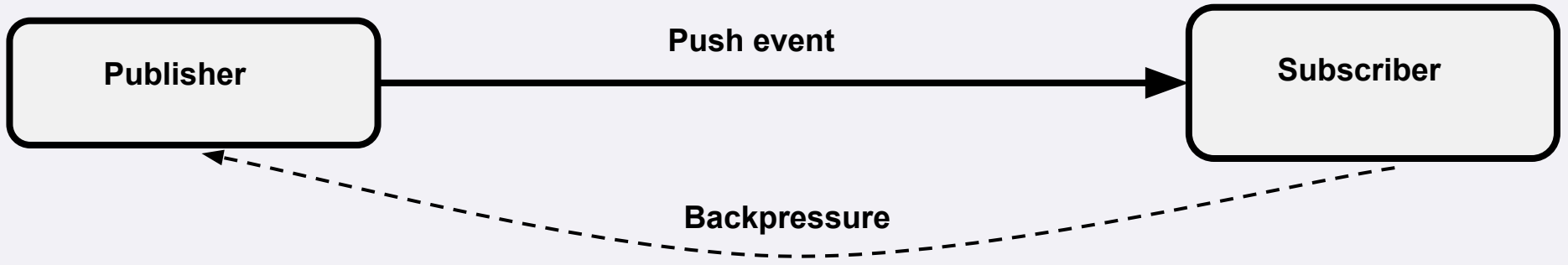

```
public interface Publisher<T>{

  void subscribe(Subscriber<? Super T> s);

}
```




```
public interface Subscriber<T>{
  void onSubscribe(Subscription s);
  void onNext(T t);
  void onError(Throwable t);
  void onComplete();
}
```




```
public interface Subscription<T>{

  void request(long n);

  void cancel();

}
```

Reactive APIs on the JVM

- RxJava
- Akka Stream
- Reactor

Reactive APIs on the JVM

Reactive API	Types for 0..n elements	Types for 0..1 element
RxJava 1	Observable	Single (1) Completable (0)
Akka Stream 2	Source Sink Flow	
Reactor Core 3	Flux	Mono (0..1)
RxJava 2	Flowable Observable	Single (1) Maybe (0..1) Completable (0)

Reactive APIs on the JVM

Reactive API	Reactive Streams Type	Non Reactive Stream Type
RxJava 1		Observable Single Completable
Akka Stream 2	Source Sink Flow	
Reactor Core 3	Flux Mono	
RxJava 2	Flowable	Observable Single Maybe Completable

Reactive APIs on the JVM

Reactive API	Generation	Support
RxJava 1	2nd	Limited back-pressure
Akka Stream 2	3rd	Reactive streams + actor fusion
Reactor Core 3	4th	Reactive streams + Operator fusion
RxJava 2	4th	Reactive streams + Operator fusion

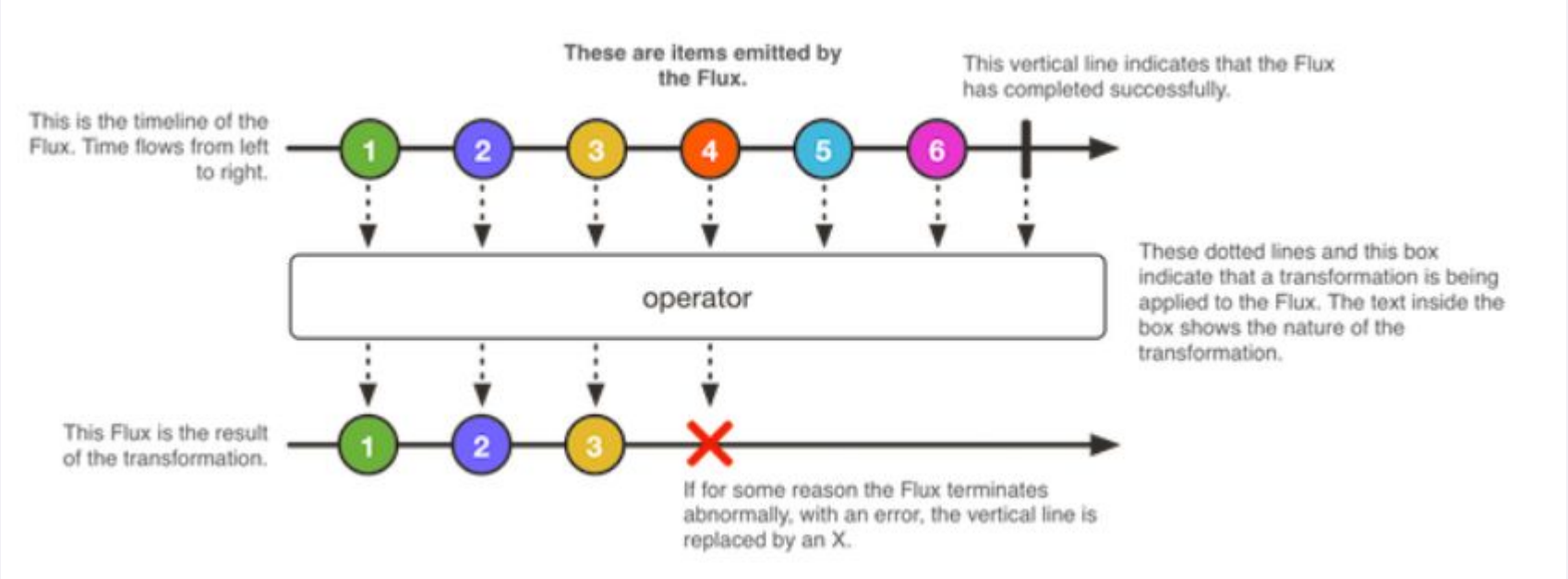
http://akarnokd.blogspot.fr/2016/03/operator-fusion-part-1.html

RxJava 2 or Reactor Core 3?

