



	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

Índice

Spring.....	1
¿Qué es Spring?.....	1
Nuevas características de Spring 5.....	1
Programación reactiva.....	3
¿Qué es?.....	3
Objetivo.....	3
Posibles casos de uso.....	4
¿Por qué su boom?.....	4
¿La solución a todos nuestros problemas?.....	5
Spring WebFlux.....	5
Reactive Streams.....	6
¿Qué es?.....	6
Componentes.....	6
API.....	7
Reactor.....	7
Flux.....	8
Mono.....	8
Operadores.....	9
Parte servidora.....	10
Modelo de programación basado en anotaciones.....	10
Modelo de programación funcional.....	11
HandlerFunctions.....	11
ServerRequest.....	11
ServerResponse.....	12
RouterFunctions.....	13
¿Cómo ejecutar una RouterFunction?.....	14
HandlerFilter.....	15
Parte cliente.....	16
Soporte a Reactive WebSocket.....	16
WebSocket.....	16
Introducción.....	16
¿Qué es?.....	17
WebSocketClient.....	17
Testing.....	18
WebTestClient.....	18
StepVerifier.....	18
Referencias.....	19

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

Spring

¿Qué es Spring?

Java, es un lenguaje de programación orientado a objetos, que es usado por miles de millones de dispositivos. Si eres programador Java seguramente habrás oído hablar de Spring y esto se debe a que es una herramienta de uso diario en el área del desarrollo web.


Spring es un framework open source del lenguaje de programación Java que nos permite desarrollar aplicaciones de manera más rápida, eficaz y corta, saltándonos tareas repetitivas y ahorrándonos líneas de código.

Está dividido en módulos. Se puede elegir qué módulos se necesitan en las aplicaciones. En el centro están los módulos del core, que incluyen un modelo de configuración y un mecanismo de inyección de dependencia. Además, Spring Framework proporciona soporte fundamental para diferentes arquitecturas de aplicaciones, incluyendo mensajería, datos transaccionales y persistencia, y web.

Nuevas características de Spring 5

La nueva versión nos proporciona:

- **Actualización del JDK:** todo Spring 5 ejecuta sobre Java 8.
- **Revisión del core del framework:** se ha modificado el core para que haga uso de las nuevas características de Java 8.
- **Actualización en el Core:** por ejemplo ahora se soporta carga desde el component index además del classpath scanning, esto mejora mucho el tiempo de carga en grandes aplicaciones
- **Programación Funcional con Kotlin:** Spring 5 incluye soporte para Kotlin.
- **Modelo de Programación reactiva:** es totalmente reactivo y no bloqueante.
- **Mejoras en los Tests:** para esto se soporta la extensión de JUnit 5, JUnit 5 Jupiter
- **Nuevas librerías soportadas:**

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

- [Jackson 2.6+](#)
- [EhCache 2.10+ / 3.0 GA](#)
- [Hibernate 5.0+](#)
- [JDBC 4.0+](#)
- [XmlUnit 2.x+](#)
- [OkHttp 3.x+](#)
- [Netty 4.1+](#)
- **Soporte discontinuado:**

A nivel del API, Spring Framework 5.0 ha suspendido el soporte para los siguientes paquetes:

- *beans.factory.access*
- *jdbc.support.nativejdbc*
- *mock.staticmock* del módulo *spring-aspects*.
- *web.view.tiles2M*. Ahora Tiles 3 es el requisito mínimo.
- *orm.hibernate3* y *orm.hibernate4*. Ahora, Hibernate 5 es el framework compatible.

Si usamos alguna de estas librerías debemos mantenernos en Spring 4.3:

- Portlet
- Velocity
- JasperReports
- XMLBeans
- JDO
- Guava

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

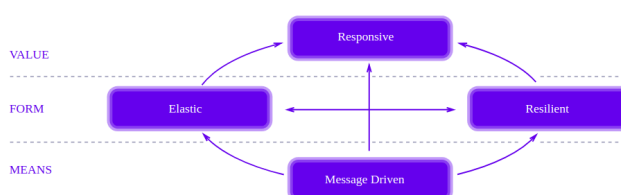
Programación reactiva

¿Qué es?

La programación reactiva es un paradigma de programación cuya concepción y evolución ha ido ligada a la publicación del [Reactive Manifesto](#), que define cuales son las características que debe tener un sistema para considerarse reactivo.

