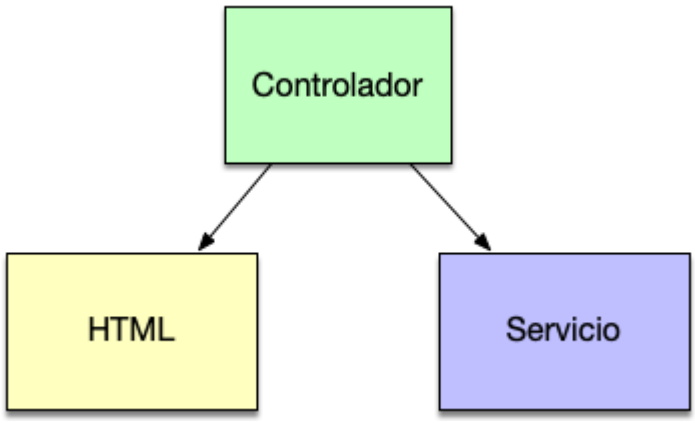
Tabla de Contenidos


- Spring 4 y REST
- REST y Clientes
- REST ,Combinando peticiones
- Spring WebFlux
- Spring Cliente RestTemplate
- Spring WebFlux al rescate
- Otros articulos relacionados


Spring WebFlux es uno de los nuevos frameworks que vienen con la nueva versión de Spring 5. ¿Para qué sirve exactamente Spring WebFlux? . Vamos a repasar algunos de los conceptos fundamentales en los cuales Spring Framework se ha basado en sus últimas versiones. Para empezar vamos a revisar un poco Spring MVC que en su momento fue un avance importante y substituyo en muchos lugares a Struts como framework de capa de presentación.

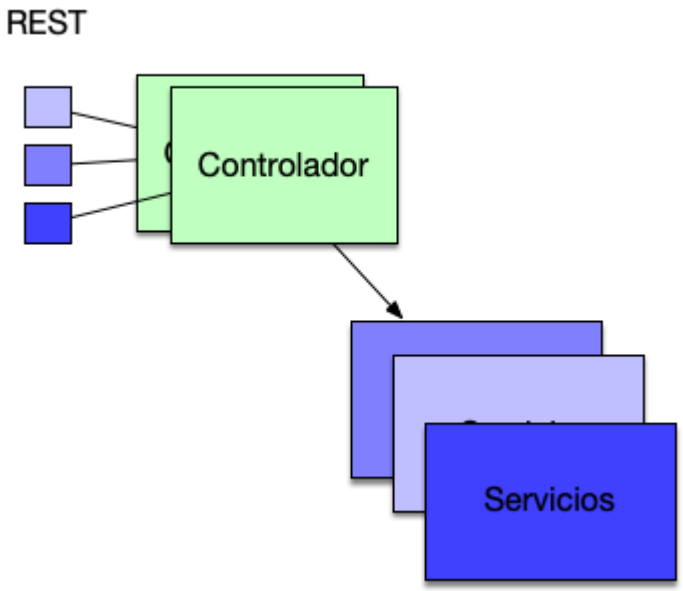
Este framework adquirió madurez en la versión 3 de Spring Framework . Siempre se ha usado para construir aplicaciones Web MVC clásicas



