Germán Caballero Rodríguez
gcaballero@gmail.com

INDICE

1) Introducción
2) Manifiesto reactivo
3) Programación Reactiva
4) Patrones de diseño implicados
5) APIs Java de programación reactiva
6) Arquitectura Spring Webflux
7) Primer Ejemplo WebFlux
8) HTTP asíncrono con Spring Web MVC
9) Enlaces de interés

Introducción

- Organizaciones que trabajan en dominios diferentes están descubriendo de manera independiente patrones similares para construir software.
- Estos sistemas son más robustos, más flexibles y están mejor posicionados para cumplir demandas modernas.
- Estos cambios están sucediendo porque los requerimientos de las aplicaciones han cambiado drásticamente en los últimos años.

Introducción

Antes con Wall-E nos hubieramos conformado…

¿Pero ahora qué se requiere, qué piden, qué exige la demanda?

Introducción

Introducción

Introducción

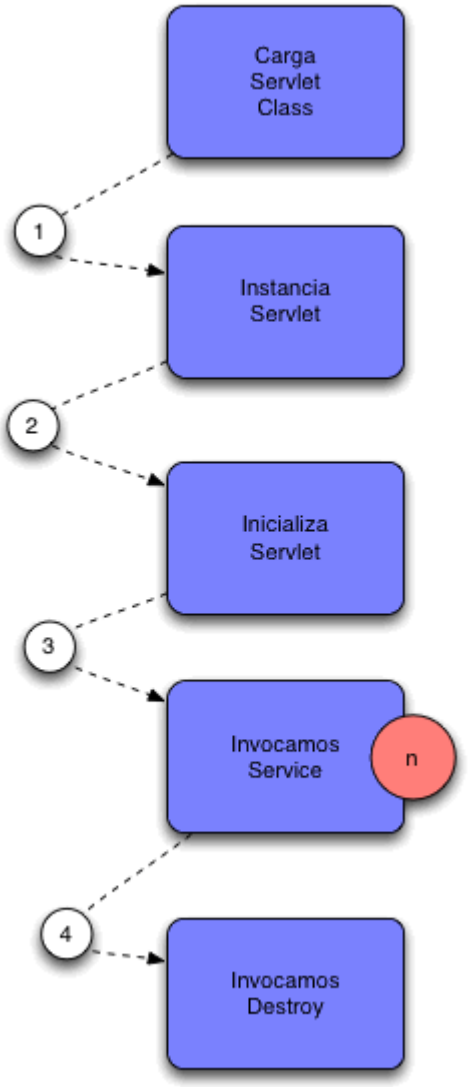
- Sólo unos pocos años atrás, una aplicación grande tenía decenas de servidores, segundos de tiempo de respuesta, horas de mantenimiento fuera de línea y gigabytes en datos.
- Hoy, las aplicaciones se despliegan en cualquier cosa, desde dispositivos móviles hasta clusters en la nube corriendo en miles de procesadores multi-core.

Introducción

- Los usuarios esperan que los tiempos de respuesta sean de milisegundos y que sus sistemas estén operativos el 100% del tiempo.
- Los datos son medidos en Petabytes.
- Las demandas de hoy simplemente no están siendo satisfechas por las arquitecturas software de ayer.
- Repasemos un poco...

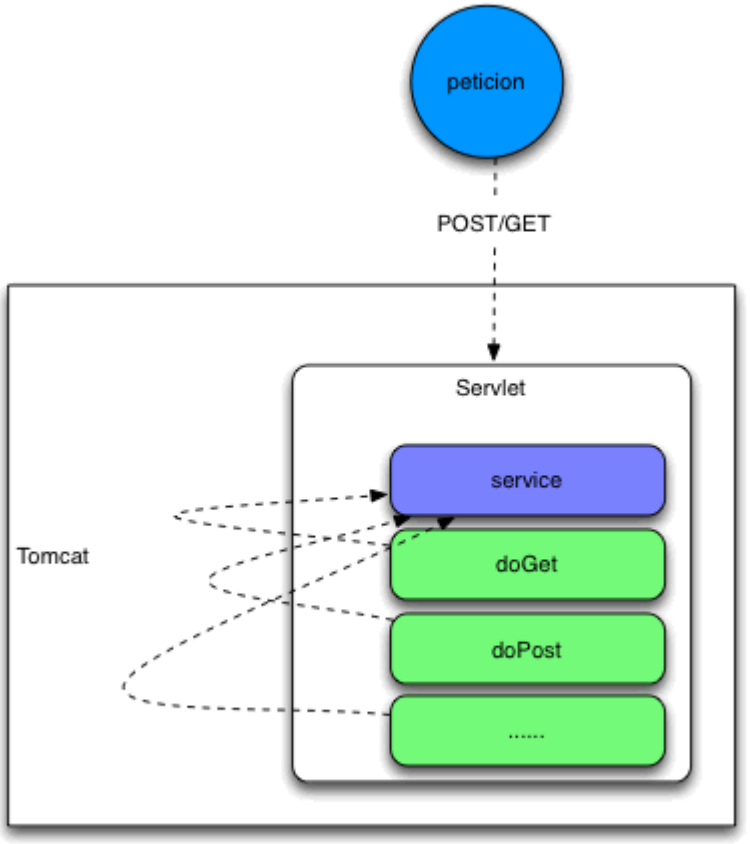
Introducción

- Entendiendo el Servlet Lifecycle en Java EE:
- Cargar el Servlet
- Instanciar el Servlet
- Invocar init()
- Invocar service():
  - El cuarto paso es el más habitual que será invocar repetidamente al método service() a través de doGet() o doPost() para que el Servlet ejecute la funcionalidad solicitada.
- Invocar destroy():



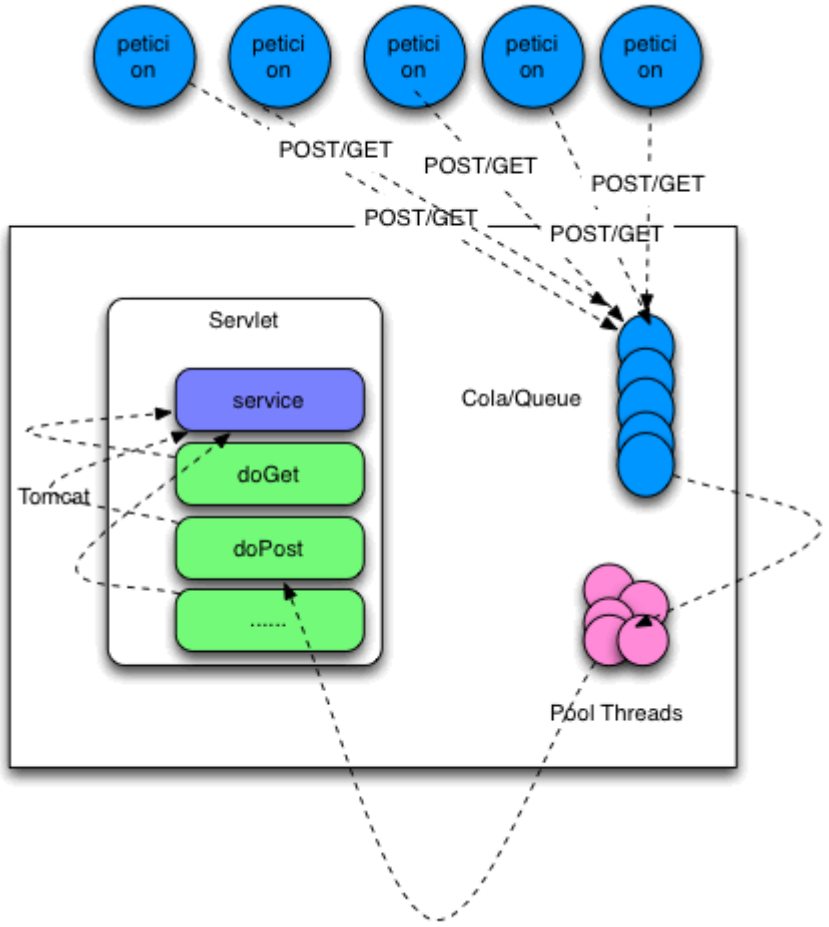
Introducción

Habitualmente un contenedor de Servlets recibe peticiones HTTP para acceder a un Servlet concreto por lo tanto instancia el Servlet y lo pone a disposición de cada una de las peticiones a través del método service() al cual doPost() doGet() etc delegan.