Un sistema reactivo es aquel que es:

- **Responsivo:** proporciona tiempos de respuesta rápidos y consistentes. Además define límites en dichos tiempos de respuesta, de forma que los problemas pueden ser detectados rápidamente y tratados de forma efectiva.
- **Resiliente:** permanece responsivo incluso cuando se encuentran situaciones de error.
- **Elástico:** se mantiene responsivo ante incrementos de la carga de trabajo.
- **Orientado a Mensajes:** se basa en el intercambio de mensajes asíncronos, lo que asegura el bajo acoplamiento, aislamiento, transparencia de ubicación y proporciona los medios para delegar errores como mensajes. Otro factor clave a tener en cuenta de este punto es que se favorece la aplicación de contrapresión cuando sea necesario. **La contrapresión es un mecanismo para asegurar que los productores no sobrecarguen a los consumidores.**



Objetivo

Este paradigma tiene varios objetivos:

1. **Propagar los cambios en un sistema requiriendo la menor cantidad de esfuerzo.** Imaginemos una hoja de cálculo, en la que se toma una celda y se coloca una fórmula que depende de los valores de otras dos celdas. Cuando colocas valores en estas celdas referenciadas, el valor de la primera automáticamente cambia, reaccionando a los cambios. Efectivamente, los cambios se propagaron en nuestro sistema.

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

2. **Hacer más sencillo el trabajar con flujos de datos finitos o infinitos de manera asíncrona.**
Volvemos al ejemplo de la hoja de cálculo. Cada vez que se presiona una tecla se está enviando un dato. El procesador de textos reacciona de acuerdo a como fue programado para hacerlo, en este caso, colocar la letra en el lugar correspondiente. Este flujo es asíncrono porque el programa no sabe cuándo vamos a escribir la siguiente letra.
3. **Solventar la necesidad de responder a las limitaciones de escalado presentes en los modelos de desarrollo actuales, que se caracterizan por su desaprovechamiento del uso de la CPU debido al I/O, el sobreuso de memoria (enormes thread pools) y la ineficiencia de las interacciones bloqueantes.**

Posibles casos de uso

Su uso puede ser aplicable en los siguientes casos:

- **Aplicaciones con interfaz gráfica en la que el usuario interactúe con la aplicación:** hace un momento vimos el ejemplo de la hoja de cálculo con el teclado, pero se puede aplicar la misma idea a los clicks que hace el usuario o a los botones de un control de consola. También hablábamos de la dependencia de las celdas. Si la celda B depende de la celda A, y la celda C depende de ambas celdas A y B, entonces, ¿cómo se propagan los cambios en A, asegurando que C se actualice antes de que cualquier evento de cambio se envíe a B?
- **Cuando trabajemos con información en tiempo real:** por ejemplo las lecturas provistas por un termómetro, un sismógrafo o, en el caso de los automóviles autónomos, la inmensa cantidad de información que reciben a través de sus sensores.
- **Trabajo con flujos interminables de datos:** como por ejemplo la inmensa cantidad de tweets que existen, o las transferencias bancarias que ocurren en un banco al día.
- **Orquestar llamadas a servicios externos:** Actualmente, muchos servicios de back-end son RESTful (es decir, operan a través de HTTP), por lo que el protocolo subyacente es fundamentalmente de bloqueo y síncrono. A menudo la implementación de dichos servicios implica llamar a otros servicios, y aún más servicios dependiendo de los resultados de las primeras llamadas. Con tanto proceso de E/S sucediendo, esperar a que se complete una llamada antes de enviar la próxima solicitud, puede ser lento y frustrante. Por lo tanto, las llamadas a servicios externos, especialmente las orquestaciones complejas de dependencias entre llamadas, podrían optimizarse.

¿Por qué su boom?

El motivo de su creciente uso es una **utilización eficiente de los recursos**, o en otras palabras, gastar menos dinero en servidores y centros de datos. La promesa de la Reactividad es que se puede hacer más con

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

menos. Específicamente puedes procesar cargas de trabajo más altas con menos hilos. Aquí es donde la intersección de E/S asíncrona reactiva y no bloqueante pasa a primer plano. **Para el problema correcto, los efectos son drásticos.**

¿La solución a todos nuestros problemas?

Como decíamos antes, para un caso adecuado, los efectos de la programación reactiva son abrumadoramente notables, pero ¿qué pasa si lo usamos para el problema equivocado?. Los efectos pueden ser contrarios y en realidad empeorar las cosas. **Si tenemos una aplicación que funciona bien, no es necesario cambiarla. La programación imperativa es la forma más fácil de escribir, comprender y depurar código.** Incluso eligiendo el problema correcto, la Reactividad no resuelve los problemas por nosotros, simplemente nos proporciona una serie de herramientas que pueden usarse para implementar soluciones.

Otra cosa a destacar es que, tan pronto como haya errores, por ejemplo una conexión de red mala, vamos a sufrir por diversos motivos:

- **El código que escribimos es declarativo, por lo que es difícil depurarlo.** Cuando se producen errores, los diagnósticos pueden ser muy opacos.
- Si cometemos un error y **bloqueamos una de nuestras devoluciones de llamada reactivas, retendremos todas las solicitudes en el mismo hilo.** En el mundo Reactivo, el bloqueo de una sola solicitud puede conducir a una mayor latencia para todas las solicitudes, y el bloqueo de todas las solicitudes puede tumbar un servidor porque las capas adicionales de búferes y subprocesos no están ahí para tomar el relevo.