- Natively designed on top of Reactive Streams
- Lightweight API with 2 types: Mono and Flux
- Native java 8 support and optimizations
- Single 1 MB jar
- Focus on performance
- Reactive foundation of spring framework 5

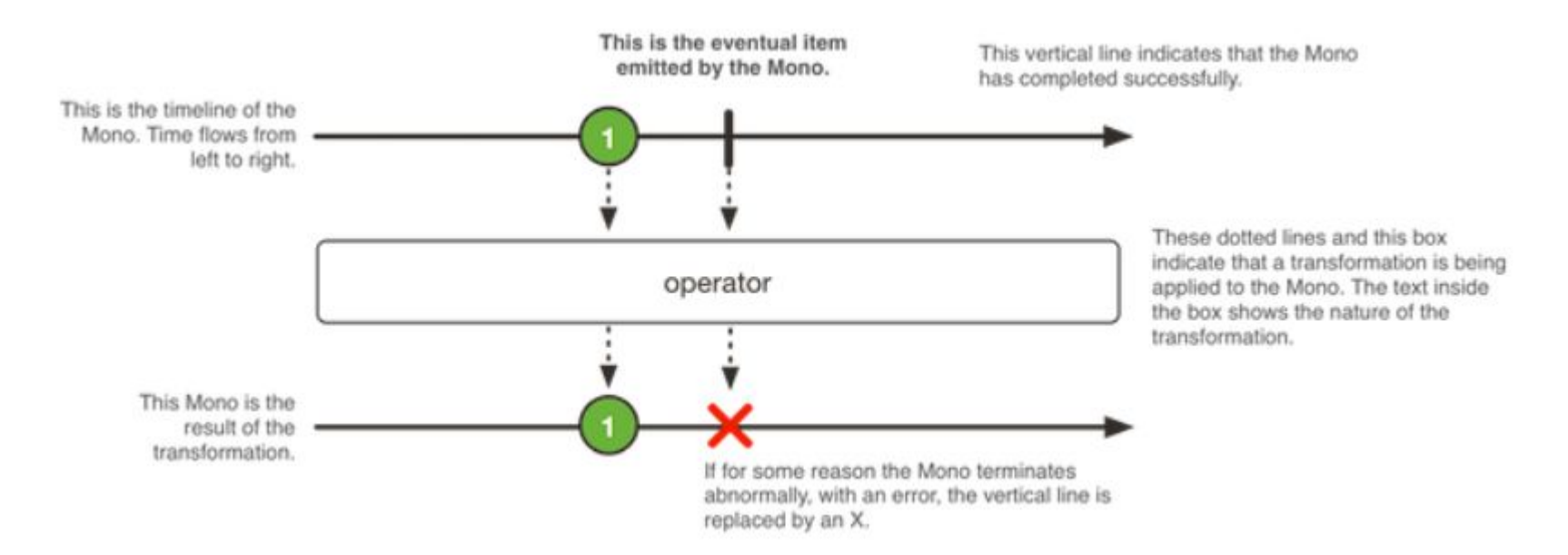
Flux<T>

- Implement reactive stream publisher
- 0 to n element
- Operators: flux.map().zip().flatMap()



Mono<T>

- Implement reactive stream publisher
- 0 to 1 element
- Operators: mono.then().otherwise()



StepVerifier

- Designed to test easily Reactive Streams publisher
- This has been carefully designed after writing thousands of Reactor and Spring reactive tests

Reactor Core 3

References and copyrights

- http://wallpaper-gallery.net/images/demo/demo-6.png
- http://www.reactivemanifesto.org/
- http://taoofdating.com/wp-content/uploads/2016/11/resilience_leafinground.jpg
- http://www.webdesigndev.com/wp-content/uploads/2015/04/Ohh-Responsive-404-Mobile-Website-Template.jpg
- http://expediafranchise.com/wp-content/uploads/2015/06/Screen-Shot-2015-06-01-at-2.52.37-PM.png
- https://image.slidesharecdn.com/reactivemachinelearningandfunctionalprogramming-150717202453-lva1-app6891/95/reactive-machine-learning-and-functional-programming-10-638.jpg?cb=1438632179
- https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape
- https://spring.io/blog/2016/06/13/notes-on-reactive-programming-part-ii-writing-some-code
- https://spring.io/blog/2016/07/20/notes-on-reactive-programming-part-iii-a-simple-http-server-application
- https://projectreactor.io/docs/core/release/reference/
- https://www.youtube.com/watch?v=Cj4foJzPF80
- https://github.com/reactor/lite-rx-api-hands-on
- https://www.youtube.com/watch?v=WJK6chc7w3o
- https://www.youtube.com/watch?v=TZUZgU6rsNY