¿Flux vs Mono y la programación reactiva? . Todos estamos cada día más interesados en los conceptos que pertenecen a la programación Reactiva. Lamentablemente en muchas ocasiones es muy difícil entender cómo funciona este tipo de programación y como conceptos fundamentales de ella trabajan . Sobre todo si van apareciendo frameworks y más frameworks que generan capas de abstracción sobre nuestro código haciéndolo más difícil de comprender . Vamos a construir un ejemplo sencillo que nos ayude a entender cómo funcionan estas cosas. Para ello nos vamos a construir en un primer lugar un ejemplo básico de Spring y un servicio REST que nos permita identificar el problema ,vamos con ello. El primer paso es construir una aplicación clásica con Spring Boot para ello solicitaremos que el proyecto maven incluya lo siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.arquitecturajava</groupId>
  <artifactId>rest1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>rest1</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
```

```
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>
```

Es momento de construir un sencillo servicio de Spring Framework el cual nos devuelva unas cadenas de texto elementales

```
package com.arquitecturajava.rest1;

import org.springframework.stereotype.Service;
```

```
@Service
public class HolaService {

  public String hola() {

    try {
      Thread.sleep(3000);
    } catch (InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }

    return "hola sincrono";

  }

  public String hola2() {

    try {
      Thread.sleep(3000);
    } catch (InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }

    return "hola sincrono 2";

  }
}
```

Este código tiene de especial que cada vez que invocamos cada uno de los métodos del servicio hola y hola2 pondremos a dormir el Thread principal durante 3 segundos . Es

momento de construir un servicio REST que invoque a estos métodos y nos permite combinarlos de alguna forma.

```
package com.arquitecturajava.rest1;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping
public class ControladorHola {

  @Autowired
  HolaService servicio;