Spring WebFlux

Como mencionábamos anteriormente, Spring-webflux es un nuevo módulo que nos proporciona Spring 5. Proporciona **soporte para clientes reactivos HTTP y WebSocket, así como para aplicaciones web de servidores reactivos** que incluyen REST, navegador HTML e interacciones estilo WebSocket.

Está basado en los siguientes aspectos clave de la programación reactiva:

- **Proporcionar aplicaciones no bloqueantes, asíncronas, accionadas por eventos** y que requieren un pequeño número de subprocesos para escalar verticalmente (es decir, dentro de la JVM) en lugar de horizontalmente (es decir, mediante clústeres).

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

- **Contrapresión**, que como ya vimos no es más que un mecanismo para asegurar que los productores no saturan a los consumidores. Por ejemplo, en un pipeline de componentes reactivos que se extiende desde la base de datos a la respuesta HTTP, cuando la conexión HTTP es demasiado lenta, el repositorio de datos también puede ralentizarse o detenerse por completo hasta que la capacidad de la red se libere.

Para conseguir esto, se adecua al estándar proporcionado por [Reactive Streams](#) usando la implementación de [Reactor](#) (aunque proporciona opciones y es totalmente compatible con el uso de RxJava también).

Reactive Streams

¿Qué es?

Reactive Streams es una especificación creada a través de la colaboración de la industria que también se ha adoptado en Java 9 como `java.util.concurrent.Flow`.

Su ámbito es encontrar un conjunto mínimo de interfaces, métodos y protocolos que describan las operaciones y entidades necesarias para lograr la meta: flujos de datos asíncros con contrapresión no bloqueante. De esta forma se permitirá la creación de muchas implementaciones de dichas interfaces que, en virtud de cumplir con las reglas, serán capaces de interoperar sin problemas, preservando los beneficios y características mencionados.

En resumen, Reactive Streams es un estándar y especificación para las librerías orientadas a Stream para la JVM que:

- **procesa un número potencialmente ilimitado de elementos**
- **en secuencia,**
- **pasando elementos de manera asíncrona entre los componentes,**
- **con contrapresión obligatoria no bloqueante.**

Componentes

La especificación consta de las siguientes partes:

- **El API:** especifica los tipos para implementar Reactive Streams y lograr la interoperabilidad entre diferentes implementaciones.
- **El *Technology Compatibility Kit* (TCK):** es un banco de pruebas estándar para las pruebas de conformidad de las implementaciones.

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

Las implementaciones son libres de implementar características adicionales no cubiertas por la especificación siempre que **cumplan con los requisitos del API y pasen las pruebas en el TCK.**

API

El API consta de los siguientes componentes que las implementaciones de Reactive Stream deben proporcionar:

1. **Publisher**
2. **Subscriber**
3. **Subscription**
4. **Processor**

Un Publisher es un proveedor de un número potencialmente ilimitado de elementos secuenciales, publicándolos de acuerdo con la demanda de sus Subscriber(s).

En respuesta a una llamada a `Publisher.subscribe(Subscriber)`, las posibles secuencias de invocación para los métodos en el `Subscriber` se proporcionan mediante el siguiente protocolo:

```
onSubscribe onNext* (onError | onComplete)?
```

Esto significa que `onSubscribe` siempre envía una señal, seguido por un número posiblemente ilimitado de señales en `onNext` (según lo solicitado por el `Subscriber`) seguido de una señal `onError` si hay una fallo, o una señal `onComplete` cuando no hay más elementos disponibles, durante el tiempo en que la `Subscription` no sea cancelada.