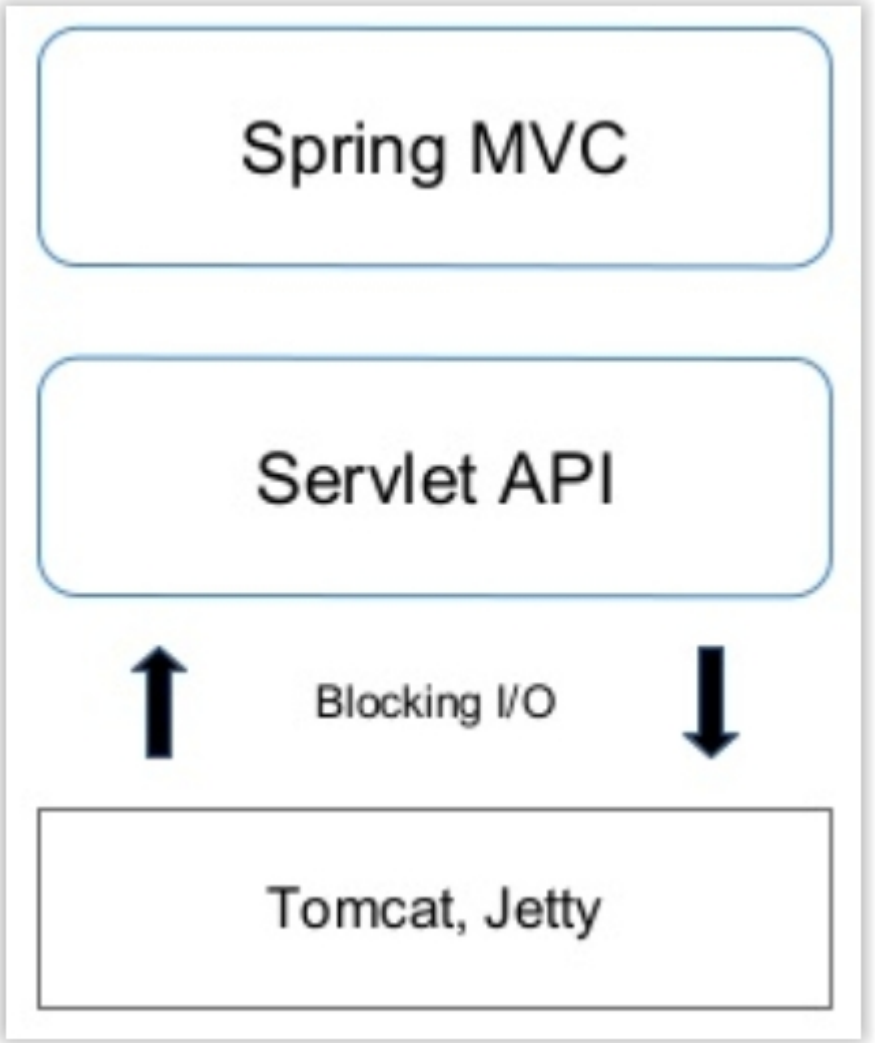
Introducción

- Para realizar esta operación el Servlet Container usa un pool de Threads que se encarga de gestionar las peticiones que han sido encoladas.
- De esta forma es como un Servlet Container habitualmente trabaja con un Servlet a la hora de gestionar las peticiones que le llegan.
- Por ello los Servlets por defecto no son Thread Safe.