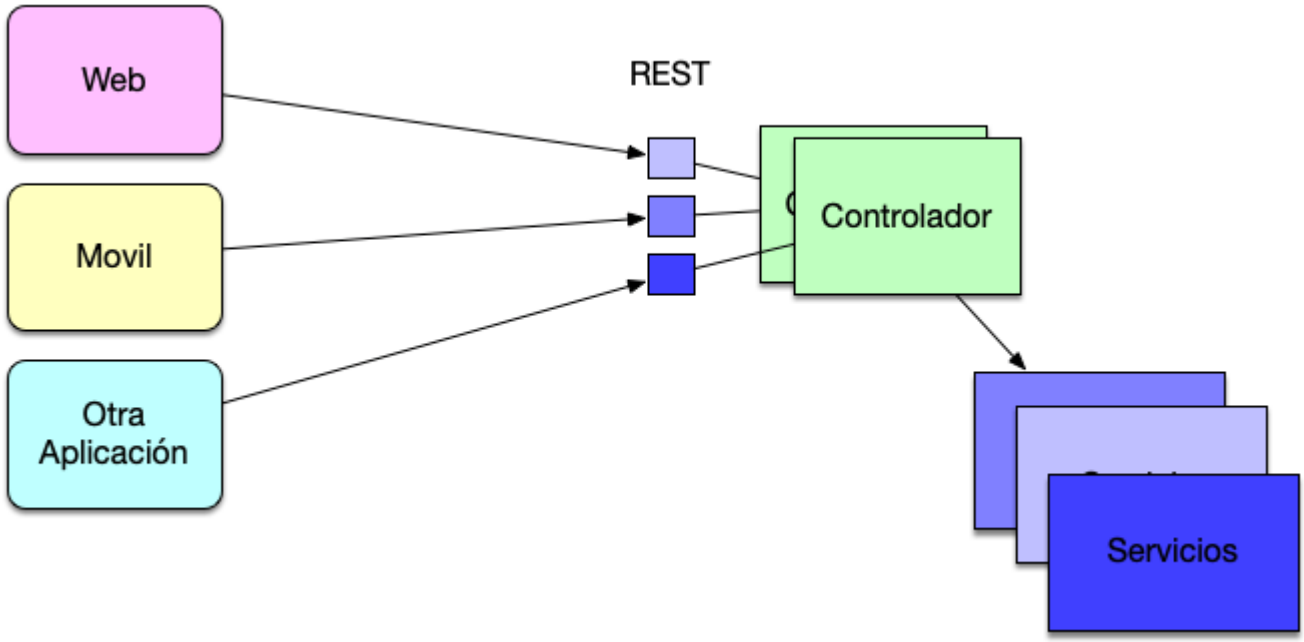

Estas aplicaciones se construyeron de forma masiva ya que aportaban orden en la construcción de aplicaciones web y la facilidad de acceder a ellas desde cualquier navegador y lugar.

# Spring 4 y REST

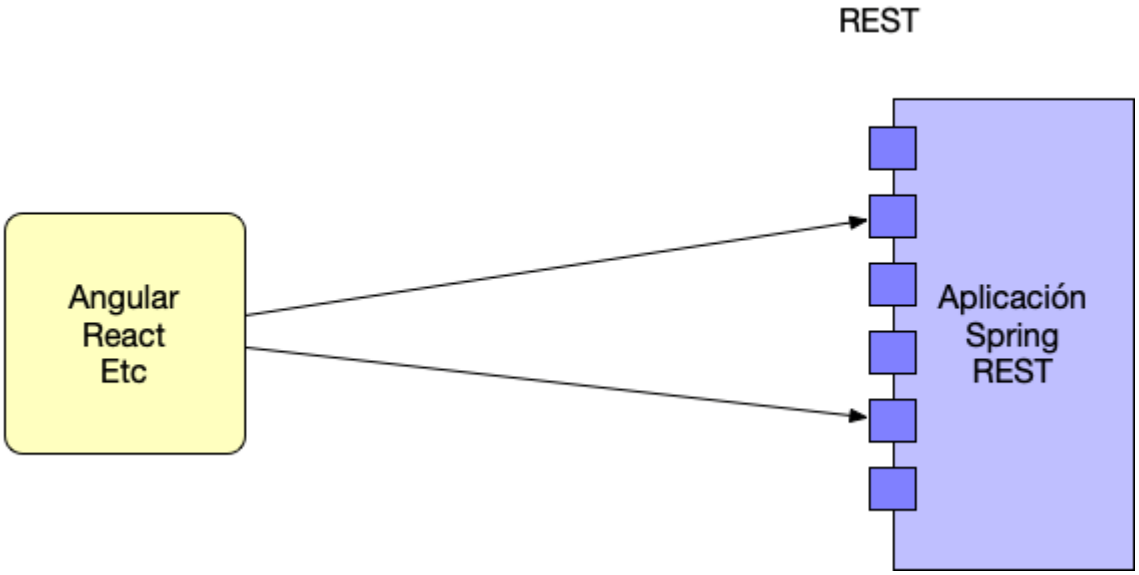
Con el paso del tiempo se necesitaron arquitecturas más flexibles en las cuales se priorizara la construcción de servicios REST y cualquier tipo de cliente se pudiera conectar a ellas .