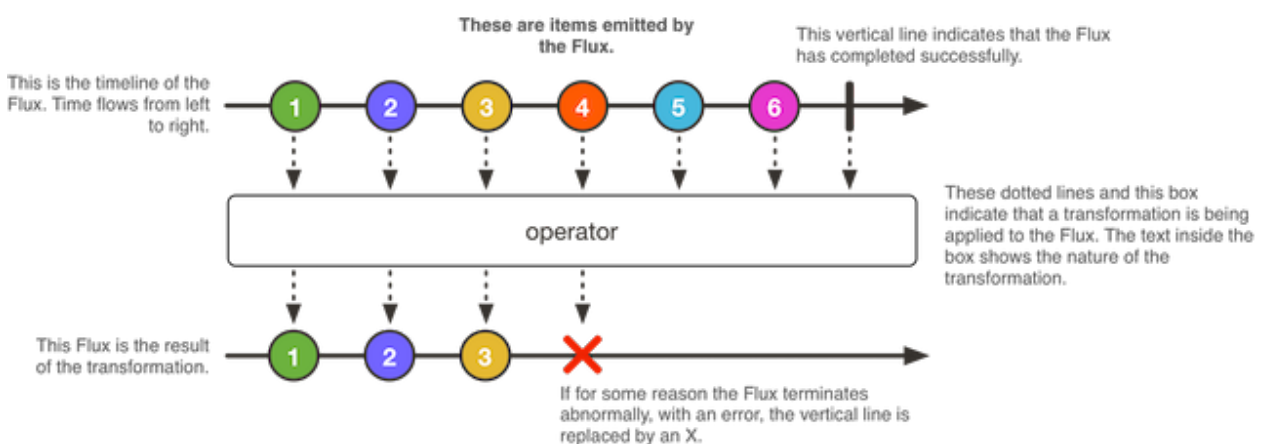
Reactor

Spring Framework usa [Reactor](#) internamente para su propio soporte reactivo. **Reactor es una implementación de Reactive Streams que amplía aún más el contrato básico de la interfaz Publisher de Reactive Streams con los tipos de API compuesta Flux y Mono para proporcionar operaciones declarativas en secuencias de datos de 0..N y 0..1.**

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

Flux

Flux es un publicador de una secuencia asíncrona, de 0 a N eventos, de un tipo de POJO específico, por lo que es genérico. Es decir, `Flux<T>` es un publicador de `T`. Dicha secuencia puede terminar opcionalmente (puede ser una secuencia ilimitada) por una señal de finalización o un error. En realidad, **solo hay dos cosas que puede hacer con un Flux: operar sobre él** (transformarlo o combinarlo con otras secuencias mediante el uso de operadores), **o suscribirse a él** (mediante el método `subscribe`).



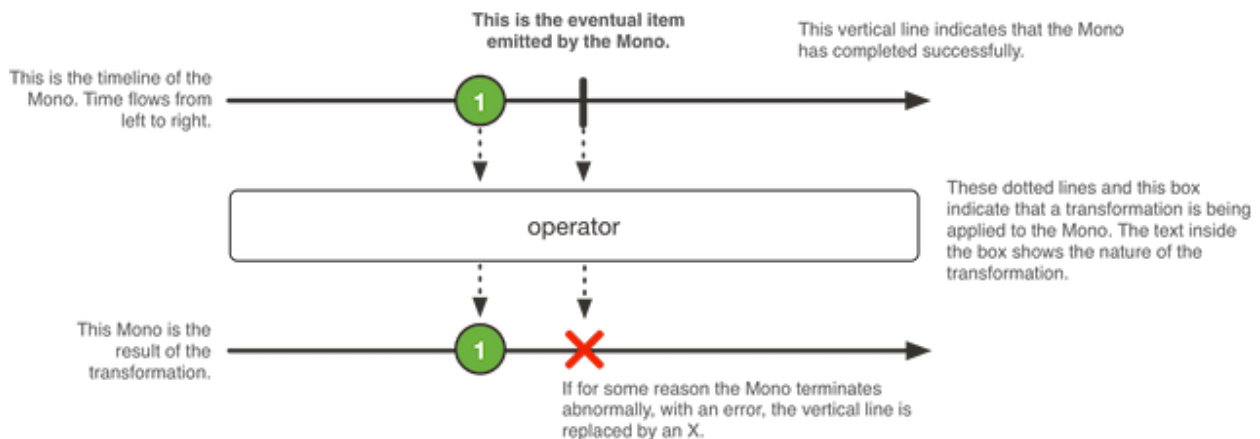
Mono

Un `Mono<T>` es un `Publisher<T>` especializado que emite como mucho un elemento y luego termina opcionalmente con una señal `onComplete` o una señal `onError`.

Es útil para los casos en los que te encuentras con una secuencia que sabes que tiene sólo uno o cero elementos. Por ejemplo, un método de repositorio que encuentra una entidad por su id. Digamos que representa un `Flux` con un valor único o vacío. La secuencia vacía es `Mono<Void>`.

`Mono` ofrece sólo un subconjunto de los operadores que están disponibles para `Flux`.

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II



Operadores

Flux proporciona muchos métodos, la mayoría de ellos [operadores](#). Dichos **operadores se usan para transformar los eventos del Flux** y además pueden concatenarse entre ellos. Por ejemplo:

```
Flux<String> flux = Flux.just("red", "white", "blue");
Flux<String> upper = flux.log().map(String::toUpperCase);
```

En el código anterior se trazan los eventos internos dentro del Flux mediante el método `.log()`, y a continuación transformamos las cadenas en la entrada convirtiéndolas en mayúsculas.

Lo que es interesante de esto es que todavía no se han procesado los datos. Nada se ha trazado ni transformado. Llamar operadores sobre Flux equivale a construir un plan de ejecución para más adelante. **La lógica implementada en los operadores solo se ejecuta cuando los datos comienzan a fluir, y eso no ocurre hasta que alguien se suscribe al Flux.**

La forma de suscribirse a un Flux es usando uno de los métodos [subscribe\(\)](#).