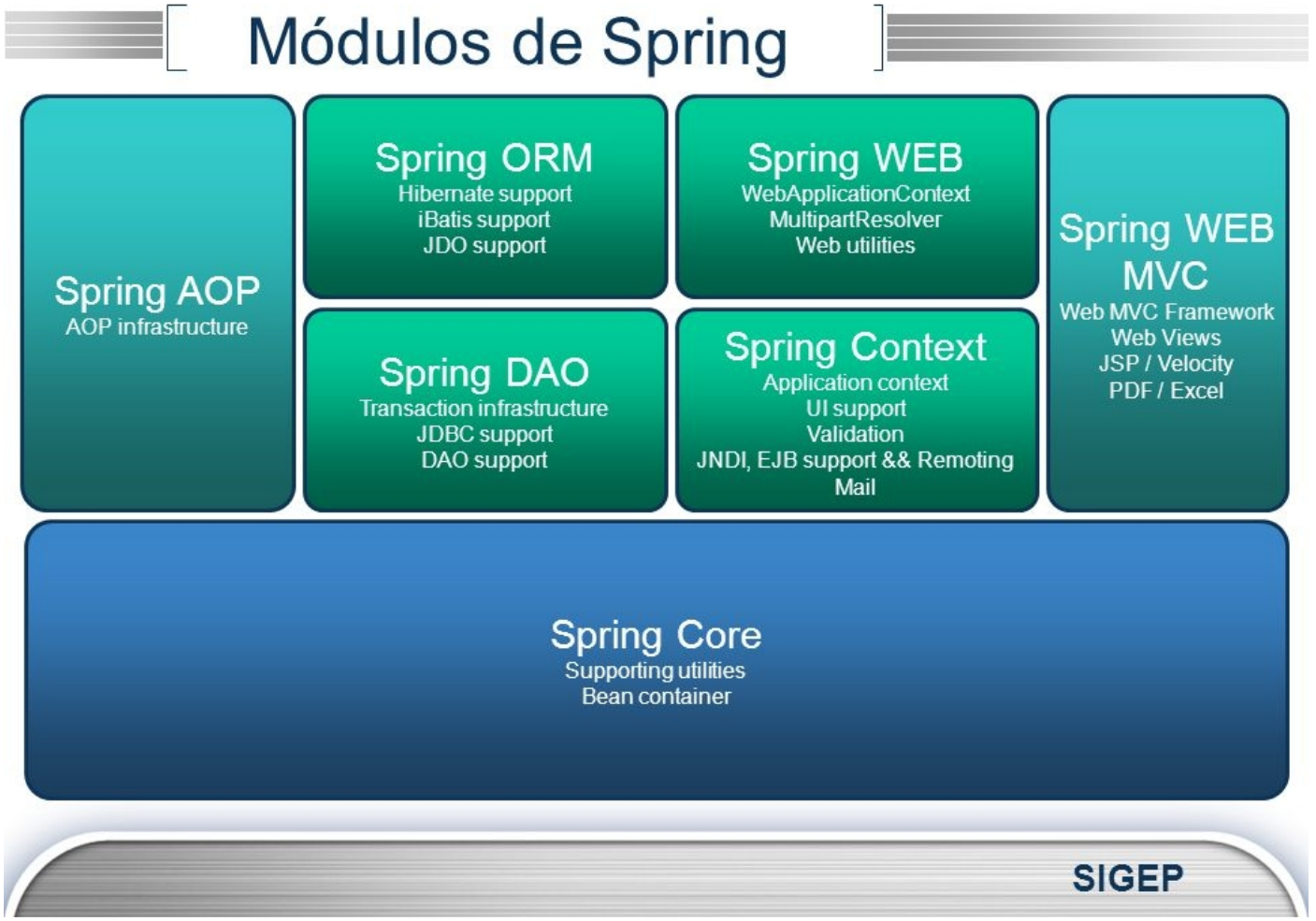
Introducción

- Arquitectura Spring MVC bloqueante



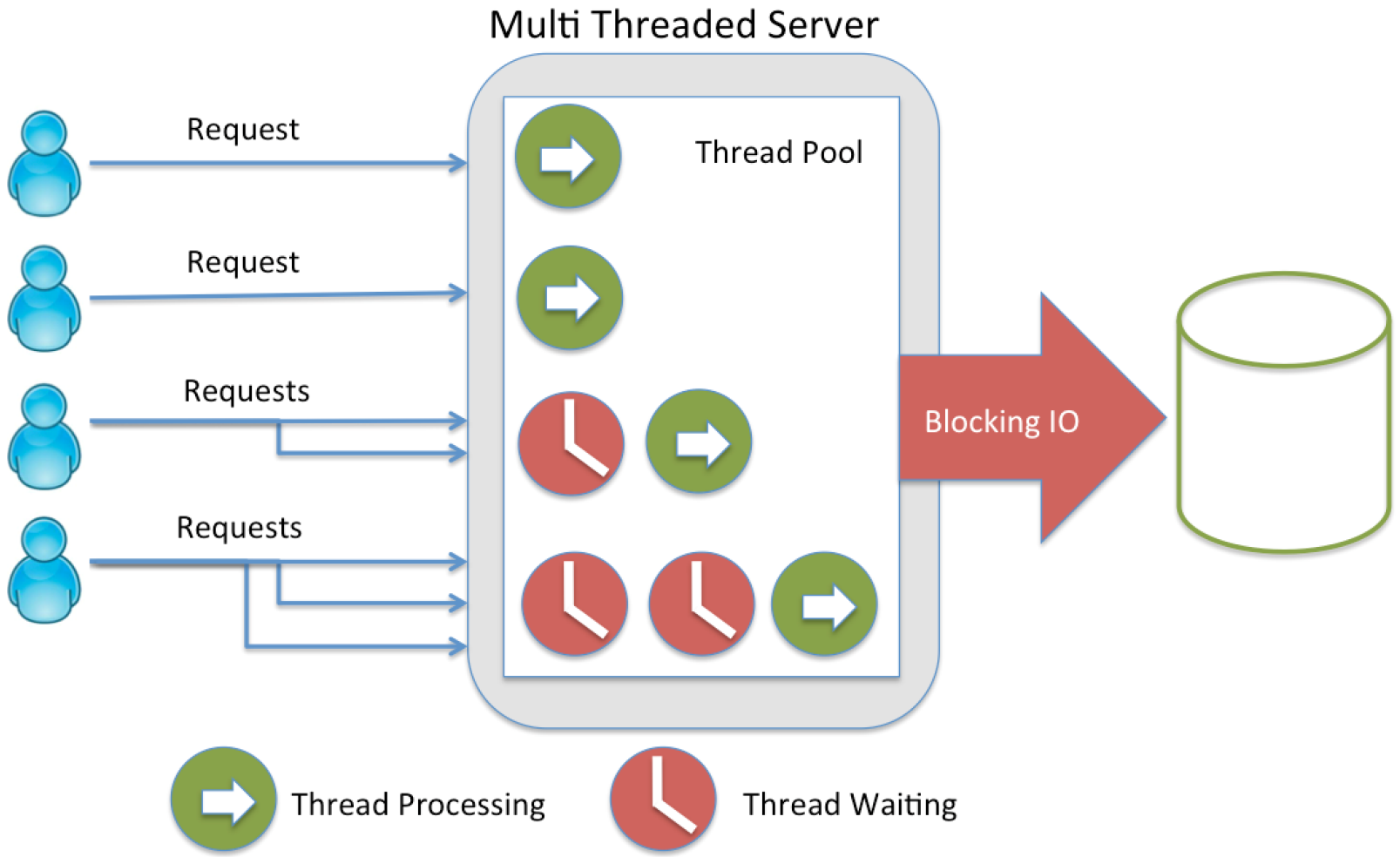
Introducción

- Arquitectura Spring MVC bloqueante



Introducción

- Arquitectura Spring MVC bloqueante

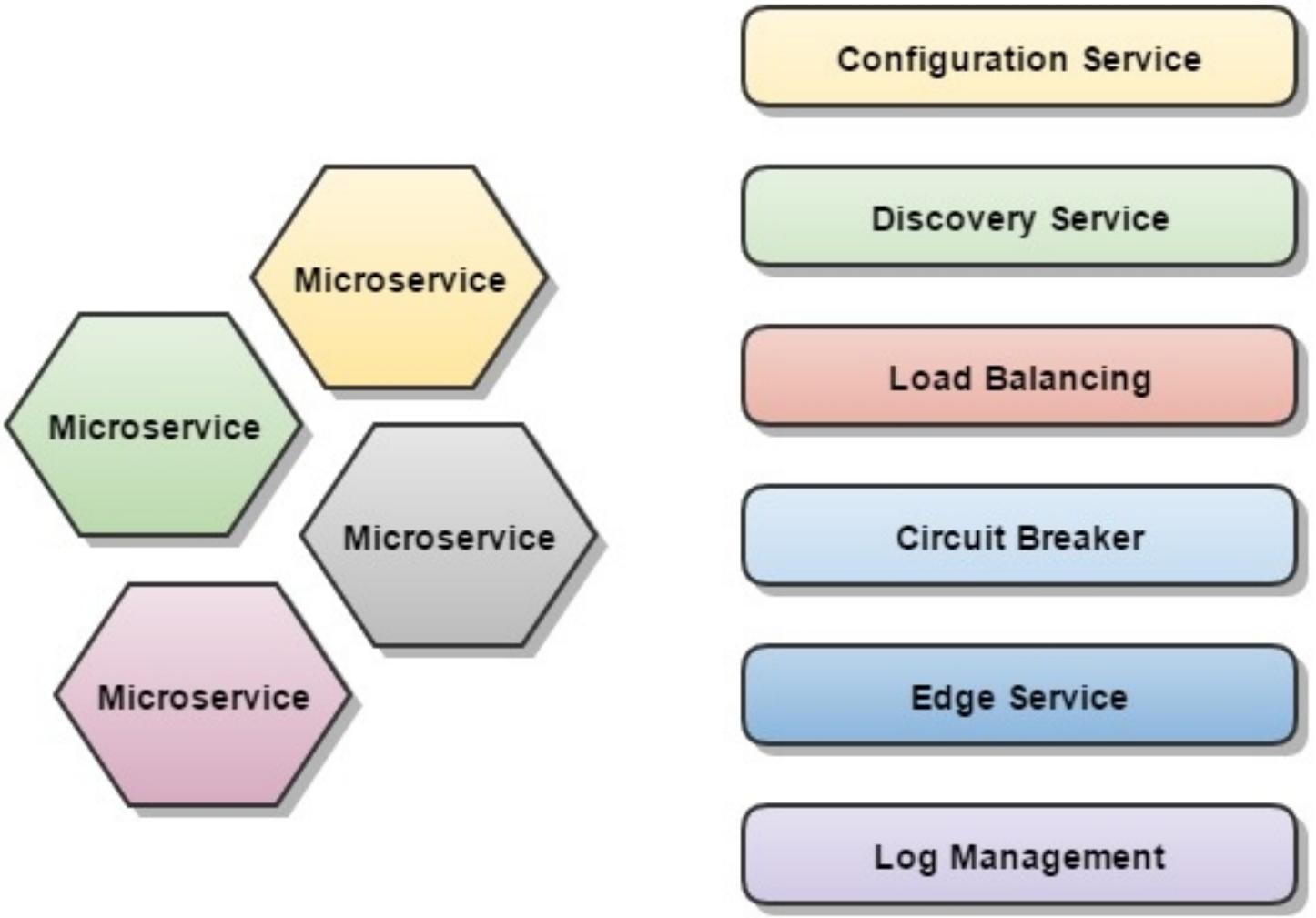


Introducción

- Se necesita un enfoque coherente para la arquitectura de sistemas, y los principales aspectos necesarios para los sistemas ya han sido identificados por separado:
  - Responsivos
  - Resilientes
  - Elásticos
  - Orientados a Mensajes.
- Se les conoce por Sistemas Reactivos, y deben cumplir los anteriores puntos

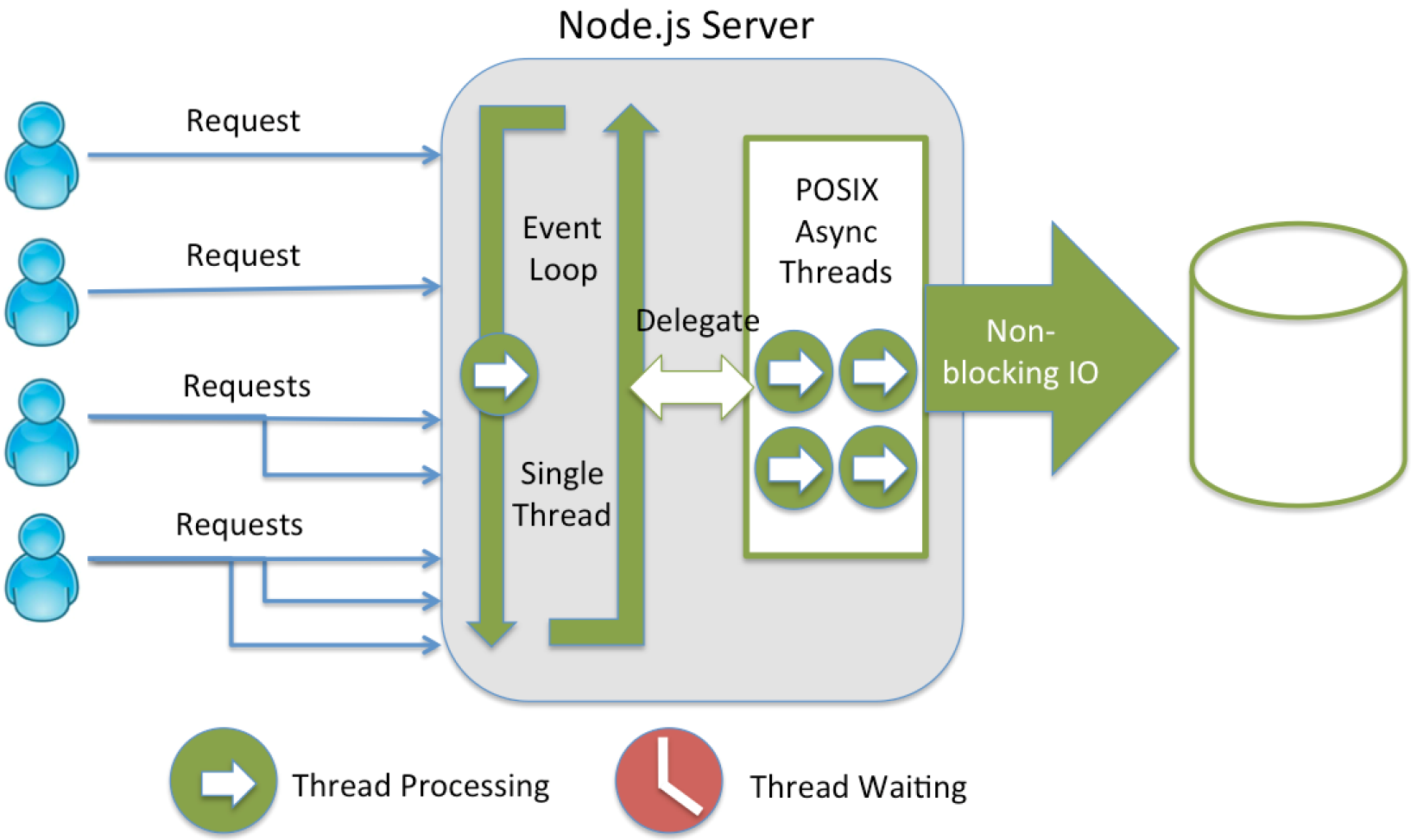
Introducción

- Arquitectura Microservicios



Introducción

- Tendencias similares: NodeJS



Introducción

- Los sistemas construidos como Sistemas Reactivos son más flexibles, con bajo acoplamiento y escalables.
- Esto hace que sean más fáciles de desarrollar y abiertos al cambio.
- Son significativamente más tolerantes a fallos y cuando fallan responden con elegancia y no con un desastre.
- Los Sistemas Reactivos son altamente responsivos, dando a los usuarios un feedback efectivo e interactivo.

Manifiesto reactivo

Según el manifiesto los sistemas deben ser:

- **Responsivos**: aseguran la calidad del servicio cumpliendo unos tiempos de respuesta establecidos.
- **Resilientes**: se mantienen responsivos incluso cuando se enfrentan a situaciones de error.
- **Elásticos**: se mantienen responsivos incluso ante aumentos en la carga de trabajo.
- **Orientados a mensajes**: minimizan el acoplamiento entre componentes al establecer interacciones basadas en el intercambio de mensajes de manera asíncrona.