  @RequestMapping("/hola")
  public String hola() {
    return servicio.hola();
  }
  @RequestMapping("/hola2")
  public String hola2() {
    return servicio.hola2();
  }
  @RequestMapping("/holas")
  public String holas() {
    long t1= System.currentTimeMillis();
    String texto=servicio.hola()+ servicio.hola2();
    long t2= System.currentTimeMillis();
    System.out.println(t2-t1);
    return texto;
```

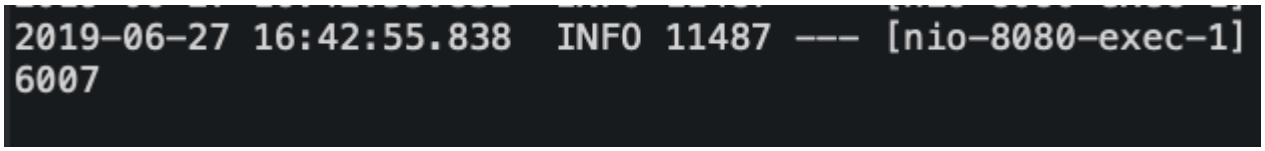
```
  }
}
```

En este caso hemos inyectado con autowired el servicio y mapeado cada uno de sus métodos pero ademas hemos añadido un método extra que nos permite combinar ambos (el método holas) . Si ejecutamos esto vía Web nos daremos cuenta de que la ejecución tarda 6 segundos en mostrar el mensaje en el navegador.

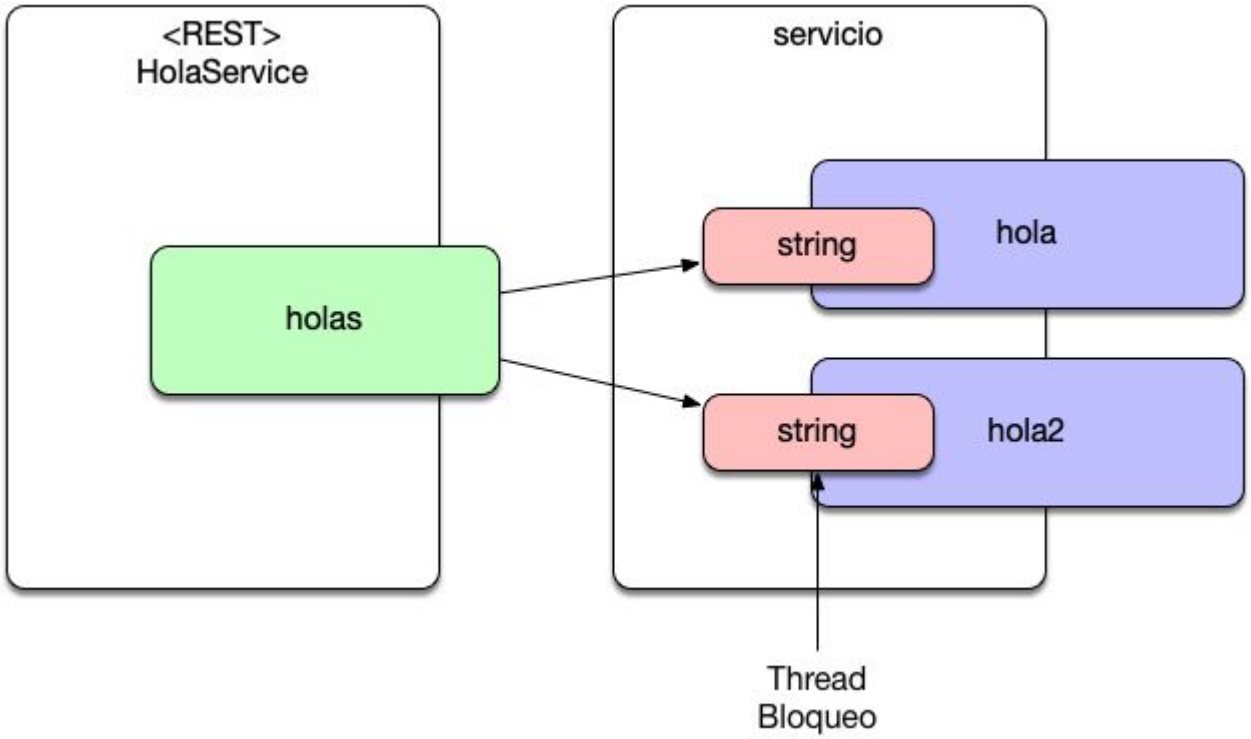
---

hola sincronohola sincrono 2

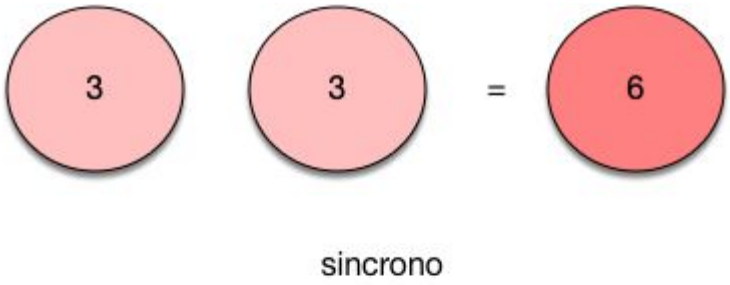
Si vemos los tiempos de ejecución por la consola con los logs que hemos realizado veremos que son 6 segundos:



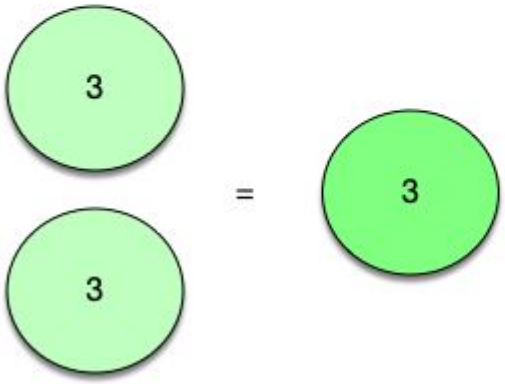
Es razonable ya que ambos métodos se ejecutan de forma sincrona , hay que esperar a que finalice el primero para ejecutar el segundo no tenemos ninguna otra opcion.



Si lo pensamos un poco más a detalle quizás cada método pudiera haberse ejecutado de forma asíncrona de tal forma que su ejecución no bloqueara o obligara a esperar para ejecutar el siguiente. Este es un tema interesante y es en el que Node.js basa sus principios . Es decir la ejecución del programa puede continuar mientras una petición asíncrona se resuelve de tal forma q por ejemplo si dos peticiones son asíncronas y cada una tarda 3 segundos se ejecuten ambas en paralelo .



sincrono

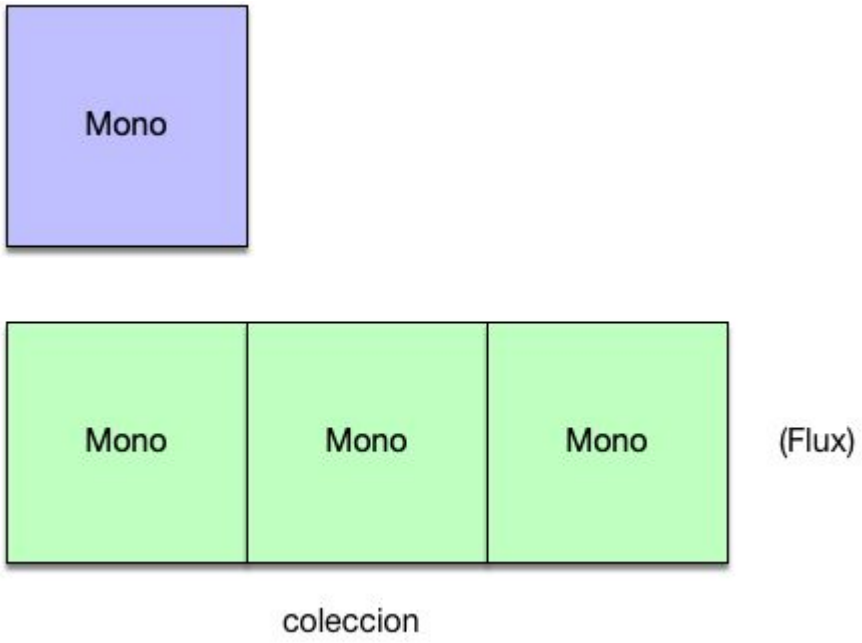


asincrono

Esto evidentemente mejora los tiempos de respuesta ya que no bloqueamos la ejecución de nuestro código. El problema es que para conseguir este tipo de comportamiento necesitamos utilizar un nuevo framework a nivel de Spring y este framework es el que se denomina Spring Reactor.

Flux vs Mono y Spring Reactor

Vamos a pasar ahora a hablar de Flux vs Mono y como Spring Reactor define estos conceptos a la hora de gestionar el uso de la programación Reactiva. ¿Para que sirve el tipo Mono y para que sirve el tipo Flux? . Muy Sencillo Mono hace referencia a un elemento de programación asincrona a uno solo y Flux hace referencia a un conjunto de elementos .



Si nos fijamos en el ejemplo anterior de programación síncrona tenemos una situación en la que cada método devuelve un String y otra que bueno aunque devolvemos un String podríamos decir que es un String compuesto de 2 cadenas cada una con su propio mensaje. Por lo tanto por hacer un simil nos encontramos ante una situación en la cual tenemos dos métodos que nos devuelven objetos Mono y un método que nos devolvería un Flux . Vamos a construir nuestro primer ejemplo con programación Reactiva. El primer paso es construirmos como siempre el proyecto con Spring Boot , en este caso no se trata de un proyecto normal y corriente sino que será reactivo.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
```

```
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.arquitecturajava</groupId>
<artifactId>rest2</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>rest2</name>
<description>Demo project for Spring Boot</description>

<properties>
  <java.version>1.8</java.version>
</properties>

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

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
```

```
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>
```