El método `subscribe()` está sobrecargado, y las otras variantes le dan diferentes opciones para controlar lo que sucede.

Otra forma de suscribirse a una secuencia, es llamando a los métodos `Mono.block()` o `Mono.toFuture()` o `Flux.toStream()`. Dichos métodos son **los llamados métodos "extractores"**, y **son bloqueantes**, por lo que **nos sacan del tipo Reactivo. Una buena regla general es "nunca llamar a un extractor"**. Hay algunas excepciones (de lo contrario, los métodos no existirían). Una excepción notable es en las pruebas porque es útil poder bloquear para permitir que los resultados se acumulen.

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

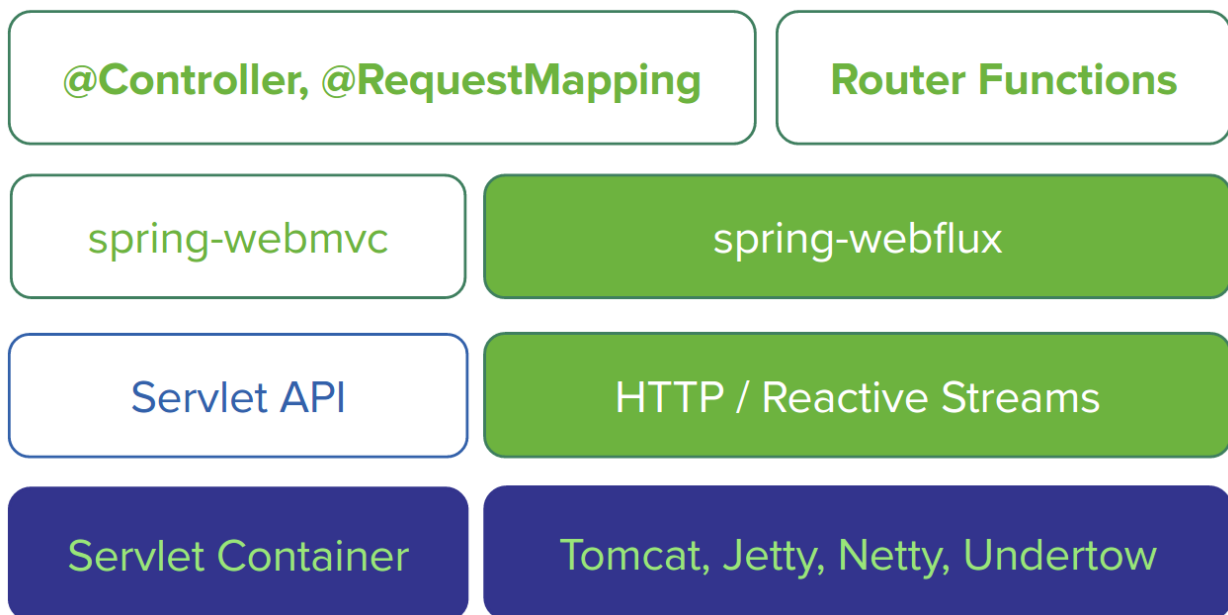
Parte servidora

WebFlux admite dos modelos de programación distintos en la parte servidora:

- **Basado en anotaciones:** mediante el uso de `@Controller` y las otras anotaciones compatibles también con Spring MVC.
- **Funcional:** enrutamiento y manejo mediante el uso de funciones lambda de Java 8.

Ambos modelos de programación se ejecutan en la misma base reactiva que adapta las librerías de ejecución HTTP no bloqueantes al API de Reactive Streams.

El siguiente diagrama muestra, en la parte de la izquierda, la pila de la parte servidora que incluye Spring MVC tradicional basado en servlets del módulo *spring-webmvc*. A la derecha muestra la pila reactiva del módulo *spring-webflux*.



WebFlux puede ejecutarse en contenedores Servlet 3.1+, así como en Netty, Undertow, Tomcat y Jetty.

Modelo de programación basado en anotaciones

El mismo modelo de programación mediante el uso de `@Controller` y las mismas anotaciones utilizadas en Spring MVC también son compatibles con WebFlux. La principal diferencia es que los contratos subyacentes básicos, es decir, `HandlerMapping`, `HandlerAdapter`, no son bloqueantes y operan usando `ServerHttpRequest` y `ServerHttpResponse` que son reactivas en lugar de `HttpServletRequest` y `HttpServletResponse`.

```
@RestController
public class PersonController {
    private final PersonRepository repository;
    public PersonController(PersonRepository repository) {
```

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

```

    this.repository = repository;
}
@PostMapping("/person")
Mono<Void> create(@RequestBody Publisher<Person> personStream) {
    return this.repository.save(personStream).then();
}
@GetMapping("/person")
Flux<Person> list() {
    return this.repository.findAll();
}
@GetMapping("/person/{id}")
Mono<Person> findById(@PathVariable String id) {
    return this.repository.findOne(id);
}
}

```

Modelo de programación funcional

Es la alternativa al modelo basado en anotaciones y se basa en el uso de los siguientes componentes:

- HandlerFunctions
- RouterFunctions

HandlerFunctions

Las peticiones entrantes HTTP son manejadas por `HandlerFunction`, que es esencialmente una función que:

- Recibe como argumento una `ServerRequest`.
- Devuelve un `Mono<ServerResponse>`.

