Streams, Functional & Reactive Programming with Java & Spring WebFlux

JOHN BRYCE
Leading in IT Education
a matrix company

INTRODUCTION

Reactive programming requires understanding the following:

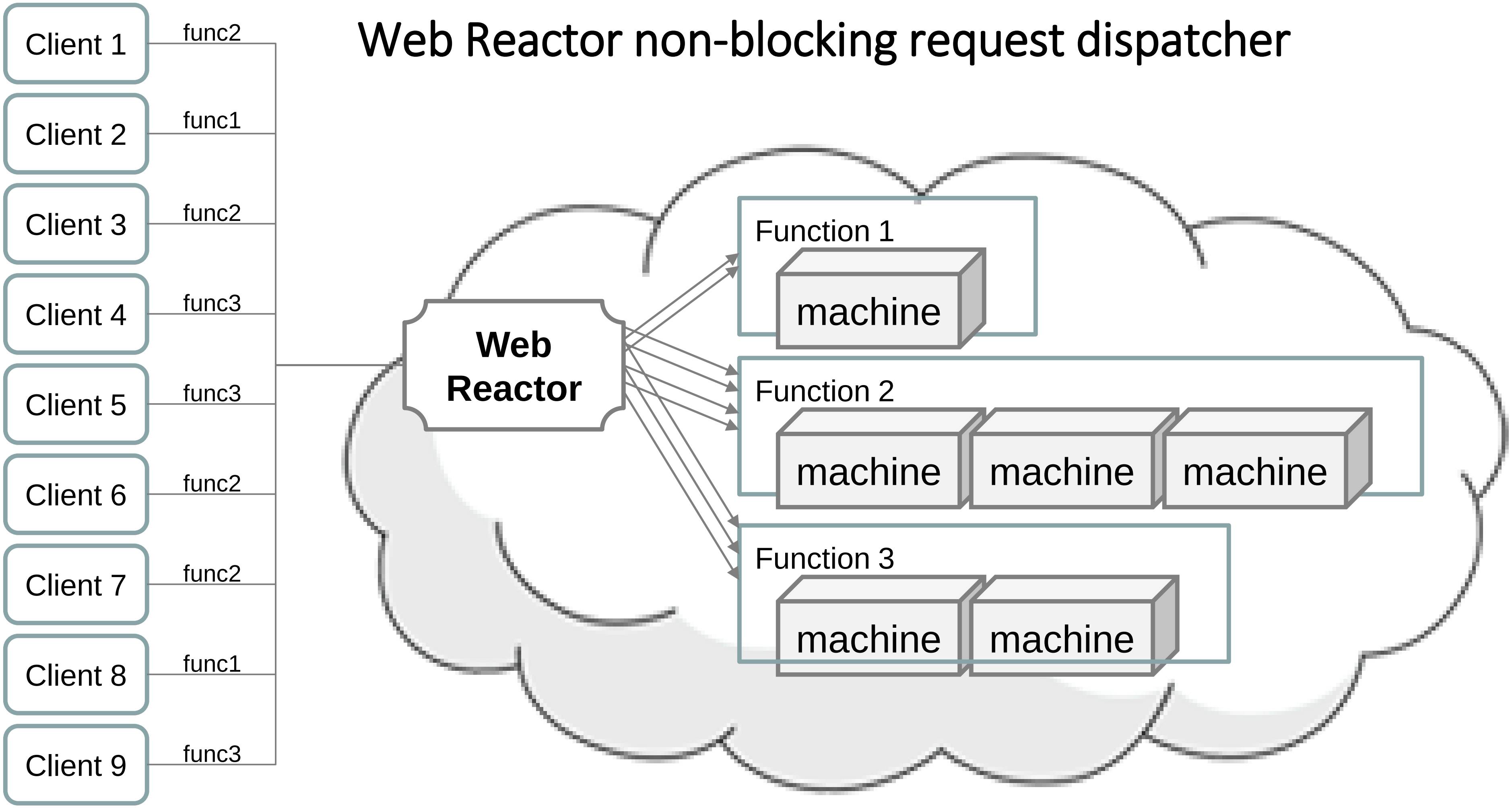
- Functional programming
- Streams
- Reactor – reactive programming
- Reactive Streams (Flow, Flux & Mono)
- Multiplexing over HTTP2

