Spring Reactive Notes

1) Synchronus based execution (Blocking Thread)

2) Asynchronus based execution (Non Blocking Thread)

=> Spring 5.x introduced Reactive Programming

=> In Spring 5.x 'starter-webflux' introduced

==========

Old Approach

==========

```
@RestController
public class WelcomeRestController{

   @GetMapping("/msg")
          public String getMsg(){
                return "Hello";
          }

}
```

==========

New Approach

==========

```
@Component
public class MessageRequestHandler{

        public Mono<ServerResponse> handle(ServletRequest request){
                return new ServerRespoinse.ok()

.contentType(MediaType.APPLICATION_JSON)

.body(BodyInserters.fromValue(data));
        }

}

@Configuration
public MsgRouter {

                @Bean
                pubilc RouterFunction<ServerResponse> route(MessageRequestHandler
requestHandler){

                                return RouterFunctions.route(GET("/hello"))

.and(accept(MediaType.APPLICATION_JSON), MessageRequestHandler::handle);

                }

}
```

==========

Perks of Using Spring WebFlux:

==========

1.Asynchronous and non blocking --> Be can handle more request(using pub-sub model)

2.Funtional Style coding --> Use more lambdas

3.Data flow as event driven stream -->Data is transfer as event driven ---whenever changes happen pubhlisher publish the event

4.Backpressure on data streams --> Add limitation on data from database

==========

Reactive programming is a programming paradigm where the focus is on developing =

asynchronous and non-blocking applications in an event-driven form. =

==========

==========

Specification of Reactive Stream Programming

==========

Reactive programming has some rules these rules is known as specification

There are 4 main interfaces:

1) Publisher

2) Subscriber

3) Subscription

4) Processor

==========

1) Publisher - Acts as DataSource

==========

Methods :

```
public interface Publisher<T>{
    public void subscribe(Subscriber<? super T> s);
}
```

==========

2) Subscriber - Acts as Data Receiver

==========

Methods :

```
public interface Subscriber<T> {

    public void onSubscribe(Subscription s);

    public void onNext(T t);

    public void onError(Throwable t);

    public void onComplete();

}
```

==========

3) Subscription - Request Data from Publisher or cancel a request

==========

Methods:

```
public interface Subscription{

    public void request(long n);
    public void cancel();

}
```

==========

4) Processor - Processor interface is the combination of both Publisher and Subscriber interface

==========

Methods:

```
public interface Processor<T,R> extends Subscriber<T>,Publisher<R>{

}
```

==========

Libraries for reactive programming :

==========

1. Reactor
2. RxJava
3. Jdk9 Flow Reactor Stream

==========

Project Reactor :

==========

Project reactor is library that implements reactive specification

for building non-blocking and asynchronous applications on JVM.

****

Implementation of Publisher:

****

1) Flux - 0...N elements

2) Mono - return 0 or 1 element

=> Creating Mono and Flux

```
Mongo<String> mono=Mono.just("data")
```

=> Error

```
Mono.just("data").then(Mono.error(new RuntimeException("ERROR")))
```

Important Mono Opeartors:

---

1) zip() & withZip() -> merge mono provide Mono of Tuple / Flux

Eg:

Mono<Tuple3<String,String,Integer>> combinedMono = Mono.zip(m1,m2,m3);

Mono<Tuple2<String,String>> zipWithMono = m1.zipWith(m2);

zip and zipWith can indeed return either Mono<TupleN> or Flux<> depending on the type of publishers they combine

* When to use zip and zipWith:

zip(): Imagine two data streams with the same flow rate.

zip combines elements from both streams at the same time point, creating a new stream with paired elements.

If one stream runs out, the combined stream stops.

zipWith(): Imagine two data streams with potentially different flow rates.

withZip allows you to define a custom logic (zipper function) that processes elements from both streams,

even if one finishes earlier. It can handle missing elements with defaults or null values.