Manifiesto reactivo

Responsivos:

- El sistema **responde a tiempo** en la medida de lo posible.
- La responsividad es la piedra angular de la usabilidad y la utilidad, pero más que esto, responsividad significa que los problemas pueden ser **detectados rápidamente y tratados efectivamente**.
- Este comportamiento consistente, a su vez, simplifica el tratamiento de errores, aporta seguridad al usuario final y fomenta una mayor interacción.

Manifiesto reactivo

Resilientes:

- El sistema permanece responsivo frente a fallos.
- Esto es aplicable no sólo a sistemas de alta disponibilidad o de misión crítica - cualquier sistema que no sea resiliente dejará de ser responsivo después de un fallo.
- La resiliencia es alcanzada con replicación, contención, aislamiento y delegación.
- Los fallos son manejados dentro de cada componente.
- La recuperación de cada componente se delega en otro componente (externo).

Manifiesto reactivo

Elásticos:

- El sistema se mantiene responsivo bajo variaciones en la carga de trabajo.
- Los Sistemas Reactivos pueden reaccionar a cambios en la frecuencia de peticiones incrementando o reduciendo los recursos asignados para servir dichas peticiones.
- Esto implica diseños que no tengan puntos de contención o cuellos de botella centralizados, resultando en la capacidad de dividir o replicar componentes y distribuir las peticiones entre ellos.

Manifiesto reactivo

Orientados a Mensajes:

- Los Sistemas Reactivos confían en el intercambio de mensajes asíncrono para establecer fronteras entre componentes, lo que asegura bajo acoplamiento, aislamiento y transparencia de ubicación.
- Estas fronteras también proporcionan los medios para delegar fallos como mensajes.
- El uso del intercambio de mensajes explícito posibilita la gestión de la carga, la elasticidad, y el control de flujo, gracias al modelado y monitorización de las colas de mensajes en el sistema, y la aplicación de back-pressure cuando sea necesario.

Manifiesto reactivo

Orientados a Mensajes:

¿Y Back-Pressure qué es?

- En el mundo del software, la "contrapresión" es una analogía tomada de la dinámica de los fluidos, como en los gases de escape de los automóviles y las tuberías.

  Resistencia o fuerza opuesta al flujo deseado de fluido a través de tuberías.

- En el contexto del software, la definición podría ajustarse para referirse al flujo de datos dentro del software:

  Resistencia o fuerza que se opone al flujo de datos deseado a través del software .

- El propósito del software es tomar los datos de entrada y convertirlos en algunos datos de salida deseados.
- Los datos de salida pueden ser JSON de una API, pueden ser HTML para una página web o los píxeles que se muestran en su monitor.

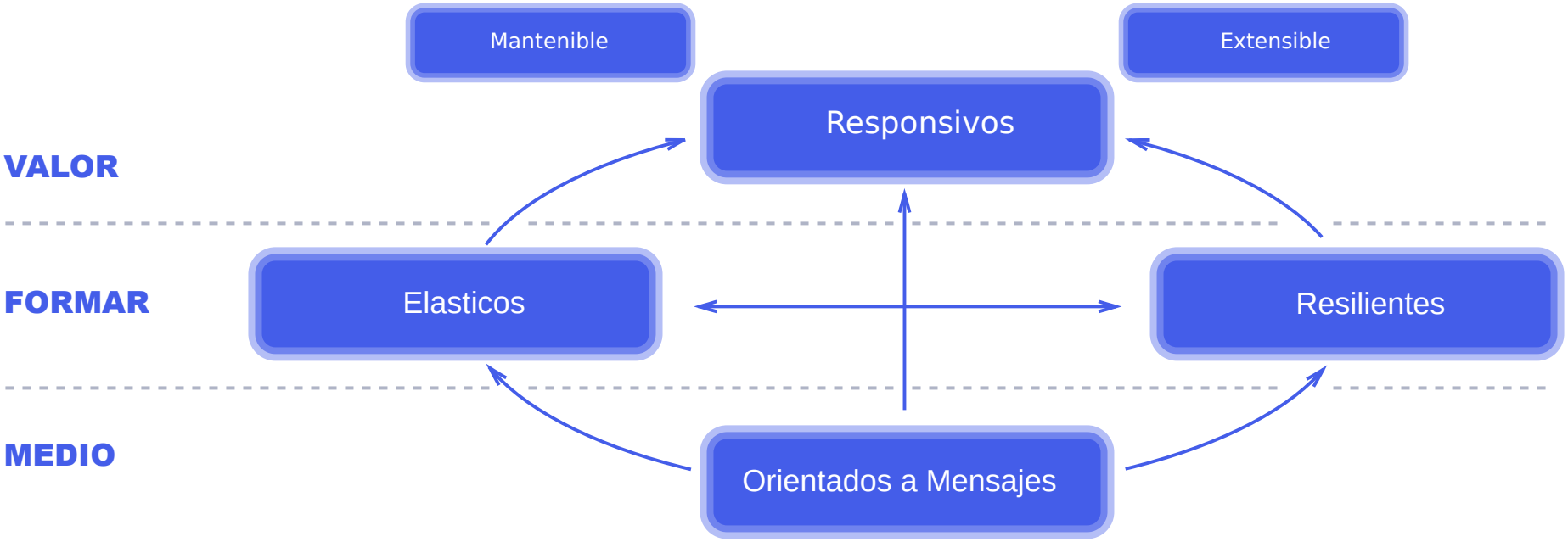
Manifiesto reactivo

Orientados a Mensajes:

¿Y Back-Pressure qué es?

- La contrapresión es cuando el progreso de convertir esa entrada en salida se resiste de alguna manera.
- En la mayoría de los casos, la resistencia es la velocidad computacional, es decir, la dificultad para calcular la salida tan rápido como la entrada, por lo que es la forma más fácil de verla.
- Pero también pueden ocurrir otras formas de contrapresión: por ejemplo, si su software tiene que esperar a que el usuario tome alguna acción.
- La comunicación No-bloqueante permite a los destinatarios consumir recursos sólo mientras estén activos, llevando a una menor sobrecarga del sistema.

Manifiesto reactivo



- Los sistemas grandes están compuestos de otros más pequeños y por lo tanto dependen de las propiedades Reactivas de sus partes.
- Esto significa que los Sistemas Reactivos aplican principios de diseño para que estas propiedades sean válidas a cualquier escala, haciéndolas componibles.
- Los sistemas más grandes del mundo confían en arquitecturas basadas en estas propiedades y atienden las necesidades de miles de millones de personas diariamente.

Programación reactiva

¿¿Pero qué es la programación reactiva??

- La programación reactiva es un **paradigma** enfocado en el trabajo con streams finitos o infinitos de manera asíncrona:
  - Paradigma: define un marco de trabajo o forma de programar.
  - Streams: flujo de datos.
  - Datos asíncronos: no sabemos cuándo se producirán o cuándo llegarán a nuestro sistema.

Programación reactiva

- Esto parece sencillo de enunciar, sin embargo, ¿por qué es tan difícil de entender?
-
-
-

- Esto se debe a que mezcla diversos conceptos y patrones no tan sencillos:
  - Programación funcional.
  - Patrón observer (más info, aquí).
  - Patrón iterator (más info, aquí).

Programación reactiva

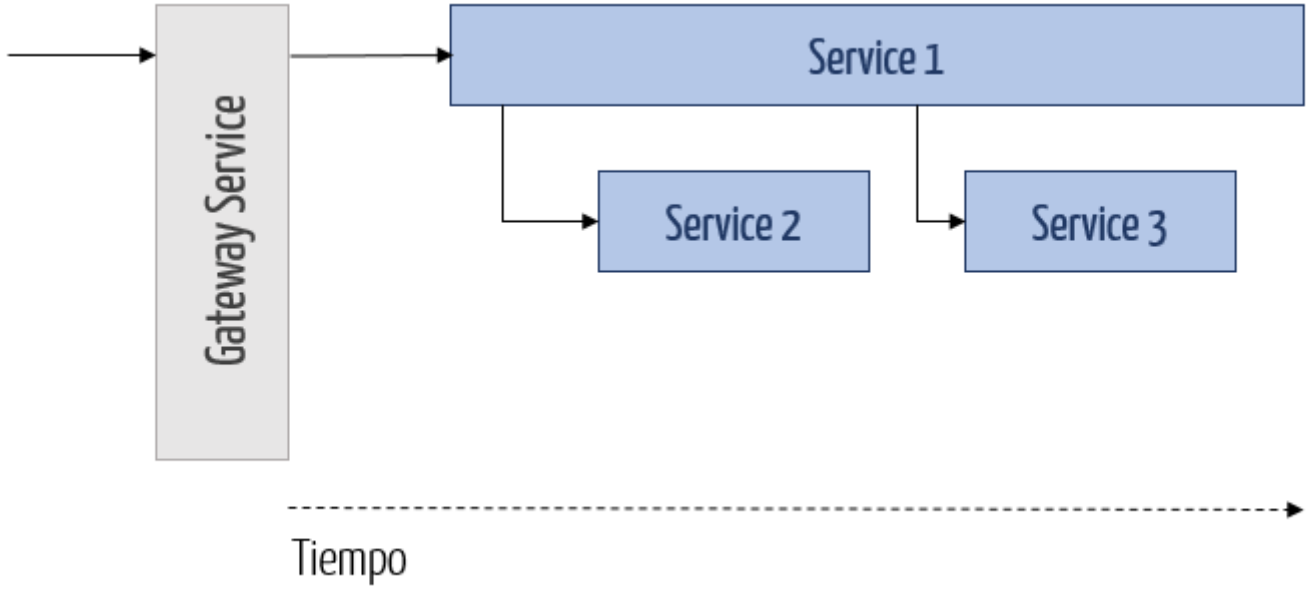
- La reactividad, al ser un paradigma (como la programación funcional), puede ser aplicada a más de un lenguaje.
- Tanto es así, que se han creado librerías para facilitar el desarrollo (y la vida de los desarrolladores), ahorrando tiempo, aumentando la productividad y construyendo sistemas más robustos y escalables.