Podría ser una aplicación Web , una aplicación móvil o otra aplicación que hiciera el rol de cliente aunque fuera una aplicación de lado servidor . La publicación se servicios REST abre las puertas a la flexibilidad a nivel de el tipo de clientes.



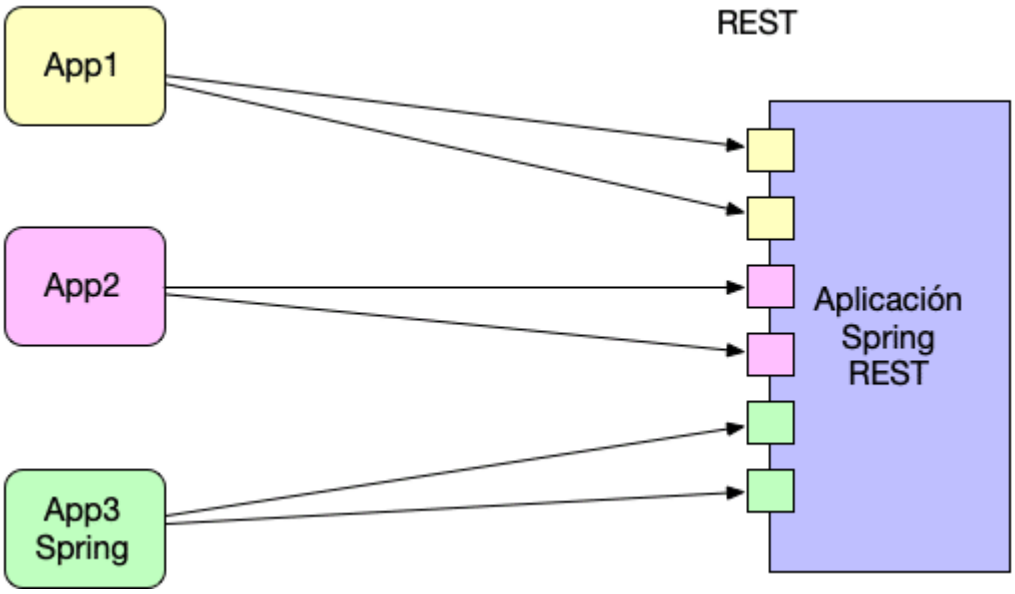

Es en este tipo de aplicaciones en las cuales es habitual por ejemplo encontrarnos un cliente de Angular o React ya que acceden a servicios REST que publican datos en formato JSON y los usan para presentar la información al usuario.