Important tip :

1) Use zip when you want a simple pairing of elements with sources emitting the same number of elements.

2) Use withZip when you need to perform custom logic on combined elements or handle sources of different lengths.

1) Using zip():

```
// Another way of doing it
Mono<Tuple3<String,String,Integer>> combinedMono = Mono.zip(m1,m2,m3);

Flux<User> users = Flux.just(new User("Alice"), new User("Bob"));
Flux<Order> orders = Flux.just(new Order("Laptop"), new Order("Headphones"));

// Only works if both have same number of elements
Flux<String> combined = Flux.zip(users, orders, (user, order) -> user.getName() + " ordered " + order.getProductName());

combined.subscribe(System.out::println); // Output: Alice ordered Laptop, Bob ordered Headphones
```

2) Using withZip():

```
// ANother way of doing it
Mono<Tuple2<String,String>> zipWithMono = m1.zipWith(m2);

Flux<String> combined = users.withZip(orders)
                                                .<String> then((user, order) ->
                                                        order != null ? user.getName() + " ordered " + order.getProductName()
```

: user.getName() + " (no order)");

combined.subscribe(System.out::println);

==========

Key takeaway:

==========

1. zip is for simple pairing with equal-length sources.

2. withZip offers more control and flexibility for combining elements and handling source length differences

2) map() --> transform the value emmited by current using syn function (used For transformation of data in Publisher)

Purpose:-> Applies a synchronous transformation function to each element in a Mono or Flux.

Transformation:-> The function you provide takes an element as input and returns a new element of the same type.

Output:-> A new Mono or Flux containing the transformed elements.

Eg1 :

```
Mono<String> m1 = Mono.just("Siddhesh is new joinee");

Function<String,String> f = str -> str.toUpperCase();

Mono<String> resultMapMono = m1.map(data -> f.apply(data));

Mono<String> resultMapMono = m1.map(value -> value.toUpperCase());
```

```
// You can also used Method reference here
Mono<String> resultMapMono = m1.map(String::toUpperCase);

resultMapMono.subscribe(System.out :: println);
```

Eg2 :

```
Flux<String> names = Flux.just("Alice", "Bob", "Charlie");
Flux<String> greetings = names.map(name -> "Hello, " + name + "!");
```

3) flatMap() -> transform the value emmited by current mono async,

returning the value emmited by another mono (change value type possible)

Purpose:-> Asynchronously transforms each element in a Mono or Flux into a new Publisher (either Mono or Flux).

Transformation:-> The function you provide takes an element as input and returns a Publisher that emits the transformed elements.

Output:-> A flattened Mono or Flux containing the elements emitted by all the returned Publishers.

Eg1 :

```
Mono<String[]> resultFlatMapMono = m1.flatMap(valueM1 -> Mono.just(valueM1.split(" ")));

resultFlatMapMono.subscribe(data -> {
    for (String str : data) {
        System.out.print(str + ",");
```

```
                }
            });
```

Eg2 :

```
        Mono<User> userMono = Mono.just(userId);
        Mono<UserDetails> detailsMono = userMono.flatMap(id -> userService.getUserDetails(id));
```

4) flatMapMany() -> transform this mono into publisher , then change to return a many value that is Flux.

Purpose:-> Similar to flatMap(), but specifically designed to transform elements into Flux instances.

Transformation:-> The function you provide takes an element as input and returns a Flux that emits the transformed elements.

Output:-> A flattened Flux containing the elements emitted by all the returned Flux instances.

Eg1:

```
        Flux<String> stringFlux = m1.flatMapMany(value -> Flux.just(value.split(" "))).log();

            stringFlux.subscribe(data ->
                    System.out.println(data)
            );

        Flux<Product> productsFlux = Flux.just(productId1, productId2);
        Flux<Review> reviewsFlux = productsFlux.flatMapMany(id -> productService.getReviews(id));
```

Choosing the Right Operator:

* Use map() for simple transformations without asynchronous operations.