Construyendo un ejemplo

Podemos observar como hemos añadido el starter de WebFlux que es el que necesitamos en Spring para poder usar los conceptos fundamentales de Reactor. El siguiente paso es volver a construir nuestras clases de Spring pero usando en este caso programación asíncrona con los conceptos de Mono y Flux

```
package com.arquitecturajava.rest2;

import java.time.Duration;

import org.springframework.stereotype.Service;

import reactor.core.publisher.Mono;

@Service
public class HolaService {

  public Mono<String> hola() {

    return Mono.just("hola
asincrono").delayElement(Duration.ofSeconds(3));
```

```
  }

  public Mono<String> hola2() {
    return Mono.just("hola
asincrono").delayElement(Duration.ofSeconds(3));


  }
}
```

El código se parece muchisimo al anterior solo que tiene la parte especial de que tratamos con el concepto de Mono en este caso . Cada vez que invoquemos estos métodos tardarán en devolvernos el valor 3 segundos cada uno de ellos . Eso sí no son métodos que se bloqueen uno al otro sino que se pueden ejecutar de forma paralela. Vamos a ver como queda la clase de servicio REST que se encarga de invocar de forma simultanea a ambos.

```
package com.arquitecturajava.rest2;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@RestController
@RequestMapping
public class ControladorHola {

  @Autowired
  HolaService servicio;
```

```
  @RequestMapping("/hola")
  public Mono<String> hola() {
    return servicio.hola();
  }
  @RequestMapping("/hola2")
  public Mono<String> hola2() {
    return servicio.hola2();
  }
  @RequestMapping("/holas")
  public Flux<String> holas() {
    Mono<String> mono1= servicio.hola();
    Mono<String> mono2=servicio.hola2();
    Flux<String> flujo=Flux.merge(mono1,mono2);
    return flujo;
  }
}
```

Cómo podemos ver el método de /holas se encarga de combinar ambos tipos Monos , recordemos que cada uno de estos tipos es único y representa un elemento. Por lo tanto la combinación de ambos será un tipo Flux que es lo que devolvemos . Si ejecutamos nuestro código con Spring Boot y arrancamos el proyecto nos podremos dar cuenta que hay muchas cosas que cambian. La primera de ellas es que este código no se ejecuta sobre el paraguas de Tomcat lo cual es lo más habitual dentro del mundo Java sino que lo hace dentro de Netty un framework de programación asíncrona puro.



Si medimos el tiempo de ejecución de la url de http://localhost/holas nos daremos cuenta

que ahora tarda 3 segundos en devolvernos ambos mensajes y no 6 como lo hacía el código anterior acabamos de ver como Spring framework usa los conceptos de Flux vs Mono para simplificar de forma importante la gestión de la programación asíncrona.

Otros artículos relacionados

- Reactor Framework y sus clases
- Spring Reactive y Thymeleaf
- Spring Reactor y manejo de transformaciones
- Curso Spring 5 WebFlux y Programación Reactiva