# REST y Clientes

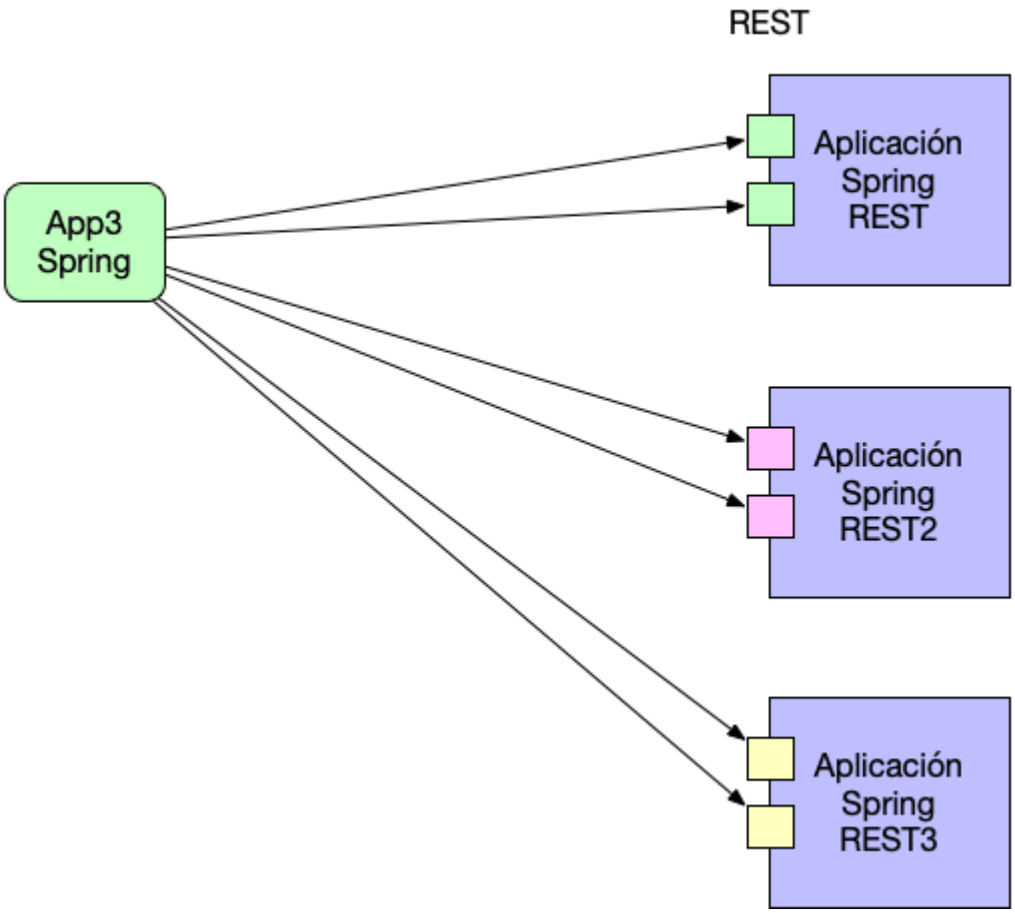
Hoy en día nos encontramos que una parte importante de los clientes a nivel de REST pueden ser clientes que necesitan consultar una información puntual de nuestra aplicación

a nivel de REST . Es decir no necesitan acceder a todo el abanico de funcionalidad que publicamos sino que necesitan un pequeño subconjunto.



