Spring Webflux

*) What is reactive programming?

Mostly 'reactive' means a few things:

i) The architecture and design of reactive systems

ii) Reactive programming.

→ Reactive system is about the big picture. Here reactive can be thought of as a set of the same principles or an architectural style that allows an application to react to changes by being able to scale up or down or recover from failures.

↳ Using a reactive library or a reactive programming style doesn't make a whole application reactive, under this perspective.

↳ What makes a system reactive from this perspective are the traits defined in the Reactive Manifesto. It has to responsive (react quickly to all users under all conditions). Responsiveness is achieved through resilience. Resilient system takes into account all conditions (good or bad), to stay responsive. Responsiveness is also achieved because the system is scalable, i.e. resources allocated to the system can increase or decrease based on demand.

↳ Foundation of all these traits, we have a message driven architecture. Reactive systems are driven by messages to ensure that the components of the system

can be scaled independently and loosely coupled. Sometimes referred to as events, messages are not the same thing. Unlike events, messages have an explicit destination. But when events happen, there are listeners listening to those events.

→ Reactive programming is event driven, and focusses on the flow of data. Events propogate changes forming a stream of data i.e. propogated to everyone listening to those changes.

↳ like basing cell values in a spreadsheet to another cell value. Changing the dependency value, automatically changes the dependent cell value i.e. dependent cell values react to changes to changes in dependency cell values. Basically those cells are listening / subscribed to the other cells.

↳ Different from the traditional (imperivate) model of execution.

↳ Observer pattern, where a producer generates events consumed by consumer. Problem of this pattern is that if the consumer cannot keep up with all the events generated by the producer, there is no mechanism to indicate to the producer to slow down. The ability of a consumer to request items at a different rate or only when it is ready to

process them is called backpressure.

↳ Apart from backpressure, reactive programming has 3 other concepts:

(i) non-blocking programming
(ii) asynchronous programming
(iii) functional / declarative programming

*) non-blocking programming

In a threaded server, where a thread handles one request, the server doesn't wait for the first request to finish before handling another request. Another thread from the pools of threads take care of that new requests and this happens until either all requests are handled or there are no more free threads.

↓

Each thread has its own stack space in memory, so if the initial size of the thread pool is 500, and each thread needs 1 MB, then server will require 500 MB to start. If the pool grows to handle more requests, the memory consumption will increase accordingly.

↓

Here thread blocks the space/resource before completion, and if a thread is left waiting for another blocking resource, then even it gets blocked.

↓

This results in inefficient use of memory, and also

an inefficient use of CPU cycles because the program does nothing while it's waiting for other resources.

↓

Reactive web applications adopt non-blocking servers based on the event loop model. In this model, we have a small number of threads handling requests. When a request is handled by threads, it is divided into events, which are smaller pieces of work involved in handling the whole request, like parsing the request body. These events wait for their turn to be processed, but the waiting time is very small because if there's a blocking operation like calling the file system, database or any intensive computation, this operation is performed by other worker threads. When they are done, a new event is generated to signal that the operation is completed, and the result processed.

↓

This way an evented web server can handle more requests than a threaded server. Nowadays most web servers work this way, but the Servlet API can act as an issue since its a blocking API. Non blocking servlet API also available.

*) asynchronously : operation executed in parallel or in background so it can execute other things w/o waiting for the op to

finish. Synchronous is opposite. Since we wait for the the operation to be completed, we know what we are working with and accordingly write the next lines. This is not entirely possible in asynchronous programming. A popular way of handling the issue of getting the result of an op sometime in the future is using a callback. The result of the operation and a Boolean indicating if there was an error are passed as parameters of the callback.

↓

Callback hell : Nesting of 2+ callbacks can make the code repitive, hard to read & maintain.

↓

Asynchronous ops in Java : thread pools

Fork / Join Framework

ParallelStreams

CompletableFuture

CompletableFuture . supplyAsync (this :: processOperation)

. thenApply (this :: sendEmail)

. thenApply (this :: completeOperation)

↓

One issue is, if not used correctly, they can end up blocking execution in some situations.

→ implementation of server pattern in terms of a publisher and subscriber, to make the code asynchronous.

```
Publisher<Product> p = db.getProduct(id);
```

*) Functional and declarative programming

```
Publisher<ProductPrice> productPrices =
    productService.getHistoricalPrices(productId)
        .flatMap(productService::getDetails)
        .switchIfEmpty(historyService.
                getProduct(productId))
        .take(2)
        .timeout(Duration.ofMillis(2000))
        .onErrorResume(this::getDummyPrices)
        .publish(Schedulers.parallel())
        .subscribe(this::graph);
```

Sequence of events is called streams. Initiative to standardize libraries that process streams in an asynchronous and non-blocking way with backpressure capabilities is called Reactive Streams.