© All rights reserved to John Bryce Training LTD from Matrix group

ARCHITECTURE

JOHN BRYCE
Leading in IT Education
a matrix company

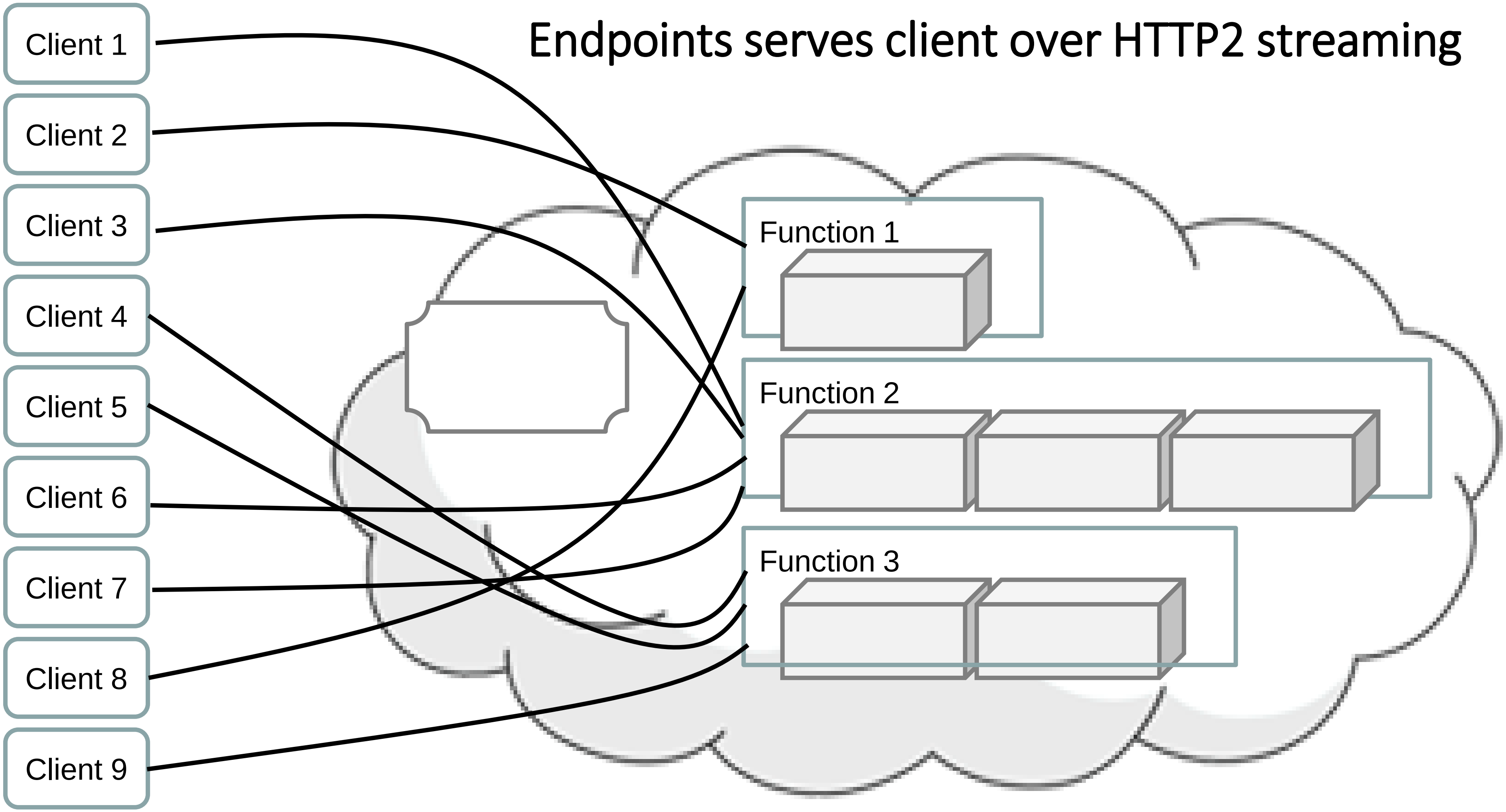
Web Reactor non-blocking request dispatcher