La anotación equivalente sería un método anotado con `@RequestMapping`.

`ServerRequest` y `ServerResponse` son interfaces inmutables que ofrecen acceso amigable para JDK-8 a los mensajes HTTP subyacentes. Ambos son completamente reactivos al construir sobre Reactor: la solicitud expone el cuerpo como `Flux` o `Mono`; la respuesta acepta cualquier `Publisher` de Reactive Streams como cuerpo.

ServerRequest

`ServerRequest` proporciona acceso a varios elementos de solicitud HTTP:

- el método

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

- la URI
- los parámetros de la petición
- los encabezados (a través de la interfaz independiente `ServerRequest.Headers`).

ServerResponse

Del mismo modo, `ServerResponse` proporciona acceso a la respuesta HTTP. Como es inmutable, para crear una `ServerResponse` ha de usarse un builder. Dicho builder permite establecer el estado de la respuesta, añadir encabezados y proporcionar un cuerpo.

```
public Mono<ServerResponse> createPerson(ServerRequest request) {
    Mono<Person> person = request.bodyToMono(Person.class);
    return ServerResponse.ok().build(repository.savePerson(person));
}
```

También podemos crear una `HandlerFunction` mediante una función lambda:

```
HandlerFunction<ServerResponse> helloWorld =
    request -> ServerResponse.ok().body(fromObject("Hello World"));
```

Sin embargo, escribir handler functions mediante el uso de funciones lambda, hace que se pierda legibilidad en el código y que éste se vuelva menos mantenible cuando trabajamos con muchas funciones. Es por esto que se recomienda agrupar las handler function relacionadas en una única clase handler o controller:


```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.BodyInserters.fromObject;

public class PersonHandler {
    private final PersonRepository repository;

    public PersonHandler(PersonRepository repository) {
        this.repository = repository;
    }

    public Mono<ServerResponse> listPeople(ServerRequest request) {
        Flux<Person> people = repository.allPeople();
        return ServerResponse.ok().contentType(APPLICATION_JSON).body(people, Person.class);
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) {
        Mono<Person> person = request.bodyToMono(Person.class);
        return ServerResponse.ok().build(repository.savePerson(person));
    }
}
```

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

```

    }

    public Mono<ServerResponse> getPerson(ServerRequest request) {
        int personId = Integer.valueOf(request.pathVariable("id"));
        Mono<ServerResponse> notFound = ServerResponse.notFound().build();
        Mono<Person> personMono = this.repository.getPerson(personId);
        return personMono
            .then(person -> ServerResponse.ok().contentType(APPLICATION_JSON)
                .body(fromObject(person)))
            .otherwiseIfEmpty(notFound);
    }
}

```

RouterFunctions

Las peticiones entrantes se enrutan a las handler function con una RouterFunction. Una RouterFunction es una función que:

1. recibe una ServerRequest.
2. devuelve un Mono<HandlerFunction>.

Si una petición coincide con una ruta en particular, se devuelve una handler function; de lo contrario, devuelve un Mono vacío. La RouterFunction tiene un propósito similar al de la anotación @RequestMapping en las clases @Controller.

Para construir una router function se usa el método

```
RouterFunctions.route(RequestPredicate, HandlerFunction)
```

Dicho método crea una utilizando un request predicate y una handler function. Si se aplica el predicado, la solicitud se enruta a la handler function dada; de lo contrario, no se realiza el enrutamiento, y se devuelve una respuesta con código 404. Aunque se puede escribir nuestro propio RequestPredicate, no es necesario: la clase de utilidad RequestPredicates ofrece predicados comúnmente utilizados, tales como coincidencia basada en la ruta, el método HTTP, el tipo de contenido, etc.

Se puede crear una router function a partir de otras dos de tal forma que enrute a cualquiera de las handler function de las dos router function que la componen: si el predicado de la primera ruta no coincide, se evalúa el segundo.

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

IMPORTANTE: Las funciones compuestas del enrutador se evalúan en orden, por lo que será necesario colocar funciones específicas antes que las genéricas.