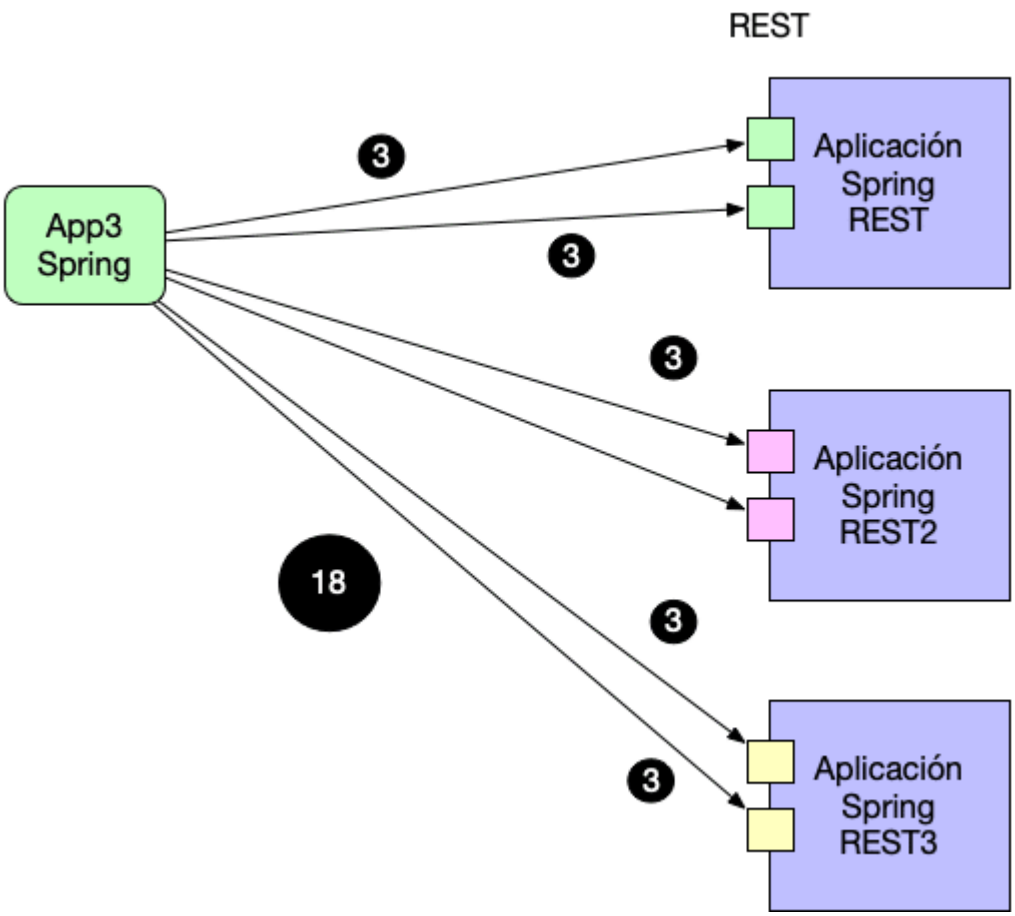

## REST ,Combinando peticiones

Con este tipo de arquitecturas es muy habitual tener la necesidad que por ejemplo nuestra aplicación cliente de Spring necesite acceder a servicios REST de multiples aplicaciones que en este caso hacen el rol de aplicaciones servidoras.