© All rights reserved to John Bryce Training LTD from Matrix group

ARCHITECTURE

Endpoints serves client over HTTP2 streaming



© All rights reserved to John Bryce Training LTD from Matrix group

FUNCTIONAL PROGRAMMING

Functional programming

- Java 8 offer these relevant @FunctionalInterfaces:
    - Consumer<T>
    - Supplier<T>
    - Function<T,R>
    - Predicate<T>

- Remember those when going SERVERLESS ! (if time permits)

© All rights reserved to John Bryce Training LTD from Matrix group

FUNCTIONAL PROGRAMMING

- Java 8 relevant @FunctionalInterfaces:

    - Predicate<T>.test(T) : boolean

      Accepts T and calculate a boolean result. True = passed the test

    - Consumer<T>.accept(T) : void

      Accept T and perform an operation. No result

    - Supplier<T>.get() : T

      Produces T. Therefore accepts no parameters and returns T

    - Function<T,R>.apply(T) : R

      Maps T value to R. Accepts T and calculate result R

© All rights reserved to John Bryce Training LTD from Matrix group

FUNCTIONAL PROGRAMMING

JOHN BRYCE
Leading in IT Education
a matrix company

Example:

```
Function<String, Integer> f = value -> value.length();
Consumer<String> c = System.out::println;
Supplier<Integer> s = () -> (int)(Math.random()*100);

System.out.println("Function: "+f.apply("Hello"));
System.out.print("Consumer: "); c.accept("Hello");
System.out.println("Supplier: "+s.get());

System.out.println("length(): "+length().apply("Hello"));
```

```
public static Function<String,Integer> length(){
return value -> value.length();
}
```

```
Output:

Function: 5
Consumer: Hello
Supplier: 26
length(): 5
```

© All rights reserved to John Bryce Training LTD from Matrix group

STREAMS

Streams

- New kind of iterating
- Uses internal iteration
  - Uses a pipeline
  - Pipeline is mounted with logic
    - Filter
    - Map
    - Reduce
    - Peek
  - Elements a streamed through the pipeline once triggered with a terminal operation
    - Collect
    - For-each
    - Groupby & Partitioning
    - Statistics (sum, min, max, count, average)

© All rights reserved to John Bryce Training LTD from Matrix group

Streams

- Streams are executed lazily
- Supports functional programming

```
List<String> words =
Arrays.asList("This","Is","Java","8","In","Action","Check","Out","This","Stream","API“,
              "Example");

Stream<String> longWords = words.stream().filter(s->s.length()>4);
IntStream wordsLength = words.stream().mapToInt(s->s.length());
Map<Character,List<String>> initials=words.stream()
   .collect(Collectors.groupingBy(s->new Character(s.charAt(0)),Collectors.toList()));

System.out.println("All words:");
words.stream().forEach(System.out::print);
System.out.println("\nLong words:");
longWords.forEach(System.out::print);
System.out.println("\nWord length avg:");
System.out.println(wordsLength.average().getAsDouble());
System.out.println("Grouping By Initials:");
System.out.println(initials);
```

© All rights reserved to John Bryce Training LTD from Matrix group

STREAMS

Streams

- Streams are executed lazily
- Supports functional programming

```
Output:

All words:
ThisIsJava8InActionCheckOutThisStreamAPIExample
Long words:
ActionCheckStreamExample
Word length avg:
3.9166666666666665
Grouping By Initials:
{A=[Action, API], S=[Stream], C=[Check], T=[This, This], E=[Example], 8=[8], I=[Is, In], J=[Java], O=[Out]}
```

