El uso de Reactor Framework poco a poco va introduciendose en el mundo de Spring Framework . La programación reactiva es algo que más pronto que tarde tendremos que ir asumiendo en nuestro día a día y las nuevas arquitecturas web nos acercaran a modelos como el de Node.js pero desde del mundo Java.

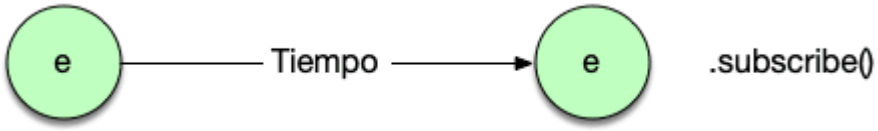
Reactor Framework y configuración

Lo primero que tendremos que hacer es construirnos un sencillo proyecto Maven que incluya las librerías de Reactor.

```
        <dependency>
                        <groupId>io.projectreactor</groupId>
                        <artifactId>reactor-core</artifactId>
                        <version>3.3.1.RELEASE</version>
                </dependency>
                <!--
https://mvnrepository.com/artifact/io.projectreactor/reactor-test -->
                <dependency>
                        <groupId>io.projectreactor</groupId>
                        <artifactId>reactor-test</artifactId>
                        <version>3.3.1.RELEASE</version>
                        <scope>test</scope>
                </dependency>
```

Usando la clase Mono

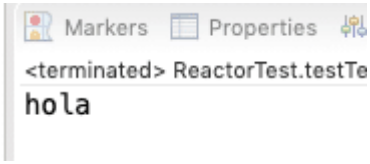
La clase más sencilla de la que dispone Reactor es la clase Mono que hace referencia a una variable asíncrona sencilla . Es decir estamos ante un elemento que se ejecuta y que lo hace de forma asíncrona.



Se puede almacenar una variable de tipo básico o un objeto. Veamos algunos de los ejemplos.

```
@Test
	public void testTexto() {
		Mono<String> mensaje=Mono.just("hola");
		mensaje.subscribe(System.out::println);
	}
```

En este primer bloque de código lo único que hacemos es generar un objeto mono y subscribirnos a su flujo . En este caso es algo tan sencillo que simplemente nos mostrará el texto “hola” por la consola.



Hasta aquí todo muy sencillo . El siguiente paso es construirnos una clase y generar un objeto Mono con ella.

```
package com.arquitecturajava;

public class Persona {

	private String nombre;
	private String apellidos;
	private int edad;
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
```

```
	}
	public String getApellidos() {
		return apellidos;
	}
	public void setApellidos(String apellidos) {
		this.apellidos = apellidos;
	}
	public int getEdad() {
		return edad;
	}
	public void setEdad(int edad) {
		this.edad = edad;
	}
	public Persona(String nombre, String apellidos, int edad) {
		super();
		this.nombre = nombre;
		this.apellidos = apellidos;
		this.edad = edad;
	}
	@Override
	public String toString() {
		return "Persona [nombre=" + nombre + ", apellidos=" +
apellidos + ", edad=" + edad + "]";
	}
}
```

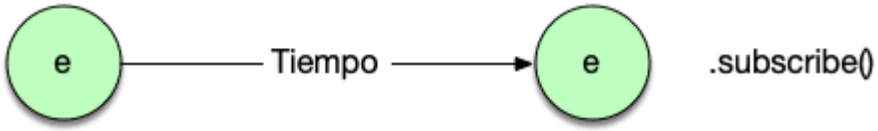
Creamos un bloque de código que nos puede ejecutar una variable de tipo Mono con este objeto.

```
@Test
	public void testObjeto() {
		Mono<Persona> mensaje=Mono.just(new
```

```
Persona("pepe","perez",20));
                mensaje.subscribe(System.out::println);
        }
```

El resultado por la consola es similar al resultado de la primera ejecución solamente que en este caso obtenemos los datos del propio objeto.

La clave aquí se encuentra en que el objeto Mono es un objeto asíncrono y se resolverá en un tiempo futuro.



En este caso su resolución es inmediata ya que no tenemos ningún tiempo de espera . Sin embargo esto lo podemos variar ya que el objeto lo permite en este caso podríamos escribir :