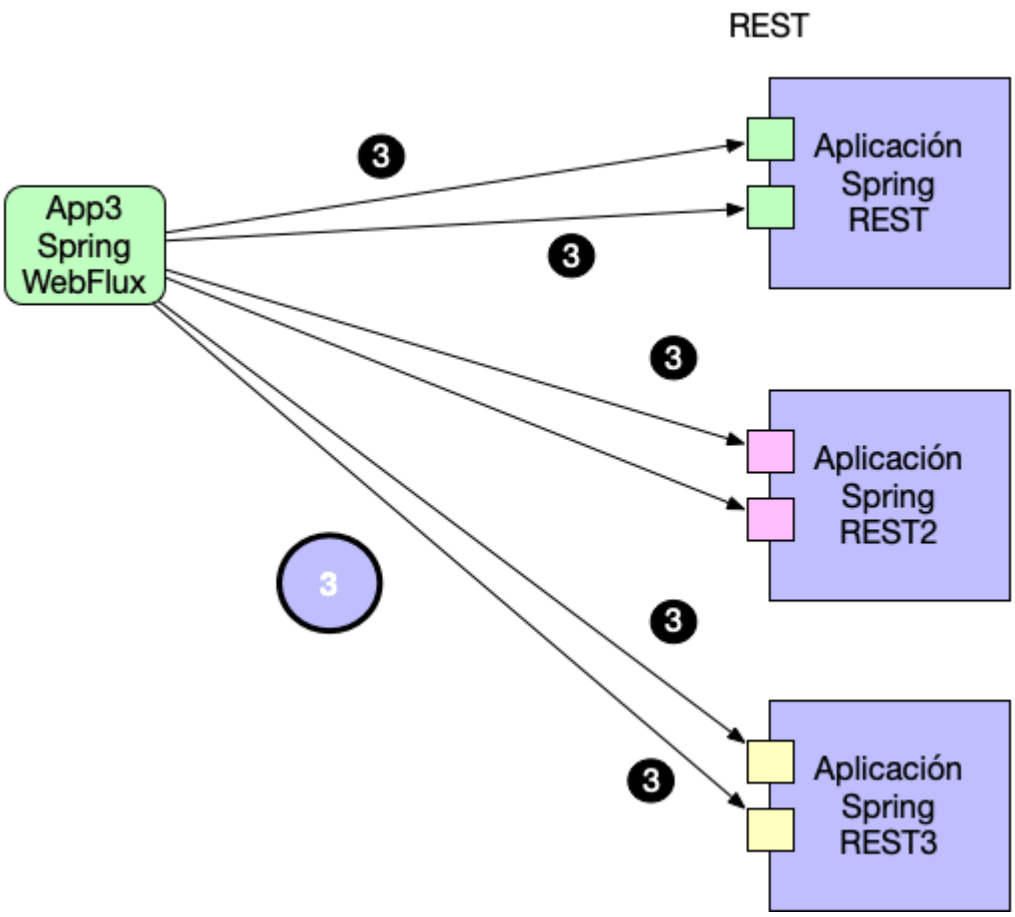



# Spring WebFlux

Es aquí donde hay un salto en cuanto a como podemos tratar este tipo de arquitectura. ¿Podemos usar Spring 4 de forma clásica? . Esta claro que si y que puede publicar servicios REST y puede consumirlos con REST Templates o otras soluciones como Feign . Ahora bien existe un problema si cada petición tarda 3 segundos en hacerse a nivel de REST y tenemos que realizar las 6 peticiones que aparecen en el diagrama . Habrá que sumar los tiempos de cada una ya que se trata de lo que habitualmente se conoce como peticiones bloqueantes es decir yo realizo una petición y espero a su resultado para poder realizar la siguiente :




Esto es algo que Spring 4 no tiene muchas capacidades para resolver . Sin embargo existen en muchas casuisticas en las cuales nos gustaría realizar todas las peticiones de forma simultanea y en cuanto tengamos el resultado de todas dar por finalizada la tarea. Es decir si cada petición tarda 3 segundos y hacemos todas a la vez . En 3 segundos tendremos todos los datos ya en nuestra aplicación y no en 18.




La ventaja es clara . Es este tipo de situaciones en las cuales Spring WebFlux brilla ya que nos provee de un framework no bloqueante . Vamos a construir un ejemplo que nos ayude a entender las cosas. En primer lugar vamos a utilizar un proyecto servidor que nos devuelva una lista de facturas utilizando Spring Boot . En él únicamente añadiremos la dependencia Web y Reactive Web a continuación se muestra el pom.xml.

```
<?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
https://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>2.2.2.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.arquitecturajava</groupId>
	<artifactId>servidor</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>servidor</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-
webflux</artifactId>
		</dependency>

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

```
                        <exclusions>
                                <exclusion>
<groupId>org.junit.vintage</groupId>
                                        <artifactId>junit-vintage-
engine</artifactId>
                                </exclusion>
                        </exclusions>
                </dependency>
                <dependency>
                        <groupId>io.projectreactor</groupId>
                        <artifactId>reactor-test</artifactId>
                        <scope>test</scope>
                </dependency>
                <dependency>
                        <groupId>com.arquitecturajava</groupId>
                        <artifactId>dominio</artifactId>
                        <version>0.0.1-SNAPSHOT</version>
                </dependency>
        </dependencies>

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

</project>
```

Una vez tenemos configurado Spring Boot vamosa crear un @RestController .

```
package com.arquitecturajava.servidor;

import java.util.ArrayList;
import java.util.List;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import com.arquitecturajava.dominio.Factura;