© All rights reserved to John Bryce Training LTD from Matrix group

REACTIVE PROGRAMMING

Reactive programming

- Reactive programming is handling different parts of the request asynchronously while propagating the change
- To do that we need to:
  - Implement request handling in piplelines
  - Use a Fork-Join platforms to encapsulate parallel pipeline executions

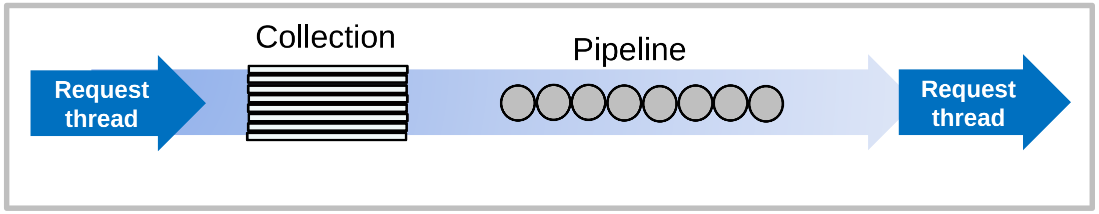
- Java 9 Flow API offers a way of splitting and forking requests

© All rights reserved to John Bryce Training LTD from Matrix group

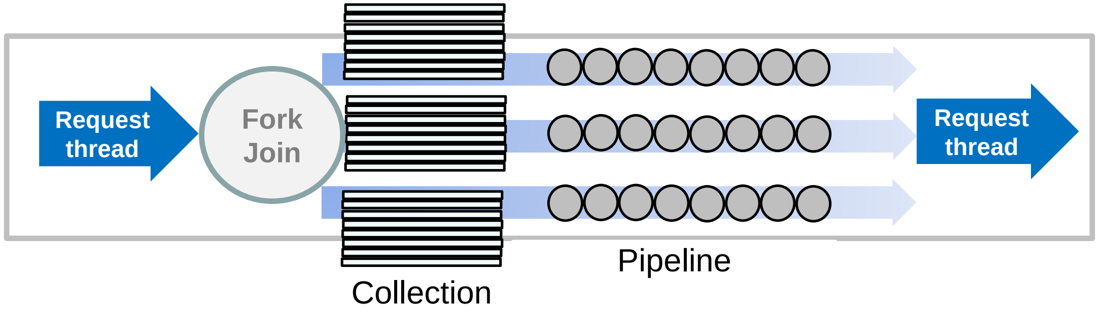
Introduction

Streams Thread Model

- Stream



- ParallelStream

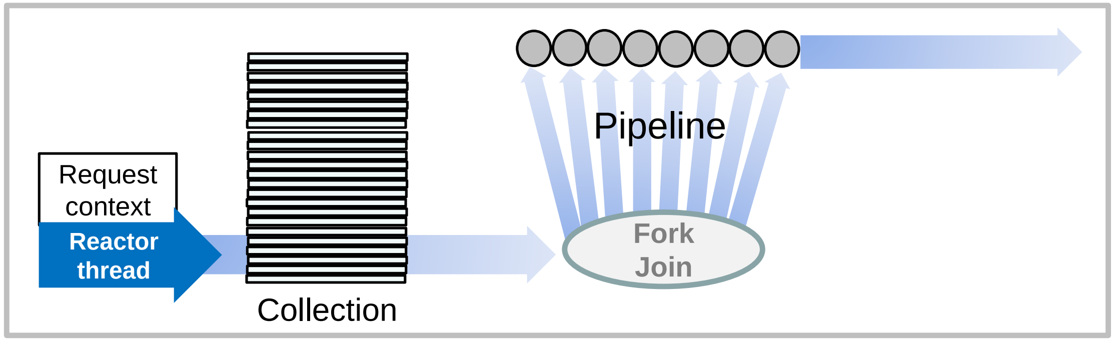


© All rights reserved to John Bryce Training LTD from Matrix group

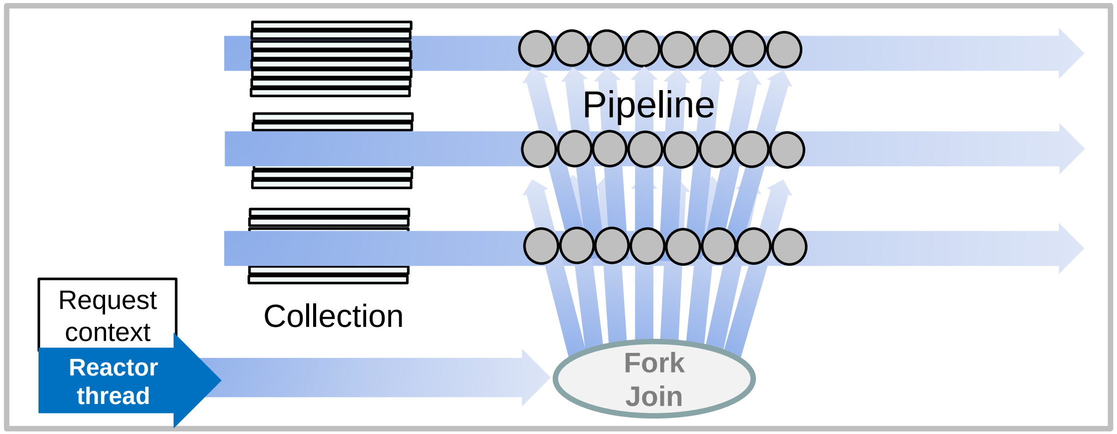
Introduction

Reactive Thread Model

- ReactiveStream



- ReactiveParallelStream



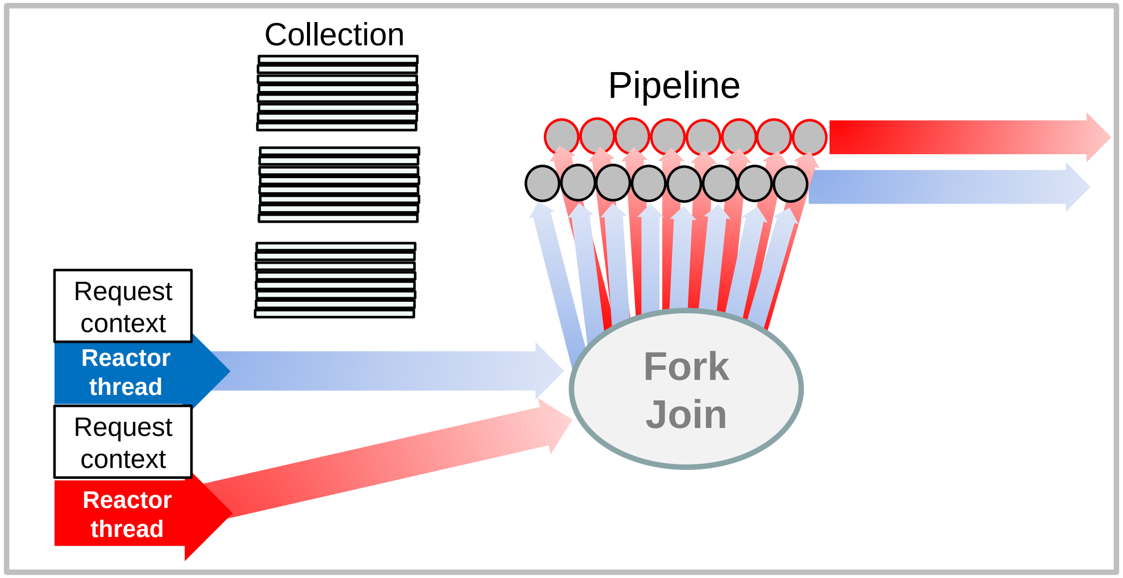
© All rights reserved to John Bryce Training LTD from Matrix group