↓

No relationship with Java stream API. In a reactive stream, values are not pulled, they are pushed through the pipeline

of operators when we subscribe to the publishers, along with the backpressure mechanism.

*) Spring WebFlux : Spring 5.

→ doesn't replace Spring MVC. Both modules can be used to build different parts of the same application.

→ does offer an alternative programming model.

Spring MVC	Spring WebFlux
i) Servlet API	i) Reactive Streams
ii) Blocking API	ii) Non-blocking API
iii) Synchronous	iii) Asynchronous
iv) One request / thread	iv) Concurrent connections with few threads i.e. evented model

Spring-web-mvc → Servlet API → Servlet container (like Tomcat)

annotations

functional end-points

→ spring-web-reactive → HTTP / Reactive Streams → Netty (by default)

Note: All components have to be reactive to be able to take benefit of the reactive programming paradigm.

Data access : an asynchronous JDBC driver for relational database doesn't exist.

Project Reactor

*) Reactive streams : purpose to provide a contract for asynchronous, non-blocking processing and with backpressure capabilities. This contract specifies a set of interfaces to the final standard behaviour for reactive libraries. Only concerned with stream of data between the components represented by the interfaces. Doesn't specify how the data can be manipulated or transformed with operators like map or filter.

↓

Implementations are free to implement additional features as long as they conform to the 2 parts of the specification. An API that specifies the types that reactive libraries have to implement and a technology compatability kit (TCK), a test suite to see if the libraries comply with the specifications

↓

API has 4 interfaces :

I publishes → II subscribers

publishes elements to any number of

at rate | specified by ↓

IV processors

III subscription

↓

publishes and subscribers at the same time. Hard to use correctly. → Most of the time aren't needed.

I) Publisher

```
public interface Publisher<T> {

    public void subscribe(Subscriber<? super T> s);

}
```

↓

only method that subscribers use to subscribe.

↓

Publisher of type T will publish elements of type T or a subclass of T to its subscribers

II) Subscriber

```
public interface Subscriber<T> {

    public void onSubscribe(Subscription s);

    public void onNext(T t);
```

↳ always called when an element is sent to the subscriber

```
    public void onError(Throwable T);
```

↳ executed if there's an exception at some point

```
    public void onComplete();
```

↳ executed when all the elements are published.

```
}
```

iii) Subscription

```
public interface Subscription {

    public void request (long n);
        ↳ req to the publisher n no. of elements
        ↳ represents the back pressure mechanism

    public void cancel();
        ↳ to cancel the subscription

}
```

→ Working process: subscribe method is called on the publisher.

↓

subscription object is created and then the onSubscribe method of the subscriber is executed, passing the subscription object.

↓

To start receiving elements, a req. should be initiated by the subscriber to the subscription object, indicating how many elements it can process. If request method is not called explicitly, an unbounded no. of elements is requested.

↓

Only then, the subscriber can receive elements via the onNext method. Keeps receiving elements until one of three things happen.

i) publisher sends all the elements req.

↳ Subscriber can request for more elements

↳ cancel the subscription

ii) No more elements to send, onComplete called by publisher, and subscription cancelled

iii) incase of an error, publisher calls the onError method.

*) Reactor: default reactive programming library of Spring WebFlux. provides 2 implementations of the publisher interface from Reactive streams.

i) Mono to publish 0 or 1 element

ii) Flux to publish more than 1 element

↓

Two types are defined for clarity. some operations only make sense for streams for 0 or 1 element, and others for any no. of elements.

↓

i) Mono can be thought of Java's optional type

ii) Think about returning a Flux everytime we want to return a list and returning a Mono everytime we want to return a single object or for void methods.

```
subscribe()

subscribe(Consumer<? super T> consumer)

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer)

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer)

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer,
          Consumer<? super Subscription> subscriptionConsumer)
```

*) Creating a project

IntelliJ → maven → com.pluralsight : groupId

reactor-test : artefactId

↓

reactor-core → maven search repository

→ MonoTest

```
@Test
void firstMono() {

    Mono.just("A")
        .log() → handy way of looking at every
        .subscribe(); event of the sequence
```

↓

log4J

↓

Incase no logging framework is available, then it logs to the console.