** Use flatMap() for transformations that involve asynchronous calls or where the transformation result is a Publisher.

*** Use flatMapMany() specifically when the transformation result is always a Flux.

5) concatWith() -> join to mono and provide Flux

Eg: Flux<String> secondString = m1.concatWith(Mono.just("second string"));

6) deleyElement() to provide delay on onNext()

Eg: Flux<String> concatFLux = m1.concatWith(m2).log().delayElements(Duration.ofMillis(2000));

Important Flux Opeartors:

---

1) map

=> Flux<String> capFlux = getFlux().map(name->name.toUpperCase());

2) filter

=> return getFlux().filter(name->name.length()>4);

3) flatMap(flatMapMany(mono to flux)) mapping convert each element to flux of element.(Asynchronous)

=> return getFlux().flatMap(name->Flux.just(name.split("")));

=> return getFlux().flatMap(name -> Flux.just("Test Flux")).delayElements(Duration.ofSeconds(2)).log();

=> return getFlux().flatMap(name -> Flux.just(name.split(""))).delayElements(Duration.ofSeconds(2)).log();

=> return Mono.just("Siddhesh").flatMapMany(value->Flux.just(value.split("")));

4) concatMap (preserve the order)

=> similar to concat/concatWith but we can apply function or apply transformation

=>

```
Flux<String> source = Flux.just("apple", "banana", "orange");

Flux<String> transformed = source.concatMap(fruit -> Flux.just(fruit + " juice", fruit + " pie"));

transformed.subscribe(System.out::println); // Output: apple juice, apple pie, banana juice, banana pie, orange juice, orange pie
```

5) delayElements- delay the elements

=> Flux.just("Test Flux")).delayElements(Duration.ofSeconds(2))

6) transform -- take functional interface to transform the data

=>

```
Function<Flux<String>,Flux<String>> funInterface = (name) -> name.map(String :: toUpperCase);

return getFlux().transform(funInterface).log();
```

7) defaultIfEmpty--> provide the default data if empty

defaultIfEmpty(pass the default value)

```
=>  return getFlux().filter(name -> name.length() > length)
                        .defaultIfEmpty("Not Found")
                        .log();
```

9) switchIfEmpty- switch to new set of data switchIfEmpty(passflux)

```
=>  return getFlux().filter(name -> name.length() > length)
                        .defaultIfEmpty("Not Found")
                        .switchIfEmpty(fruitsFlux())
                        .log();
```

11)concat(static) & concatWith(instance method) (sync) --> combines streams with order is preserved

```
=>
Flux.concat(getFlux().delayElements(Duration.ofSeconds(1)),fruitsFlux().delayElements(Duration.ofSeconds(2)));
```

```
=>  getFlux().concatWith(fruitsFlux());
```

12)merge & and mergeWith (async) --> (order is not preserved)

```
=>
Flux.merge(getFlux().delayElements(Duration.ofSeconds(1)),fruitsFlux().delayElements(Duration.ofSeconds(2)));
```

=>
getFlux().delayElements(Duration.ofSeconds(1)).mergeWith(fruitsFlux().delayElements(Duration.ofSeconds(2)));

13)mergeSequential --> merges elements from multiple Publisher instances (typically Flux)
into a single Flux while maintaining the order of elements within each source publisher.

```
=>  Flux<Integer> flux1 = Flux.just(1, 3, 5);
        Flux<Integer> flux2 = Flux.just(2, 4, 6);

        Flux<Integer> merged = Flux.mergeSequential(flux1, flux2);

        merged.subscribe(System.out::println); // Output: 1, 2, 3, 4, 5, 6
```

##working with different type reactive type

14)zip & zipWith

```
=>  Flux.zip(getFlux(),Flux.just(1,2,3));

=>  Flux.zip(getFlux(),Flux.just(1,2,3,4),(first,second)->{
                return first +" : "+ second;
            });
```

##SideEffect operators (dont change the actual behaviour)