Introduction

Reactive Thread Model

ReactiveStream

multiple requests



© All rights reserved to John Bryce Training LTD from Matrix group

REACTIVE PROGRAMMING

Reactive programming – Java9.Flow

- Flow – unit that processes events and encapsulates concurrency
- Subscriber – event endpoint
  - Subscriber events
    - onSubscribe() – after calling Publisher.subscribe(Subscriber s)
    - onNext() – notifies Subscription.request(long)
- Publisher – generates events and publishes to registered subscribers
  - Publisher supports registering subscribers for consuming data over reactive platforms
    - Subscribe(Subscriber<T> sub)
    - Extends Subscriber. Publishers are also Subscribers
- Processors – subscribing interceptors (for creating subscription chain)

© All rights reserved to John Bryce Training LTD from Matrix group

Reactive programming – Java9.Flow

```
public class MySubscriber<T> implements Subscriber<T> {

     private Subscription subscription;

       @Override
       public void onSubscribe(Subscription subscription) {
         this.subscription = subscription;
         //a value of  Long.MAX_VALUE may be considered as effectively unbounded
         subscription.request(1);
       }
       @Override
       public void onNext(T item) {
         System.out.println("Got : " + item);
         //a value of  Long.MAX_VALUE may be considered as effectively unbounded
         subscription.request(1);
       }
       @Override
       public void onError(Throwable t) {
         t.printStackTrace();
       }
       @Override
       public void onComplete() {
         System.out.println("Done");
       }
}
```

© All rights reserved to John Bryce Training LTD from Matrix group

Reactive programming – Java9.Flow

```
public class MyProcessor<T,R> extends SubmissionPublisher<R> implements Processor<T, R> {

  private Function<? super T, ? extends R> function;
  private Subscription subscription;

  public MyTransformProcessor(Function<? super T, ? extends R> function) this.function = function;}
  @Override
  public void onSubscribe(Subscription subscription) {
    this.subscription = subscription;
    subscription.request(1);
  }
  @Override
  public void onNext(T item) {
    submit((R) function.apply(item));
    subscription.request(1);
  }
  @Override
  public void onError(Throwable t) {
    t.printStackTrace();
  }
  @Override
  public void onComplete() {
    close();
  }
}
```

© All rights reserved to John Bryce Training LTD from Matrix group

Reactive programming – Java9.Flow

```
//Create Publisher
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

    //Creating Endpoint Subscriber
    MySubscriber<Integer> subscriber = new MySubscriber<>();

    //Creating Midpoints Processors
    MyProcessor<String, String> p1 = new MyProcessor<>(s -> {if(s.equals("x"))return "0"; return s;});
    MyProcessor<String, Integer> p2 = new MyProcessor<>(s -> Integer.parseInt(s));

    //Configuring subscription chain
    publisher.subscribe(p1);
    p1.subscribe(p2);
    p2.subscribe(subscriber);

    //Publish items
    System.out.println("Publishing Items...");
    String[] items = {"1", "x", "2", "x", "3", "x"};
    Arrays.asList(items).stream().forEach(publisher::submit);
    publisher.close();
```

```
Output:

Publishing Items...
Got : 1
Got : 0
Got : 2
Got : 0
Got : 3
Got : 0
Done
```

© All rights reserved to John Bryce Training LTD from Matrix group

Spring Streams

- Flux & Mono are Reactive Streams
  - Pipeline is evaluated using executors
- Flux – multi-element stream
- Mono – single element stream

© All rights reserved to John Bryce Training LTD from Matrix group

Flux & Mono

- Java8 Streams are for in-memory powerful stream manipulation
- Flux & Mono are mostly focused on how streams are consumed
- Clients may specify all types of consumer settings
  - Numbers of elements
  - Delays
  - Re-iterating
  - Pausing
  - Caching & replays
  - Joining…

© All rights reserved to John Bryce Training LTD from Matrix group

Spring WebFlux

Supported servers:

- Netty (default)
- Tomcat
- Jetty

- Tomcat & Jetty
  - are based on Servlet API
  - relevant when using both SpringMVC & WebFlux
  - WebFlux uses low-level Servlet TCP communication for streaming
  - must support Servlet 3.1 (NIO)

© All rights reserved to John Bryce Training LTD from Matrix group

Spring WebFlux

Configuration

- Maven dependency
  - Includes embedded Netty

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

- Spring property `spring.main.web-application-type` is set to ‘reactive’ by default

```
spring.main.web-application-type=reactive
```

  - Set the web engine to be reactive
  - Disabling Spring-MVC is a must!