@Test

```
void monoWithConsumer() {

    Mono.just("A")
        .log()
        .subscribe(s -> System.out.println(s));

}
```

↑ implementation of the consumer interface

↓ can pass more than one subscriber

@Test

```
void monoWithDoOn() {

    Mono.just("A")
        .log()
        .doOnSubscribe(subs -> System.out.println
            ("subscribed: " + subs))
        .doOnRequest(request -> System.out.println
            ("request: " + request))
        .doOnSuccess(complete -> System.out.println
            ("complete:" + complete))
        .subscribe(System.out::println);

}
```

↓ method reference instead of lambda expression

empty Mono is useful to emulate the void return in traditional programming

*) Flux : FluxTest

Flux also has a just method to create a Flux that publishes the proposed values

@Test
void firstFlux() {

    Flux.just("A", "B", "C")
        .log()
        .subscribe();

}

@Test
void fluxFromIterable() {

    Flux.just(Arrays.asList("A", "B", "C"))
        .log()
        .subscribe()

}

↗ creating a Flux from an iterable

can't be used, else the iterable will be treated as a single item.

↳ need to use fromIterable instead

↳ There is also a "from" method to create a Flux from another Publisher.

↳ can also range function

↳ interval to emit elements, starting from 0, incrementing at

a specified time interval.

```
@Test
void fluxFromInterval() throws Exception {
    Flux.interval(Duration.ofSeconds(1))
        .log()
        .subscribe();
    Thread.sleep(5000);
}
```

→ infinite stream of elements
→ will never be completed, only killed

↳ won't print anything unless the main thread is put to sleep for some time, since this method runs on another thread, and doesn't block the main thread, publishing elements at regular intervals

↓

when the test class finishes, all the threads are killed and nothing happens in this case.

↓

can be regulated using "take" method.

Subscription gets cancelled when req. elements are published.

↓

To give time to other thread to publish some values.

```
@Test
void fluxRequest() {
    Flux.range(1, 5)
        .log()
        .subscribe(null, null, null,
            s -> s.request(3)
        );
}
```

3 elements published and then killed.

}

*) Operators : map, flatMap, concat and merge, zip

OperatorTest

@Test
void map() {

can use imperative code here

```
Flux.range(1, 5)
    .map(i -> i * 10)
    .subscribe(System.out::println);
```

synchronous fn

}

@Test
void flatMap() {

transfers the element in an asynchronous way to a publisher, meaning we have to return a Flux or Mono

```
Flux.range(1, 5)
    .flatMap(i -> Flux.range(i*10, 2))
    .subscribe(system.out::println);
```

}

@Test

convert Mono to Flux

void flatMapMany() {

```
Mono.just(3)
    .flatMapMany(i -> Flux.range(1, i))
    .subscribe(System.out::println);
```

}

@Test

void concatenate() {

delay the publishing of an element by the given duration.

Flux<Integer> oneToFive = Flux.range(1, 5)

.delayElements(Duration.ofMillis(2000));

Flux<Integer> sixToTen = Flux.range(6, 5)

.delayElements(Duration.ofMillis(4000));

Flux.concat(oneToFive, sixToTen) → can concat any no. of publishers

.subscribe(System.out::println);

// oneToFive.concatWith(sixToTen)

.subscribe(system.out::println);

Thread.sleep(4000);

}

→ merge doesn't combine the publishers in a sequential manner. Interleaves the value

→ Another method to combine publishers : zip

```
@Test
void zip()  {
    Flux<Integer> oneToFive = Flux.range( start: 1,  count: 5);
    Flux<Integer> sixToTen = Flux.range( start: 6,  count: 5);

    Flux.zip(oneToFive, sixToTen,
            (item1, item2) -> item1 + ", " + + item2)
            .subscribe(System.out::println);

     oneToFive.zipWith(sixToTen)
             .subscribe(System.out::println);
}
```

→ two helpful documentation :

i) which operator do I need ?

ii) FAQ, Best practices, and "How do I ... ?"

→ like Rx API hands on

Building a REST API with annotated controllers

*) Annotated Controllers : annotation model is practically the same as compared to Spring MVC.

two annotations for our controllers :

i) Controller → model and build rendering

ii) RESTController → combination of Controller and ResponseBody annotations to indicate that every method of controller writes to the response body directly.

→ RequestMapping, same as that for MVC, can also use GetMapping, PostMapping, PutMapping, DeleteMapping and PatchMapping

→ The biggest difference b/w MVC & WebFlux is how the request and response are handled, and support of reactive types.

↓

In WebFlux, we have a ServerWebExchange class that acts as a container for the HTTP request and response, request and session attributes, and other related properties and features.

↳ ServerHttpRequest

↳ ServerHttpResponse

→ Actually, as arguments for the methods of our controllers, WebFlux supports all the types supported by MVC. But some of them can be treated as non-blocking & reactive types like Flux, or Mono can be used.