15) doOnNext()

=> when onNext is called

16) doOnSuscribe()

=> when onSubscribe is called

17) doOnEach()

=> called for every request

18) doOnComplete()

=> called for onComplete() is called

Eg:

```
	return getFlux().doOnNext(data -> {
				System.out.println(data + " on Next");
		}).doOnSubscribe(data->{
				System.out.println(data + "on subscribe");
		}).doOnEach(data->{
				System.out.println(data + "on each");
		}).doOnComplete(() ->{
				System.out.println("completed");
		});
```

Code Example
##
#####

==========

SprinBoot Reactive Example

==========

1) Create Boot application with 'Reactive Web' dependency

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

Note: Reactive Web dependency means 'starter-webflux' dependency. It will provide 'Netty' as default embedded container.

2) Create Binding class to response

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Greeting {

    private String msg;

}
```

3) Create Request Handler class like below

```
@Component
public class GreetingHandler {
```

```
	public Mono<ServerResponse> hello(ServerRequest request){

		return ServerResponse.ok()
					.contentType(MediaType.APPLICATION_JSON)
					.body(BodyInserters.fromValue(new Greeting("Hello
World")));
	}
}
```

4) Create Router class

```
import static org.springframework.web.reactive.function.server.RequestPredicates.GET;
import static org.springframework.web.reactive.function.server.RequestPredicates.accept;

@Configuration
public class GreetingRouter {

	@Bean
	public RouterFunction<ServerResponse> route(GreetingHandler greeting){
		return RouterFunctions
			.route(GET("/hello")
			.and(accept(MediaType.APPLICATION_JSON)), greeting::hello);
	}
}
```

5) Run the application and test it.

==========

Book Rest API(By Reactive Programming)

==========

==========

1) Add Below Dependencies in Pom.xml

==========

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

		<!-- https://mvnrepository.com/artifact/org.springframework.data/spring-data-r2dbc -->
		<dependency>
		  <groupId>org.springframework.data</groupId>
		  <artifactId>spring-data-r2dbc</artifactId>
		</dependency>
		<dependency>
		  <groupId>com.mysql</groupId>
		  <artifactId>mysql-connector-j</artifactId>
		</dependency>
		<dependency>
		  <groupId>org.springframework.boot</groupId>
		  <artifactId>spring-boot-starter-data-r2dbc</artifactId>
```

```
		</dependency>

		<dependency>
			<groupId>io.asyncer</groupId>
			<artifactId>r2dbc-mysql</artifactId>
			<scope>runtime</scope>
		</dependency>
```

==========

2)Create Entity class called Book which matches with Table in database

==========

```
@Data
@Table("book_details")
@NoArgsConstructor
@AllArgsConstructor
public class Book {

	@Id
	private int bookId;

	private String name;

	@Column("book_desc")
	private String description;

	private String publisher;

	private String author;
```

}

==========

3) Add mysql r2dbc connection details in .properties file

==========

spring.r2dbc.url=r2dbc:mysql://localhost:3306/reactivedb

spring.r2dbc.username=root

spring.r2dbc.password=password

==========

4) Create BookRepository interface which extends ReactiveCrudRepository for CRUD operations on database

(It is similar to CrudRepository in data-jpa but instead of sending Entity, it sends Mono<Entity> /Flux<Entity>)

==========

```
@Repository
public interface BookRepository extends ReactiveCrudRepository<Book,Integer> {

    public Mono<Book> findByName(String name);

    public Flux<Book> findByAuthor(String author);

    public Flux<Book> findByPublisher(String publisher);

    public Flux<Book> findByNameAndAuthor(String name,String author);
```

```
	@Query("select * from book_details where author = :auth")
	public Flux<Book> getAllBooksByAuthor(@Param("auth") String author);

	@Query("select * from book_details where name LIKE :title")
	public Flux<Book> searchBookByTitle(String title);
}
```