@RestController
public class FacturasController {

	@GetMapping("/facturas")
	public List<Factura> buscarTodas() {
		List<Factura> lista= new ArrayList<Factura>();
		lista.add(new Factura(1,"ordenador",200));
		lista.add(new Factura(2,"tablet",300));
		try {
			Thread.sleep(3000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		return lista;
	}
}
```

Este controlador nos devolverá una lista de Facturas . Esta lista se encuentra en otro proyecto que se denomina dominio.

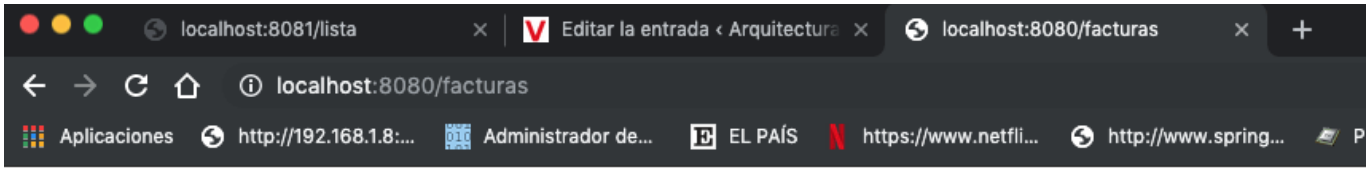
```
package com.arquitecturajava.dominio;

public class Factura {

	private int numero;
	private String concepto;
	private double importe;
	public int getNumero() {
		return numero;
	}
	public void setNumero(int numero) {
		this.numero = numero;
	}
	public String getConcepto() {
		return concepto;
	}
	public void setConcepto(String concepto) {
		this.concepto = concepto;
	}
	public double getImporte() {
		return importe;
	}
	public void setImporte(double importe) {
		this.importe = importe;
	}
	public Factura(int numero, String concepto, double importe) {
		super();
		this.numero = numero;
		this.concepto = concepto;
		this.importe = importe;
	}
```

```
	public Factura() {
		super();
	}
}
```

Una vez que lo tenemos configurado , arrancamos el proyecto de Boot y accedemos a la URL que nos tardara en mostrar la lista unos 3 segundos.




# Spring Cliente RestTemplate

Es momento de construir otro proyecto con Spring Boot y RestTemplates y acceder a los datos para imprimirlo en una página HTML . Para ello crearemos un proyecto de SpringBoot similar al anterior pero que incluya las dependencias de ThymeLeaf

En este proyecto nos crearemos una clase de servicio que se encargue de invocar a la url de /facturas y un controlador que la invoque varias veces seguidas.

```
package com.arquitecturajava.cliente;

import java.util.Arrays;
import java.util.List;

import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import com.arquitecturajava.dominio.Factura;
@Service
public class ClienteFacturasService {
```

```
        public List<Factura> buscarTodas() {

                RestTemplate plantilla = new RestTemplate();
                Factura[] facturas =
plantilla.getForObject("http://localhost:8080/facturas",
Factura[].class);
                List<Factura> lista = Arrays.asList(facturas);
                return lista;

        }
}
```

```
package com.arquitecturajava.cliente;

import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

import com.arquitecturajava.dominio.Factura;

@Controller
public class FacturaClienteController {

        @Autowired
        ClienteFacturasService servicio;

        @RequestMapping("/lista")
        public String lista(Model modelo) {
```

```
                List<Factura> lista= new ArrayList<Factura>();
                lista.addAll(servicio.buscarTodas());
                lista.addAll(servicio.buscarTodas());
                lista.addAll(servicio.buscarTodas());
                modelo.addAttribute("lista", lista);
                return "lista";
        }
}
```

Una vez realizada esta operación nos queda de ver el código de la plantilla y ejecutar:

```
<table>
    <thead>
        <tr>
            <th> id </th>
            <th> concepto </th>
               <th> importe </th>
        </tr>
    </thead>
    <tbody>
        <tr th:each="factura : ${lista}">
            <td><span th:text="${factura.numero}"> </span></td>
            <td><span th:text="${factura.concepto}"></span></td>
              <td><span th:text="${factura.importe}"></span></td>
        </tr>
    </tbody>
</table>
```

No nos sorprenderá mucho que tarda más de 9 segundos en renderizar la tabla con los datos en la web.

| id | concepto | importe |
|---|---|---|
| 1 | ordenador | 200.0 |
| 2 | tablet | 300.0 |
| 1 | ordenador | 200.0 |
| 2 | tablet | 300.0 |
| 1 | ordenador | 200.0 |
| 2 | tablet | 300.0 |

## Spring WebFlux al rescate

Cuando las aplicaciones comienzan a funcionar con este enfoque en las cuales unos servicios REST invocan a otros y estos a otros con combinaciones etc . El problema del bloqueo se hace más y más evidente. Es aquí donde podemos usar Spring WebFlux. El framework nos permitirá construir un servicio REST no bloqueante de tal forma que podamos hacer varias peticiones simultaneas y las procese en paralelo. Vamos a ver el código del Controlador REST.