↓

WebSession

→ arguments with reactive support

Java.security.Principal

↳ represents the current authenticated user

@RequestBody

↳ to access the HttpRequestBody

HttpEntity

↳ for accessing request headers and body

@RequestPart

↳ to access a part in a multi-part form data request

→ Reactive types supported for all return values i.e. we can return a Mono of ResponseEntity to specify the full response including HTTP headers and body. can return a Mono or a Flux of any object or Flux or ServerSentEvent object to emit server-side events.

String

- @RestController : Mono<String>

→ Spring Initializr

groupId : com.wiredbraincoffee

artefactId : product-api-annotation

dependencies : reactive web

reactive mongodb

Spring-data-mongodb module

mongodb reactive driver

embedded mongodb database

On running, there will be an error: MongoSocketOpenException

↓

Since no mongodb server is running.

↓

Remove the embedded database from test scope

→ To create the repository class, we need an entity object to represent the product.

new pkg : model → Product class

mongodb : entities are not tables like in a relational database, rather a collection of JSON documents.

need to annotate our entity classes with Document annotation

@Document
public class Product {

    @Id
    private String id;

    private String name;

    private Double price;

→ default and parameterized constructor
→ getters and setters
→ equals and hashcode methods using all fields
→ override toString() method

new pkg: Repository → ProductRepository → Interface

public interface ProductRepository

extends ReactiveMongoRepository<Product, String> {

↓

will work with Product object

→ want to execute some code right after SpringBoot applcn. finishes booting: SpringBoot provides 2 interfaces for this purpose, CommandLineRunner and ApplicationRunner, difference being how they get access to application arguments.

↓

we want to insert some initial products to our embedded database, so either will work

↓

CommandLineRunner

↳

main class of the Application, annotated with SpringBootApplication.

↓

Init Bean

↓

@Bean → so that Spring can process CommandLineRunner automatically

```
CommandLineRunner init (ProductRepository
                        repository) {
    return args -> {
        Flux<Product>
            productFlux =
                Flux.just (Product 1,
                           Product 2,
                           Product 3);
        .flatMap (p -> repository.save (p))
        productFlux
            .thenMany (repository.findAll())
            .subscribe (System.out::println);
    }
```

↳ injected automatically by Spring

functional interface that receives a variable number of String arguments

embedded database not persisted b/w executions

waits for the publisher to finish, and then execute the publisher it receives as argument

*) Building the Controller

new pkg : controller → ProductController class

```
@RestController
@RequestMapping("/products")
public class ProductController {

    private ProductRepository repository;

    public ProductController(ProductRepository repository) {
        this.repository = repository;
    }
```

↳ constructor injection

```
    @GetMapping
    public Flux<Product> getAllProducts() {
        return repository.findAll();
    }
```

↓

Spring will call subscribe at the right time

```
    @GetMapping("{id}")
    public Mono<ResponseEntity<Product>> getProduct(@PathVariable
                                                    String id) {
        return repository.findById(id)
            .map(product -> ResponseEntity.
                ok(product))
            .defaultIfEmpty(ResponseEntity.
                notFound().build());
    }
```

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

public Mono<Product> saveProduct(@RequestBody Product product){

return repository.save(product);

}

```
@PutMapping("{id}")
public Mono<ResponseEntity<Product>> updateProduct(@PathVariable(value = "id") String id,
                                                   @RequestBody Product product) {
    return repository.findById(id)
            .flatMap(existingProduct -> {
                existingProduct.setName(product.getName());
                existingProduct.setPrice(product.getPrice());
                return repository.save(existingProduct);
            })
            .map(updateProduct -> ResponseEntity.ok(updateProduct))
            .defaultIfEmpty(ResponseEntity.notFound().build());
}
```

```
@DeleteMapping("{id}")
public Mono<ResponseEntity<Void>> deleteProduct(@PathVariable(value = "id") String id) {
    return repository.findById(id)
            .flatMap(existingProduct ->
                    repository.delete(existingProduct)
                            .then(Mono.just(ResponseEntity.ok().<Void>build()))
            )
            .defaultIfEmpty(ResponseEntity.notFound().build());
}
```

→ test using postman

*) Server-side events

Model → ProductEvent class

eventId

eventType

```
@GetMapping(value = "/events", produces = MediaType.
                                    TEXT_EVENT_STREAM_VALUE)
public Flux<ProductEvent> getProductEvents() {
    return Flux.interval(Duration.ofSeconds(10))
        .map(val ->
            new ProductEvent(val,
                "ProductEvent")
        );
```

Since we mentioned that we are producing a stream, we did not have to return Flux of type ServerSideEvent