Programación reactiva

- Su concepción y evolución ha ido ligada a la publicación del Reactive Manifesto.
- La motivación detrás de este nuevo paradigma procede de 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.

Programación reactiva

- El diagrama siguiente muestra una interacción típica entre microservicios: Service1 tiene que invocar dos servicios Service2 y Service3 independientes.
- La programación actual no permite la paralelización de las peticiones ni la composición de los resultados de una manera sencilla y eficiente.



Programación reactiva

El modelo de programación reactiva ha evolucionado de manera significativa desde su concepción en 2010. Las librerías que lo implementan se clasifican en generaciones de acuerdo a su grado de madurez:

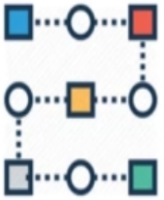
- Generación 0: java.util.Observable/Observer proporcionaban la base de uso del patrón Observer del Gang of Four. Tienen los inconvenientes de su simplicidad y falta de opciones de composición.

Programación reactiva

- 1ª Generación: en 2010 Microsoft publica RX.NET, que en 2013 sería portado a entorno Java a través de la librería RxJava.
- 2ª Generación: se solucionan los problemas de backpressure y se introducen dos nuevas interfaces: Subscriber y Producer.
- 3ª y 4ª Generación: se caracterizan principalmente por haber eliminado la incompatibilidad entre las múltiples librerías del ecosistema reactive a través de la adopción de la especificación Reactive Streams, que fija dos clases base Publisher y Subscriber. Entran dentro de esta generación proyectos como RxJava 2.x, **Project Reactor** y Akka Streams.

Programación reactiva

Crea fácilmente eventos streams o flujos de datos continuos.



Transforma y compone flujos con los operadores map, fliter, merge, delay, foreach etc.

Suscribir a cualquier flujo observable para realizar alguna tarea.

Multi-plataforma Java, JavaScript, Scala, C#, C ++, Python, PHP y otros

Programación reactiva

El uso de Reactive Streams es similar al del patrón Iterator (incluyendo Java 8 Streams) con una clara diferencia, el primero es push-based mientras que el segundo es pull-based:

Evento	Iterable (push)	Reactive (pull)
Obtener dato	next()	onNext(Object data)
Error	throws Exception	onError(Exception)
Fin	!hasNext()	onComplete()

- Iterable delega en el desarrollador las llamadas para obtener los siguientes elementos.
- Por contra, los Publisher de Reactive Streams son los encargados de notificar al Subscriber la llegada de nuevos elementos de la secuencia.

Programación reactiva

Adicionalmente las librerías reactivas se han ocupado de mejorar los siguientes aspectos:

- **Composición y legibilidad:**
  - Hasta el momento las única manera de trabajar con operaciones asíncronas en entorno Java consistía en el uso de Future, callbacks, o, desde Java 8, CompletableFuture.
  - Todas ellas presentan el gran inconveniente de dificultar la comprensión del código y la composición de operaciones, pudiendo fácilmente degenerar hacia un callback hell.
- **Operadores**:
  - Permiten aplicar transformaciones sobre los flujos de datos.
  - Si bien no forman parte de la especificación Reactive Streams, todas las librerías que la implementan los soportan de manera completamente compatible.

Programación reactiva

Adicionalmente las librerías reactivas se han ocupado de mejorar los siguientes aspectos:

- **Backpressure**: debido al flujo push-based, se pueden dar situaciones donde un Publisher genere más elementos de los que un Subscriber puede consumir. Para evitarlo se han establecido los siguientes mecanismos:
  - Los Subscriber pueden indicar el número de datos que quieren o pueden procesar mediante la operación subscriber.request(n), de manera que el Publisher nunca les enviará más de n elementos.
  - Los Publisher pueden aplicar diferentes operaciones para evitar saturar a los subscriptores lentos (buffers, descarte de datos, etc.).

Patrones de diseño implicados

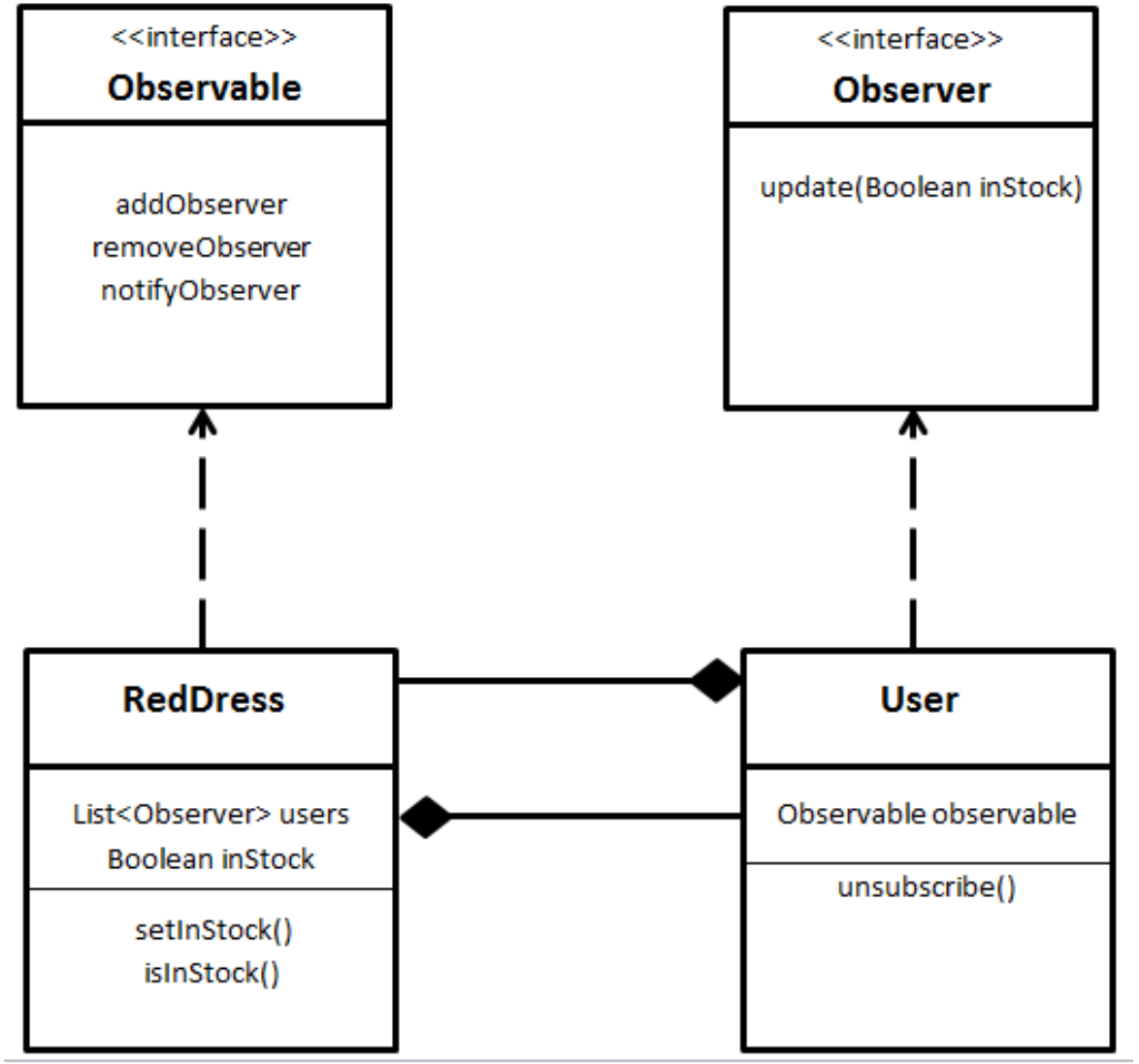
- La proramación reactiva toma lo mejor de patrones como el patrón observer, el patrón iterador, la arquitectura en pipeline y paradigmas como la programación funcional.
- Vamos a trabajar un poco con estos patrones para entenderlos bien, poder aplicarlos en otras situaciones antes de entrar de lleno en Spring WebFlux y sobre todo, llegar a entender bien la programación reactiva

Patrones de diseño implicados

- **Patrón Obvserver**
  - Es un patrón de diseño de software que define una dependencia del tipo uno a muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes.
  - Se trata de un patrón de comportamiento (existen de tres tipos: creación, estructurales y de comportamiento), por lo que está relacionado con algoritmos de funcionamiento y asignación de responsabilidades a clases y objetos.