© All rights reserved to John Bryce Training LTD from Matrix group

Flux & Mono

```
Flux.range(0,10);
Flux.range(1,10).delayElements(Duration.ofMillis(100));
Flux.range(0,10).take(endAt);
Flux.range(0,10).doOnNext(System.out::print)
```

```
List<Person> people = Lists.of(...);
Flux<Person> flux=Flux.fromIterable(people);
flux.subscribe(System.out::println);
```

```
ConnectableFlux<Integer> counter = flux. range(0,10).delayElements(Duration.ofSeconds(1)).replay(1);
counter.autoConnect().subscribe();
```

© All rights reserved to John Bryce Training LTD from Matrix group

REACTOR DP

Reactor Design Pattern

- A single treaded reactor maintains handlers in a pool
- The thread uses an Event-Loop:
  - When a request is initialized or become unblocked an event is received
  - Handler is attached to the event and becomes active
  - When request are blocked – paired handlers becomes inactive
- Active handlers are asynchronously signaled to subsystems
- When request is completed - its handler can be reused

© All rights reserved to John Bryce Training LTD from Matrix group

WEBFLUX

Spring WebFlux

- Reactive web framework
- Uses Reactor to dispatch events to IOSessions
- Uses multiplexing and non-blocking IO
- Flux endpoint – forces multiple client events
- Mono endpoint – is a single client event

© All rights reserved to John Bryce Training LTD from Matrix group

Spring WebFlux - Streaming with HTTP2

HTTP2

- **Stream MIME types:**
  - Text (JSON) Streaming - MediaType.TEXT_EVENT_STREAM_VALUE
  - Binary data –
    MediaType. APPLICATION_OCTET_STREAM_VALUE

© All rights reserved to John Bryce Training LTD from Matrix group

Spring WebFlux

Reactive
REST

```
@RestController
public class ReactiveController {

     @GetMapping(value="read/{num}", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
     public Flux<Integer> readNums(@PathVariable(“nums")int num){
       return Flux.range(0,num);
     }

     @GetMapping(value="read/{fromId}/{toId}", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
     public Flux<Person> loadPeople(@PathVariable("fromId")long from,
                    @PathVariable("toId")long to){
       List<Person> people = Lists.of(...);
       Flux<Person> flux=Flux.fromIterable(people);
       return flux.skipWhile(p->p.getId()<from).takeUntil(p->p.getId()>=to);
     }
}
```

Handling text/event-stream content with Spring WebClient

- Since WebClient supports reactive responses – it can make multiple web requests asynchronously
- Creating WebClient is done via builders pipeline
  - WebClient inner classes configures request & response
    - get(), post(), put(), delete(), head()
  - RequestSpec
    - accept(), attribute(), cookie()
    - exchange() : Mono<ClientResponse>
    - retrieve() : ResponseSpec
  - ResponseSpec
    - bodyToFlux(), bodyToMono()
    - toEntity(Class<T>) : ResponseEntity<T>
    - onStatus(Predicate<HttpStatus>)

© All rights reserved to John Bryce Training LTD from Matrix group

WEBFLUX

JOHN BRYCE
Leading in IT Education
a matrix company

WebClient Example

get() –results with RequestSpec

retrieve() – results with a ResponseSpec

bodyToFlux() – processes a stream result as Flux<T>

```
WebClient client=WebClient.create("http://localhost:8080/items");
client.get().retrieve().bodyToFlux(Item.class).subscribe(System.out::println);

//asynchronously reusing ‘client’ to make another call:
client.get().uri("http://localhost:8080/item/40").retrieve().bodyToFlux(Item.class).subscribe(System.out::println);
```

© All rights reserved to John Bryce Training LTD from Matrix group