Se puede componer dos router function llamando a los métodos:

- `RouterFunction.and(RouterFunction)`
- `RouterFunction.andRoute(RequestPredicate, HandlerFunction)`

Un ejemplo podría ser el siguiente:

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.RequestPredicates.*;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> personRoute =
    route(GET("/person/{id}").and(accept(APPLICATION_JSON)), handler::getPerson)
    .andRoute(GET("/person").and(accept(APPLICATION_JSON)), handler::listPeople)
    .andRoute(POST("/person").and(contentType(APPLICATION_JSON)), handler::createPerson);
```

Además de las router function, también puede componer request predicates, llamando a los métodos:

- `RequestPredicate.and(RequestPredicate)`: el predicado resultante coincide si ambos predicados dados coinciden.
- `RequestPredicate.or(RequestPredicate)`: coincide si cualquier predicado lo hace.

La mayoría de los predicados encontrados en `RequestPredicates` son composiciones. Por ejemplo, `RequestPredicates.GET(String)` es una composición de `RequestPredicates.method(HttpMethod)` y `RequestPredicates.path(String)`.

¿Cómo ejecutar una RouterFunction?

Para ejecutar una router function en un servidor HTTP basta con usar el método

```
RouterFunctions.toHttpHandler(RouterFunction)
```

Dicho método:

- Recibe una `RouterFunction` como argumento.
- Devuelve un `HttpHandler`.

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

`HttpHandler` le permite ejecutar en una amplia variedad de librerías de ejecución reactivas: Reactor Netty, RxNetty, Servlet 3.1+ y Undertow.

Ejemplo en Reactor Netty:

```
RouterFunction<ServerResponse> route = ...
HttpHandler httpHandler = RouterFunctions.toHttpHandler(route);
ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(httpHandler);
HttpServer server = HttpServer.create(HOST, PORT);
server.newHandler(adapter).block();
```

Ejemplo en Tomcat:

```
RouterFunction<ServerResponse> route = ...
HttpHandler httpHandler = RouterFunctions.toHttpHandler(route);
HttpServlet servlet = new ServletHttpHandlerAdapter(httpHandler);
Tomcat server = new Tomcat();
Context rootContext = server.addContext("", System.getProperty("java.io.tmpdir"));
Tomcat.addServlet(rootContext, "servlet", servlet);
rootContext.addServletMapping("/", "servlet");
tomcatServer.start();
```

HandlerFilter

Las rutas mapeadas por una router function se pueden filtrar llamando al método

```
RouterFunction.filter(HandlerFilterFunction)
```

donde `HandlerFilterFunction` es esencialmente una función que:

- Recibe una `ServerRequest` y una `HandlerFunction` como argumentos.
- Devuelve una `ServerResponse`.

El parámetro handler function representa el siguiente elemento de la cadena: por lo general, esta es la `HandlerFunction` que se enruta, pero también puede ser otra `FilterFunction` si se aplican múltiples filtros. Con las anotaciones, se puede lograr una funcionalidad similar usando `@ControllerAdvice` y/o un `ServletFilter`. Por ejemplo:

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

```
RouterFunction<ServerResponse> route = ...

RouterFunction<ServerResponse> filteredRoute =
    route.filter(request, next) -> {
        log.debug("Requested {} {}", request.method(), request.path());
        return next.handle(request);
    });
```

Parte cliente

WebFlux proporciona un `WebClient` funcional y reactivo que ofrece una alternativa a `RestTemplate` totalmente reactiva y no bloqueante. Expone la entrada y salida de red como `ClientHttpRequest` y `ClientHttpResponse` reactivos, donde el cuerpo de la solicitud y respuesta es un `Flux<DataBuffer>` en lugar de un `InputStream` y `OutputStream`. Además, admite el mismo mecanismo de serialización reactiva JSON, XML y SSE que en la parte servidora para que pueda trabajar con objetos tipados.

Ejemplo:

```
WebClient client = WebClient.create("http://example.com");

Mono<Account> account = client.get()
    .url("/accounts/{id}", 1L)
    .accept(APPLICATION_JSON)
    .exchange(request)
    .then(response -> response.bodyToMono(Account.class));
```

Soporte a Reactive WebSocket

WebSocket

Introducción

Desde el principio de su existencia, la Web se ha construido alrededor del paradigma petición/respuesta de HTTP: el usuario carga una página y no ocurre nada hasta que éste accede a la siguiente. A partir del año

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

2005, AJAX empezó a modificar este paradigma añadiendo la posibilidad de, una vez cargada la página, realizar peticiones para obtener información adicional del servidor, ya sea de forma periódica o debido a la interacción del usuario. La principal característica de AJAX es que el servidor no puede iniciar una comunicación con el cliente, ya que es este último quien siempre toma la iniciativa.

En oposición a estas tecnologías (conocidas como *pull*), en las que el cliente realiza la petición de envío, existen las conocidas como *push*, que permiten a los servidores enviar información al cliente en cualquier momento; normalmente cuando tienen nueva información disponible.

El protocolo WebSocket plantea un modelo elegante y sencillo de comunicaciones para la Web que no rompe con las tecnologías ya existentes.

Compañías de éxito como [Slack](#) (mensajería corporativa), [Trello](#) (gestión de proyectos) o [WhatsApp](#) (mensajería personal) utilizan WebSocket para ofrecer sus servicios.

¿Qué es?

WebSocket es un protocolo que permite crear un canal de comunicación bidireccional sobre una sola conexión TCP.

Está pensado para ser implementado en navegadores y servidores web, aunque no hay ningún impedimento a la hora de implementarlo en cualquier otro tipo de aplicación que siga el modelo cliente/servidor.

Las comunicaciones se realizan a través de los mismos puertos que utiliza HTTP con el fin de ofrecer compatibilidad con el software HTTP del lado del servidor ya existente. Es decir, cuando el protocolo trabaja directamente sobre TCP utiliza el puerto 80 y cuando lo hace sobre TLS utiliza el 443. No obstante, WebSocket es un protocolo independiente.

WebSocketClient

WebFlux incluye soporte reactivo de cliente y servidor WebSocket. Tanto el cliente como el servidor son soportados por Java WebSocket API (JSR-356), Jetty, Undertow, Reactor Netty y RxNetty.

En el lado del servidor, se declara un `WebSocketHandlerAdapter` y luego simplemente se añade las asignaciones a los endpoints basados en `WebSocketHandler`:

```
@Bean
public HandlerMapping websocketMapping() {
    Map<String, WebSocketHandler> map = new HashMap<>();
    map.put("/foo", new FooWebSocketHandler());
    map.put("/bar", new BarWebSocketHandler());

    SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
    mapping.setUrlMap(map);
    return mapping;
}
```

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

```

}

@Bean
public WebSocketHandlerAdapter handlerAdapter() {
    return new WebSocketHandlerAdapter();
}

```

En el lado del cliente, se crea un `WebSocketClient` para una de las librerías compatibles listadas arriba. Por ejemplo, para Netty:

```

WebSocketClient client = new ReactorNettyWebSocketClient();
client.execute("ws://localhost:8080/echo", session -> {... }).blockMillis(5000);

```

Testing

WebTestClient

El módulo *spring-test* incluye `WebTestClient` que puede ser usado para probar endpoints de un servidor WebFlux con o sin un servidor corriendo.

Las pruebas sin un servidor en ejecución son comparables `MockMvc` de Spring MVC donde se utilizan la solicitud y la respuesta simuladas en lugar de conectarse a través de la red con un socket. Sin embargo, `WebTestClient` también puede realizar pruebas contra un servidor en ejecución.

StepVerifier

Reactor nos proporciona el API `StepVerifier` para poder probar el funcionamiento de nuestras secuencias (`Flux` o `Mono`) y ver cómo se comportarían cuando nos suscribamos a ellas.

`StepVerifier` proporciona una forma declarativa de crear un script verificable para una secuencia de `Publisher` asíncrono, expresando expectativas que sucederán sobre los eventos con la suscripción. La verificación debe activarse después de que se hayan declarado las expectativas del terminal (finalización, error, cancelación), llamando a uno de los métodos `verify()`.

Los pasos a seguir serían:

	Empresa	Título	Autor
	BI Geek S.L.	Spring WebFlux	Álvaro Martín Consultor Senior II

1. Crear un `StepVerifier` en torno a un `Publisher` utilizando `create(Publisher)` o con `VirtualTime(Supplier <Publisher>)` (en cuyo caso debe crear el `Publisher` de forma perezosa dentro de la lambda proporcionada).
2. Configurar las expectativas de valores individuales utilizando `expectNext`, `expectNextMatches(Predicate)`, `assertNext(Consumer)`, `expectNextCount(long)` o `expectNextSequence(Iterable)`.
3. Desencadenar acciones de suscripción durante la verificación utilizando `thenRequest(long)` o `thenCancel()`.
4. Finalizar el escenario de prueba utilizando una expectativa terminal: `expectComplete()`, `expectError()`, `expectError(Class)`, `expectErrorMatches(Predicate)` o `thenCancel()`.
5. Lanzar la verificación del `StepVerifier` resultante en su `Publisher` utilizando `verify()` o `verify(Duration)`. Se ha de tener en cuenta que algunas de las anteriores expectativas terminales tienen una alternativa prefijada de "verificación" que declara la expectativa y activa la verificación de forma conjunta.
6. Si alguna de las expectativas falla, se lanzará un `AssertionError` que indica los errores.

Por ejemplo:

```
StepVerifier.create(Flux.just("foo", "bar"))
    .expectNext("foo")
    .expectNext("bar")
    .expectComplete()
    .verify();
```

Referencias

- Spring:

<https://docs.spring.io/spring/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/web-reactive.html>

<https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html>

- Reactive Streams:

<http://www.reactive-streams.org/>

<https://github.com/reactive-streams/reactive-streams-jvm#reactive-streams>

- Reactor

<https://projectreactor.io/docs/core/release/reference/>