Patrones de diseño implicados

- **Patrón Obvserver**



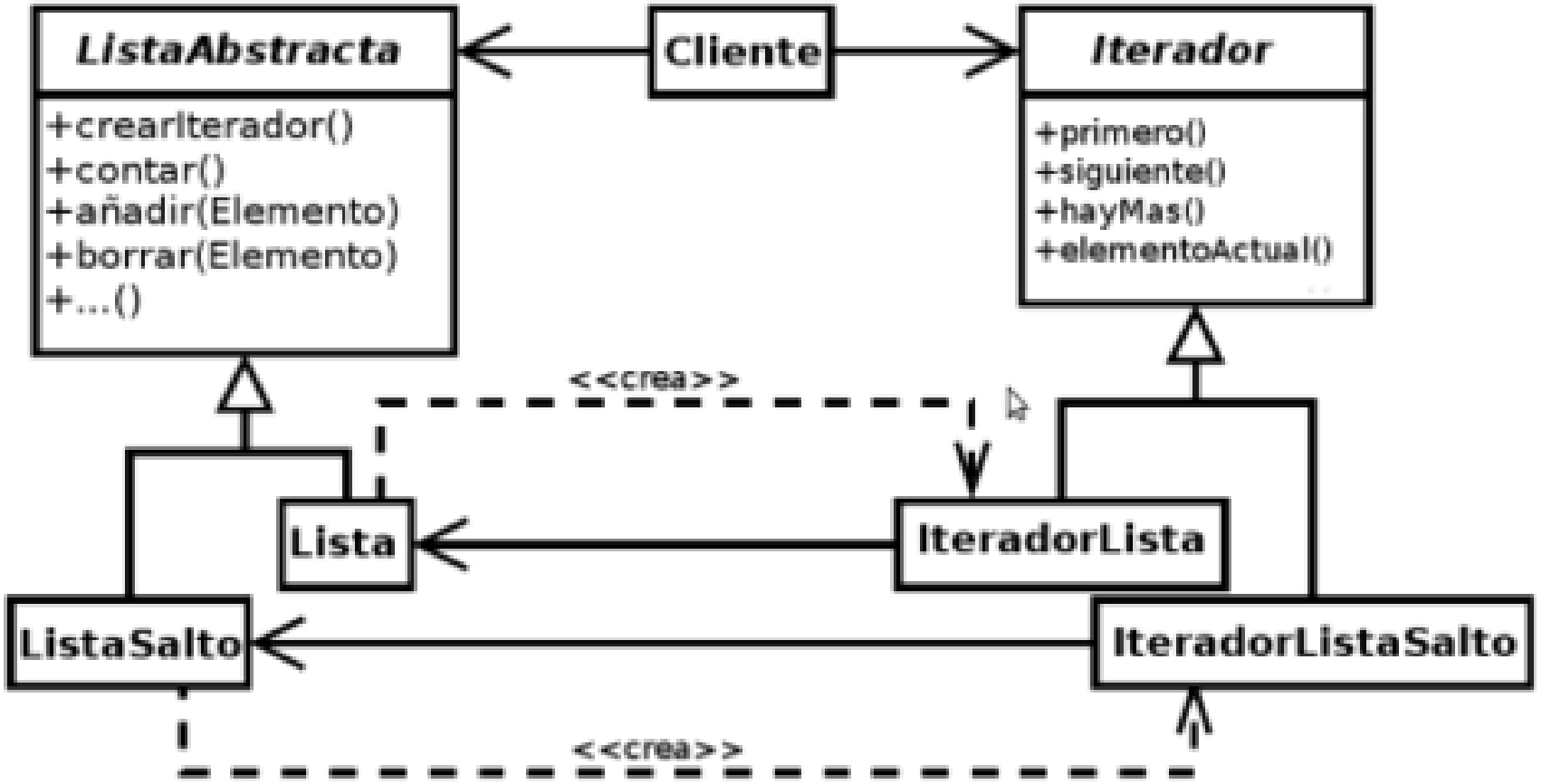
Veamos un ejemplo

Patrones de diseño implicados

- Patrón Iterator
  - Define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección.
  - Algunos de los métodos que podemos definir en la interfaz Iterador son:
- Primero(), Siguiente(), HayMas() y ElementoActual().
- Este patrón de diseño permite recorrer una estructura de datos sin que sea necesario conocer la estructura interna de la misma.

Patrones de diseño implicados

- Patrón Iterator



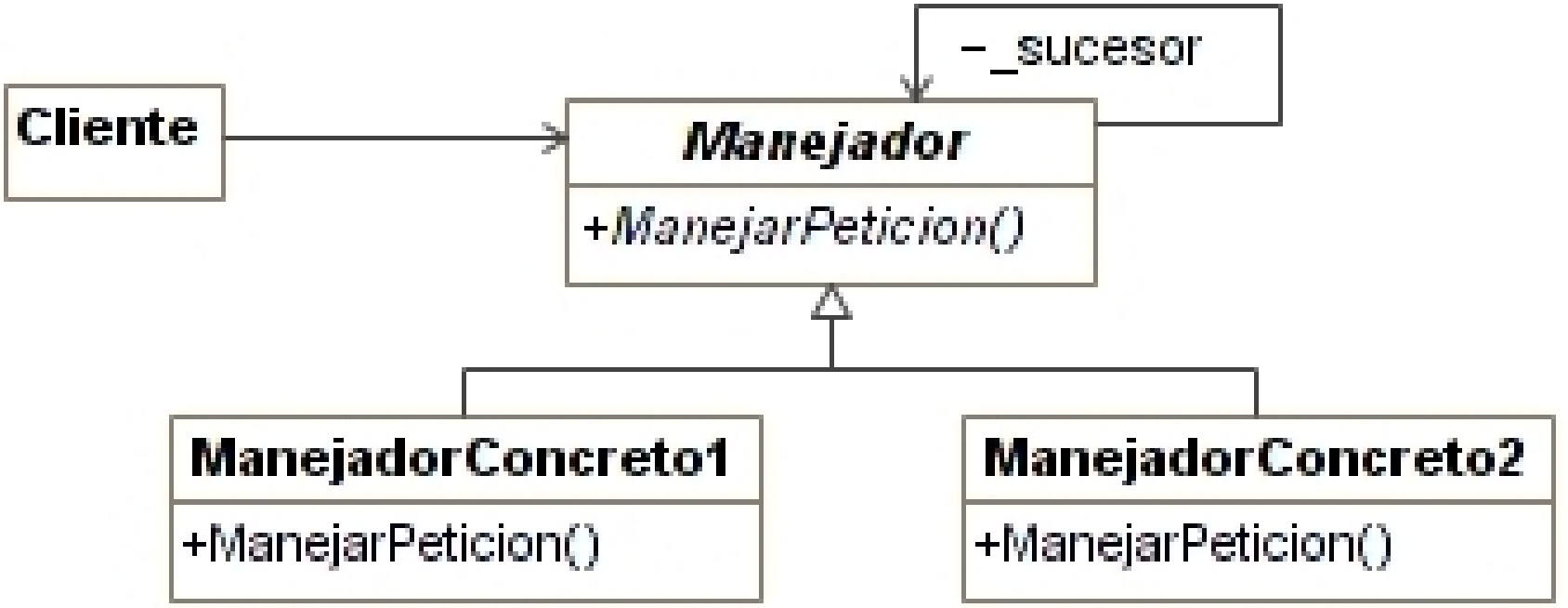
Veamos un ejemplo

Patrones de diseño implicados

- Patrón Chain of Responsibility
  - Es un patrón de comportamiento que evita acoplar el emisor de una petición a su receptor dando a más de un objeto la posibilidad de responder a una petición.
  - Para ello, se encadenan los receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.

Patrones de diseño implicados

- Patrón Chain of Responsibility



Patrones de diseño implicados

- Stream y programación funcional.
  - Un Stream no es ni mas ni menos que un conjunto de funciones que se ejecutan de forma anidada.
  - En algunos aspectos es más cercano a como las personas envocamos cierta resolución de problemas.

Veamos un ejemplo

Patrones de diseño implicados

- **Futuros en java**
  - El concepto de Futuro es el de un objeto que, en algún momento, contendrá el resultado de un método.
  - Cuando llamemos a ese método, devolverá inmediatamente el resultado (un objeto Futuro), y se irá ejecutando de manera asincrona, mientras nuestro método “llamante” continua.
  - En algún momento, el método llamado terminará, y dejara el resultado en el futuro.

Veamos un ejemplo

Patrones de diseño implicados

- **Futuros en java**
  - No podemos crear directamente un futuro. Es una interfaz, y nos lo devolverá, por ejemplo, un método.

```
Future<String> future =
someBigProcess.processVeryLong("test");
```

  - Tiene un parámetro genérico que será el tipo del objeto que esperamos obtener cuando el futuro se complete.
  - Para crearlo, no hay una manera directa.

Patrones de diseño implicados

- **Futuros en java**
  - Por ejemplo, la más sencilla sería esta:

```
private final ExecutorService executor =
Executors.newFixedThreadPool(5);

public Future<String> processVeryLong(String param1) throws
InterruptedException {

    return executor.submit(() -> {

        TimeUnit.SECONDS.sleep(5);

        LOG.info("Terminando processVeryLong...");

        return param1.concat(" result");

    });

}
```

Patrones de diseño implicados

- **Futuros en java**
  - Usamos la interfaz ExecutorService para "lanzar" un Callable (con un lambda de Java 8).
  - Para eso necesitamos haber definido el Executor, que es una clase que se encarga de gestionar pool de threads.
  - Su método 'submit' ejecuta en otro thread el Callable o Runnable recibido, y devuelve un Future, que, cuando se complete la ejecución, contendrá el resultado.

Patrones de diseño implicados

- **Futuros en java**
  - El uso principal de la interfaz Future es su método ‘get’:

String result = future.get();

  - Este método es el que obtiene el valor real del futuro.
  - Si el futuro todavía no se ha completado, al llamar a este método nos quedaremos bloqueados hasta que se complete.
  - Ojo, si al completarse el futuro se genera una excepción, la llamada a este método es la que lanzará esa excepción (“envuelta” en una ExecutionException).