```
package com.arquitecturajava.servidor;

import java.time.Duration;
import java.util.ArrayList;
import java.util.List;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import com.arquitecturajava.dominio.Factura;

import reactor.core.publisher.Flux;
```

```
@RestController
public class FacturasController {

	@GetMapping("/facturas")
	public Flux<Factura> buscarTodas() {
Flux<Factura> lista= Flux.just(new Factura(1,"ordenador",200)
,new Factura(2,"tablet",300)).delaySequence(Duration.ofSeconds(3));
		return lista;
	}
}
```

Como se puede observar hace uso de un objeto de tipo Flux . Este tipo representa una lista asíncrona de objetos. Ademas obligamos a esperar 3 segundos antes de devolver la lista. Con esto abrimos las capacidades noblocking de Spring.

El siguiente paso es ver como construimos el cliente . En este caso usaremos un servicio que se encargará de realizar como en el caso anterior tres peticiones.

```
import org.springframework.web.reactive.function.client.WebClient;

import com.arquitecturajava.dominio.Factura;

import reactor.core.publisher.Flux;
@Service
public class ClienteFacturasService {

	public Flux<Factura> buscarTodas() {

		WebClient cliente =
WebClient.create("http://localhost:8080/facturas");
		Flux<Factura>
facturas=cliente.get().retrieve().bodyToFlux(Factura.class);
```

```
                Flux<Factura>
facturas2=cliente.get().retrieve().bodyToFlux(Factura.class);
                Flux<Factura>
facturas3=cliente.get().retrieve().bodyToFlux(Factura.class);
                Flux<Factura>
todas=Flux.merge(facturas,facturas2,facturas3);
                System.out.println(todas);
                return todas;

        }
}
```

Estas tres peticiones se realizan a traves de un WebClient que es el nuevo objeto de Spring 5 que permite programación reactiva que no bloquea. Convertimos cada una de las peticiones en un objeto Flux y las fusionamos con el método merge . Realizado este último paso devolvemos la estructura para que un controlador pueda asignarlo a una plantilla.

```
package com.arquitecturajava.cliente;

import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import
org.thymeleaf.spring5.context.webflux.IReactiveDataDriverContextVariab
le;
import
org.thymeleaf.spring5.context.webflux.ReactiveDataDriverContextVariabl
e;
```

```
import com.arquitecturajava.dominio.Factura;

@Controller
public class FacturaClienteController {

        @Autowired
        ClienteFacturasService servicio;

        @RequestMapping("/lista")
        public String lista(final Model modelo) {
                List<Factura>
lista=servicio.buscarTodas().collectList().block();
                modelo.addAttribute("lista", lista);
                return "lista";
        }
}
```

Acabamos de construir nuestro primer ejemplo con Spring WebFlux la lista solo tardará 3 segundos en cargarse.

| id | concepto | importe |
|---|---|---|
| 1 | ordenador | 200.0 |
| 2 | tablet | 300.0 |
| 1 | ordenador | 200.0 |
| 2 | tablet | 300.0 |
| 1 | ordenador | 200.0 |
| 2 | tablet | 300.0 |

Es el tiempo que tardan todas las peticiones en ejecutarse de forma simultanea. Eso sí nadie nos asegura el orden de llegada de los datos .

## Otros articulos relacionados

- Flux vs Mono ,Spring y la programación reactiva
- Curso Spring 5 WebFlux y Programación Reactiva
- Spring Webflux Test con StepVerifier
- Spring WebFlux Router y REST