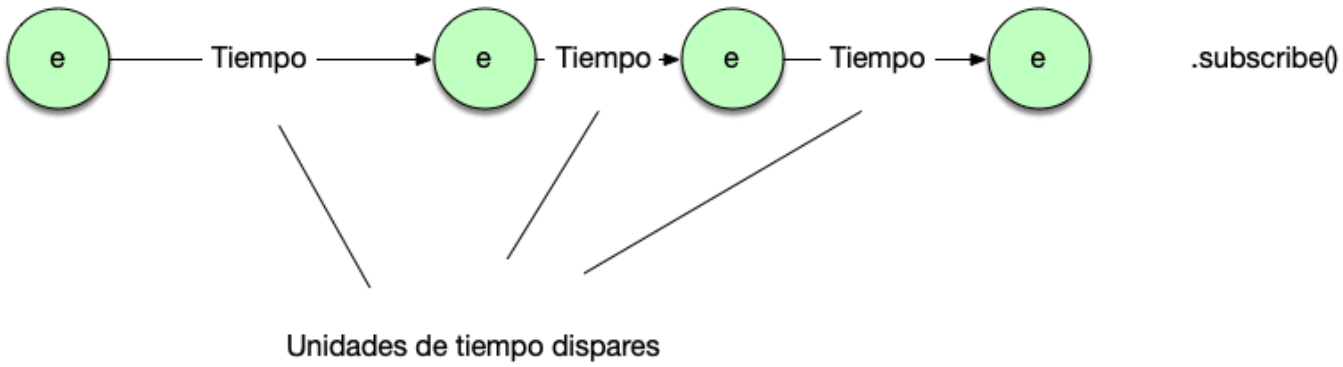
```
        @Test
        public void testAsincrono() throws InterruptedException {
                Mono<Persona> mensaje=Mono.just(new
Persona("pepe","perez",20)).delayElement(Duration.ofSeconds(5));
                mensaje.subscribe(System.out::println);
                Thread.sleep(10000);
        }
```

Es aquí donde le pedimos al objeto mono que espere un tiempo antes de resolverse usando el método delayElement. Obtendremos el mismo resultado pero tendremos que esperar esos 5 segundos para verlo en la consola

Es aquí donde Reactor Framework nos ayuda a simplificar las cosas y hace que la tarea sea trivial.

Usando la clase Flux

Flux a diferencia de Mono se encarga de gestionar una lista de elementos asíncronos.



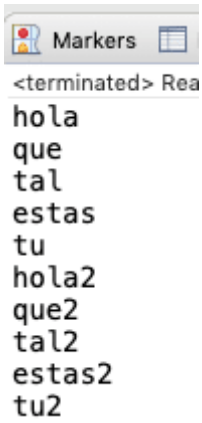
Nos aporta mayor flexibilidad en su uso . Veamos un ejemplo:

```
        @Test
        public void testLista() throws InterruptedException {
                Flux<String> mensaje=Flux.just("hola" ,"que"
,"tal","estas","tu").delayElements(Duration.ofSeconds(1));
                mensaje.subscribe(System.out::println);
                Thread.sleep(10000);
        }
```

En este caso estamos recorriendo con el método subscribe una lista de elementos cada uno de los cuales nos lleva segundo a segundo. La ventaja de Reactor Framework es que nos permite combinar estructuras asíncronas de forma muy sencilla . Por ejemplo podemos usar el método concat para combinar dos flujos asíncronos.

```
                Flux<String> mensajes1=Flux.just("hola" ,"que"
,"tal","estas","tu").delayElements(Duration.ofSeconds(1));
                Flux<String> mensajes2=Flux.just("hola2" ,"que2"
,"tal2","estas2","tu2").delayElements(Duration.ofSeconds(1));
mensajes1.concatWith(mensajes2).subscribe(System.out::println);
                Thread.sleep(15000);
```

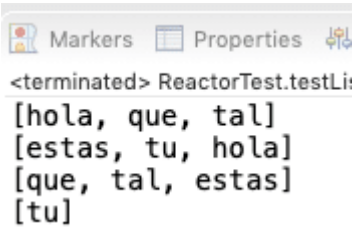
Esto hará que la lista de elementos nos salga en pantalla una a una . Primero la primera lista de elementos segundo a segundo. Para luego imprimir de la misma forma la segunda.



Este es un ejemplo de cómo podemos usar Flux y Reactor para combinar dos listas de elementos asíncronos. Sin embargo Reactor soporta situaciones mucho más diversas. Por ejemplo podemos querer esperar hasta tener un numero concreto de elementos antes de imprimirlos . Esta es una operación asíncrona básica y se denomina buffer ,vamos a verla.

```
 @Test
        public void testLista3() throws InterruptedException {
                Flux<String> mensajes1=Flux.just("hola" ,"que"
,"tal","estas","tu").delayElements(Duration.ofSeconds(1));
                Flux<String> mensajes2=Flux.just("hola" ,"que"
,"tal","estas","tu").delayElements(Duration.ofSeconds(1));
mensajes1.concatWith(mensajes2).buffer(3).subscribe(System.out::printl
n);
                Thread.sleep(15000);
```

Como se puede observar el código prácticamente no cambia simplemente añadimos la operación de buffer. Ahora bien el resultado en la consola cambia de forma significativa.



En este caso se van almacenando los elementos para devolverlos como un array hasta llegar a 3. El framework Reactor es el encargado de simplificar los flujos de programación asíncrona complejos y es la base de Spring WebFlux como framework asíncrono de servidor a nivel de aplicaciones web

Otros artículos relacionados

- Spring Boot
- Flux vs Mono
- Spring Boot JPA
- WebFlux