==========

5) Create BookService interface with all required services method
which can be used by RestController

==========

```
public interface BookService {

	public Mono<Book> create(Book book);

	public Flux<Book> getAll();

	public Mono<Book> get(int bookId);

	public Mono<Book> update(Book book, int bookId);

	public Mono<Void> delete(int bookId);

	public Flux<Book> search(String query);

	public Flux<Book> searchBook(String titleKeyword);
```

```
	// add more methods as your requirements
}
```

==========

6) Create BookServiceImpl class implemneting BookService interface

==========

```
@Service
public class BookServiceImpl implements BookService{

	@Autowired
	private BookRepository bookRepo;

	@Override
	public Mono<Book> create(Book book) {
		System.out.println(Thread.currentThread().getName());
		Mono<Book> createdBook = bookRepo.save(book).doOnNext(data->{
			System.out.println(Thread.currentThread().getName());
		});

		return createdBook;

	}

	@Override
	public Flux<Book> getAll() {
		return bookRepo.findAll()
					.delayElements(Duration.ofSeconds(2))
					.log()
```

```
                                        .map(book->{

book.setName(book.getName().toUpperCase());
                                                                return book;
                                        });

}

@Override
public Mono<Book> get(int bookId) {
        Mono<Book> item = bookRepo.findById(bookId)
                                                                .map(book->{

book.setName(book.getName().toUpperCase());
                                                                        return book;
                                                                });
        return item;
}

@Override
public Mono<Book> update(Book updatedBook, int bookId) {

        Mono<Book> existingBook = bookRepo.findById(bookId);
        existingBook.subscribe(System.out::println);

        return existingBook.flatMap(retrievedBook -> {
                retrievedBook.setName(updatedBook.getName());
                retrievedBook.setDescription(updatedBook.getDescription());
                retrievedBook.setPublisher(updatedBook.getPublisher());
```

```
                retrievedBook.setAuthor(updatedBook.getAuthor());
                return bookRepo.save(retrievedBook).log(); // this return is part of Function Body within flatMap that return Mono<Book> object
        });



    }

    @Override
    public Mono<Void> delete(int bookId) {

        return bookRepo.findById(bookId)
                                .flatMap(book ->bookRepo.delete(book));

    }

    @Override
    public Flux<Book> search(String query) {

        return null;
    }

    @Override
    public Flux<Book> searchBook(String titleKeyword) {
        return this.bookRepo.searchBookByTitle("%"+titleKeyword+"%");
    }
}
```

==========

7) Create BookController with all URL endpoints

==========

```
package in.api.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import in.api.entities.Book;
import in.api.services.BookService;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@RestController
@RequestMapping("/books")
public class BookController {

	@Autowired
	private BookService bookService;

	//create
	@PostMapping
	public Mono<Book> create(@RequestBody Book book){
		return bookService.create(book);
	}

	//get all books
```

```
@GetMapping
public Flux<Book> getAll(){
        return bookService.getAll();
}

//get single book
@GetMapping("/{bid}")
public Mono<Book> get(@PathVariable("bid") int bookId){
        return bookService.get(bookId);
}

//update
@PutMapping("/{bid}")
public Mono<Book> update(@RequestBody Book book, @PathVariable("bid") int bookId){
        return bookService.update(book,bookId);
}

//delete
@DeleteMapping("/{bid}")
public Mono<Void> delete(@PathVariable("bid") int bookId){
        return bookService.delete(bookId);
}

//search
@GetMapping("/search")
public Flux<Book> searchBooks(@RequestParam("query") String query){
        System.out.println(query);
        return this.bookService.searchBook(query);
}
```

```
        @ExceptionHandler(Exception.class) // Can be more specific for relevant exceptions
    public Mono<ResponseEntity<String>> handleCreateException(Exception ex) {
        // Log the exception for debugging
        System.out.println("Error creating book: "+ex);

        return Mono.just(ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body("An error occurred while creating the book."));
    }

}
```