Patrones de diseño implicados

- **Futuros en java**
  - Aparte tiene algunos métodos más:
    - cancel(boolean mayInterruptIfRunning) -> Cancela la ejecución del futuro.
    - isCancelled() -> Comprueba si ha sido cancelado.
    - isDone() -> Comprueba si el futuro se ha completado.
  - Pero no nos ofrece ninguna posibilidad más.
  - Si usamos la interfaz Future, en algún momento, si o si, tendremos que bloquear nuestro thread para obtener el resultado.
  - La interfaz Future de java se queda muy corta para siquiera empezar a montar un sistema “reactivo”.
  - Nos ofrece una manera sencilla de ejecutar métodos de manera asíncrona, pero nada más.

Veamos un ejemplo

Patrones de diseño implicados

- **Java 8 – Interfaces funcionales**
  - Se le conoce como interface funcional a toda aquella interface que tenga **solamente un método abstracto**, es decir puede implementar uno o más métodos default, pero deberá tener forzosamente un único método abstracto.
  - Si no te queda claro que es un método abstracto en una interface, es **un método sin implementar**.
  - Las interfaces funcionales es un nuevo concepto que fue agregado a partir de la versión 8 de Java y cobran su importancia al utilizar expresiones lambda (λ).

Patrones de diseño implicados

- **Java 8 – Interfaces funcionales**
  - A continuación, veremos ejemplos de dos interfaces funcionales válidas:

```
public interface IStrategy {
    public String sayHelloTo(String name);
}
```

  - En esta interface, definimos un método que manda un saludo a la persona que le pasamos como parámetro, pero todavía no está definido el cuerpo del método.

Patrones de diseño implicados

- **Java 8 – Interfaces funcionales**
  - Esta segunda interface, también es una interface funcional, ya que solo tiene un método abstracto.

```
public interface IStrategy {

    public String sayHelloTo(String name);

    public default String sayHelloWord(){
        return "Hello word";
    }
}
```

Patrones de diseño implicados

- **Java 8 – Interfaces funcionales**
  - Otra forma de asegurarnos de que estamos definiendo correctamente una interface funcional, es anotarla con **@FunctionalInterface** , ya que al anotarla el IDE automáticamente nos arrojara un error si no cumplimos con las reglas de una interface funcional.
  - Sin embargo, es importante resaltar que en tiempo de ejecución no nos dará un comportamiento diferente, puesto que es utilizada para prevenir errores al momento de definir las interfaces.

Patrones de diseño implicados

- **Java 8 – Interfaces funcionales**

```
@FunctionalInterface
public interface IStrategy {

    public String sayHelloTo(String name);

    public default String sayHelloWord(){
        return "Hello word";
    }
}
```

- Ya con la teoría, veamos cómo podría utilizar la clase IStrategy mediante expresiones lambda:

Patrones de diseño implicados

```
package com.intro.a_patrones.f_interfaz_funcional;

public class UsandoInterfacesFuncionales {
    public static void ejecutarEjemplo() {
        IStrategy strategy = (name) -> "Hello " + name;

        System.out.println(strategy.sayHelloTo("Oscar López"));
        System.out.println(strategy.sayHelloWord());
    }
}
```

- Veamos como en la línea 8 definimos una expresión lamba en donde concatena Hello con el nombre de la persona que es pasada como parámetro, luego esta expresión es asignada a la variable a la variable strategy , la cual es de tipo IStrategy .
- Luego de esto mandamos llamar al método sayHelloTo el cual nos regresa el saludo para Oscar López, seguido en la línea 11 llamamos al método sayHelloWord , el cual manda un Hello Word.

Patrones de diseño implicados

```java
package com.intro.a_patrones.f_interfaz_funcional;

public class UsandoInterfacesFuncionales {
    public static void ejecutarEjemplo() {
        IStrategy strategy = (name) -> "Hello " + name;

        System.out.println(strategy.sayHelloTo("Oscar López"));
        System.out.println(strategy.sayHelloWord());
    }
}
```

- Como vemos la expresión lambda siempre se asigna al método abstracto, por este motivo el método sayHelloTo es implementado con el cuerpo de la expresión lamba, mientras que el método sayHelloWord continua con la misma implementación que viene desde la interface.
- Observemos la imagen anterior, cuando una expresión lambda es asignada a una interface, está siempre implementará el método abstracto, es por esta razón por la que solo puede existir un método abstracto y muchos defaults.

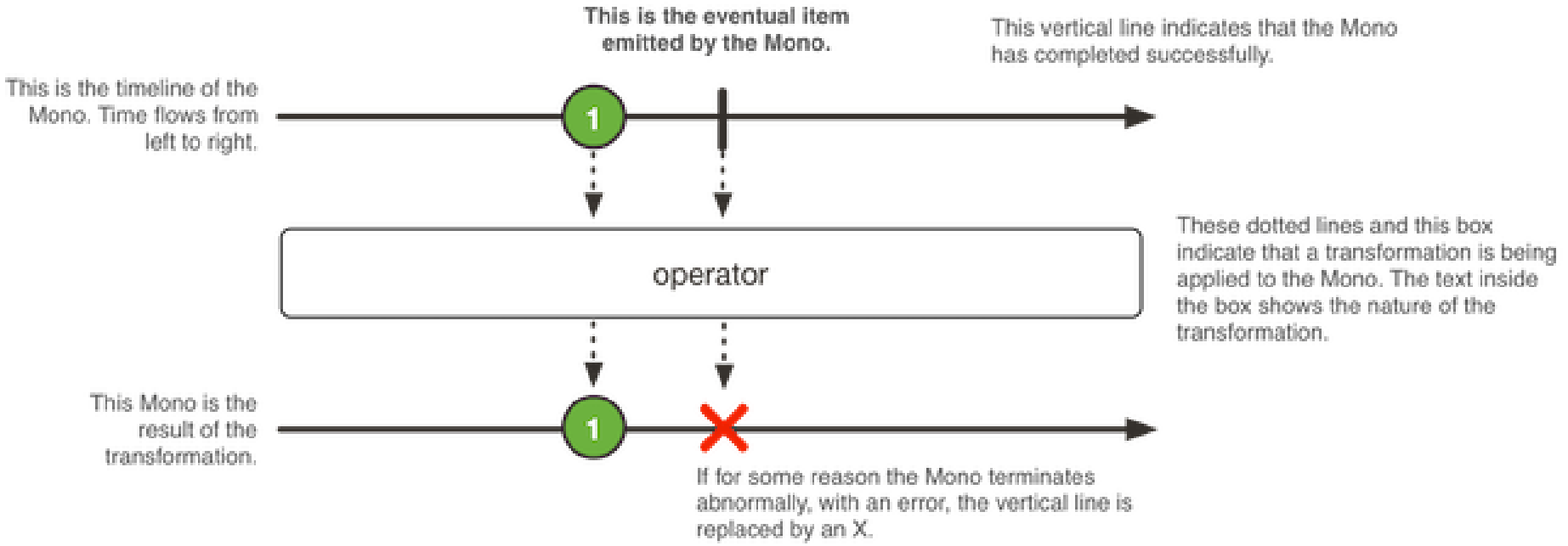
Veamos un ejemplo

APIs Java de programación reactiva

- **RxJava 2**
  - Esta librería, y su versión 1.x fueron las pioneras en el desarrollo reactivo Java.
  - Se encuentran completamente integradas en frameworks como Spring MVC, Spring Cloud y Netflix OSS.
- **Project Reactor**
  - Fue concebida con la implicación del equipo responsable de RxJava 2 por lo que comparten gran parte de la base arquitectónica.
  - Su principal ventaja es que al ser parte de Pivotal ha sido la elegida como fundación del futuro Spring 5 WebFlux Framework.

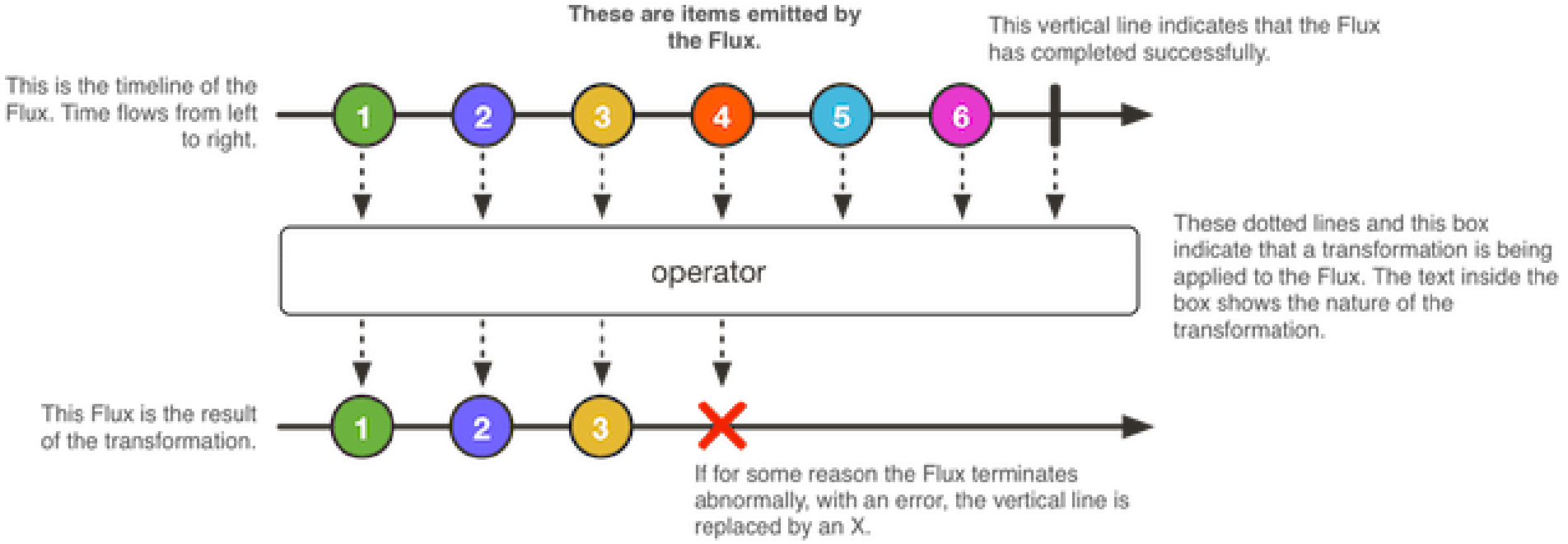
APIs Java de programación reactiva

- **Project Reactor**
  - Este API introduce los tipos Flux y Mono como implementaciones de Publisher, los cuales generan series de 0…N y 0…1 elementos respectivamente.
  - **Mono**:



APIs Java de programación reactiva

- **Project Reactor**
  - Este API introduce los tipos Flux y Mono como implementaciones de Publisher, los cuales generan series de 0…N y 0…1 elementos respectivamente.
  - **Flux**:



APIs Java de programación reactiva

- **Project Reactor**
  - El siguiente ejemplo muestra la creación y suscripción a una secuencia de números aleatorios generados cada segundo.
  - Como se puede ver se trata de un API sencilla e intuitiva que le resultará familiar a cualquiera que haya trabajado con Java 8 Streams.

```
Flux<Double> randomGenerator = Flux.range(1, 4)
.delayMillis(1000)
.map(i -> Math.random())
.log();
randomGenerator.subscribe(number -> logger.info("Got random number {}", number);
```

APIs Java de programación reactiva

- **Project Reactor**

```
1: [main] INFO reactor.Flux.Peek.1 - onSubscribe(FluxPeek.PeekSubscriber)
2: [main] INFO reactor.Flux.Peek.1 - request(unbounded)
3: [timer-1] INFO reactor.Flux.Peek.1 - onNext(0.05361127029313428)
4: [timer-1] INFO es.profile.spring5.playground.reactive.service.RandomNumbersServiceImpl - Got random number
0.05361127029313428
3: [timer-1] INFO reactor.Flux.Peek.1 - onNext(0.711925912668467)
4: [timer-1] INFO es.profile.spring5.playground.reactive.service.RandomNumbersServiceImpl - Got random number
0.711925912668467
3: [timer-1] INFO reactor.Flux.Peek.1 - onNext(0.8631082308572313)
4: [timer-1] INFO es.profile.spring5.playground.reactive.service.RandomNumbersServiceImpl - Got random number
0.8631082308572313
3: [timer-1] INFO reactor.Flux.Peek.1 - onNext(0.2797390339259114)
4: [timer-1] INFO es.profile.spring5.playground.reactive.service.RandomNumbersServiceImpl - Got random number
0.2797390339259114
5: [timer-1] INFO reactor.Flux.Peek.1 - onComplete()
```

Veamos un ejemplo

APIs Java de programación reactiva

- **Project Reactor**
- Los logs de ejecución muestran los diferentes eventos:
  - Suscripción.
  - Solicitud de elementos sin límite.
  - Generación de elementos en un hilo timer-1.
  - Entrega de los elementos al suscriptor en el hilo main.
  - Fin de la secuencia señalizado por el evento onComplete().

APIs Java de programación reactiva

- **Project Reactor**
  - Cominación de Flujos: Project reactor proporciona diversas maneras para combinar flujos.
  - Algunos ejemplos API de Flux:
    - .concat(): Concatena todas las fuentes proporcionadas en un Iterable, enviando elementos emitidos por las fuentes en sentido descendente.
    - .concatMap(): Transforma los elementos emitidos por este Flujo de forma asíncrona en Publishers, luego aplana estos editores internos en un solo Flujo, secuencialmente y conservando el orden mediante concatenación.
    - .count(): Cuenta el número de valores en este Flujo.
    - .combineLatest(): Crea un flujo cuyos datos se generen mediante la combinación del valor publicado más recientemente de cada uno de los orígenes de Publisher.
    - Y muchas más formas que puedes ver en la API de Flux.

APIs Java de programación reactiva

- **Project Reactor**
  - También podemos añadir “disparadores” o “triggers” y modificar el comportamiento del flujo.
  - Alguno ejemplos de la API de Flux:
    - `doOnComplete(Runnable onComplete)`
    - `doOnDiscard(Class<R> type, Consumer<? super R> discardHook)`
    - `doOnEach(Consumer<? super Signal<T>> signalConsumer)`
    - `doOnError(Class<E> exceptionType, Consumer<? super E> onError)`
    - `doOnNext(Consumer<? super T> onNext)`
    - `doOnSubscribe(Consumer<? super Subscription> onSubscribe)`
    - `doOnRequest(LongConsumer consumer)`
    - `doOnTerminate(Runnable onTerminate)`

APIs Java de programación reactiva

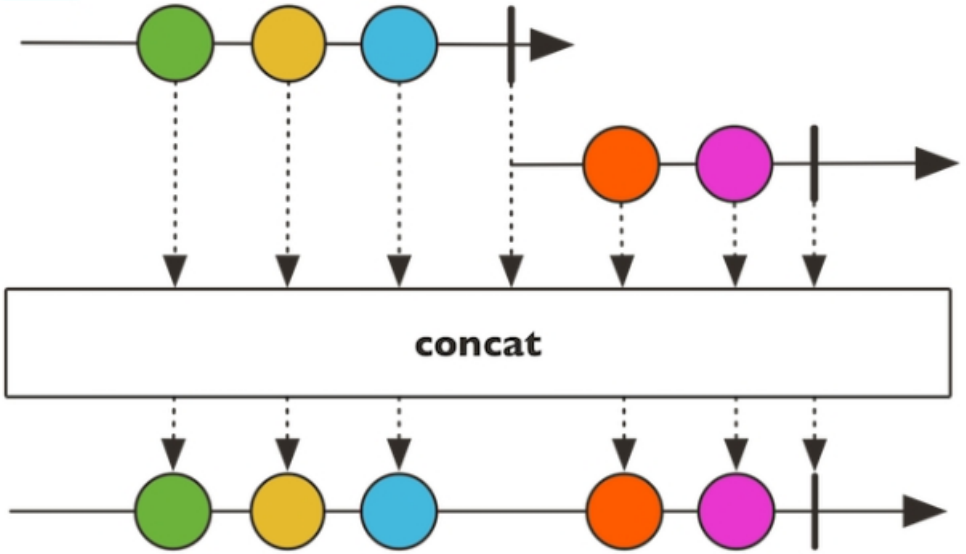
- **Project Reactor**
  - Concat, primero un flujo, y luego se le concatena el otro, digamos que hace combinación lenta.
  - Merge, sin embargo, ambos flujos están “vivos”, por lo que los datos se irán “mezclando” tan pronto como sean emitidos.
  - La forma de implementarlos es relativamente sencilla:

```
2  Flux.concat(mono1, mono3, mono2)
3          .subscribe(System.out::print);
4
5  Flux.concat(flux2, flux1)
6          .subscribe(System.out::print);
```

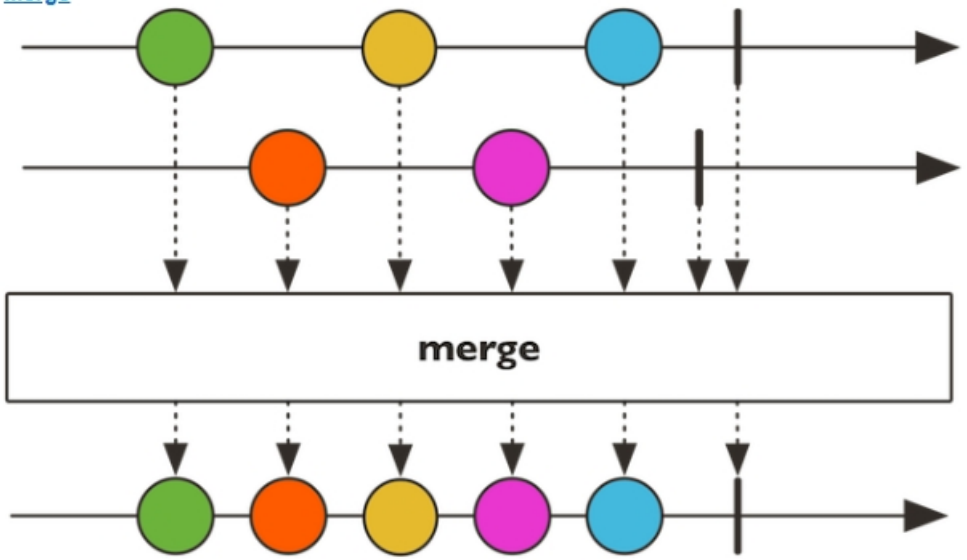
```
Flux<Integer> fluxOfIntegers = Flux.merge(evenNumbers, oddNumbers);
```

APIs Java de programación reactiva

Concat



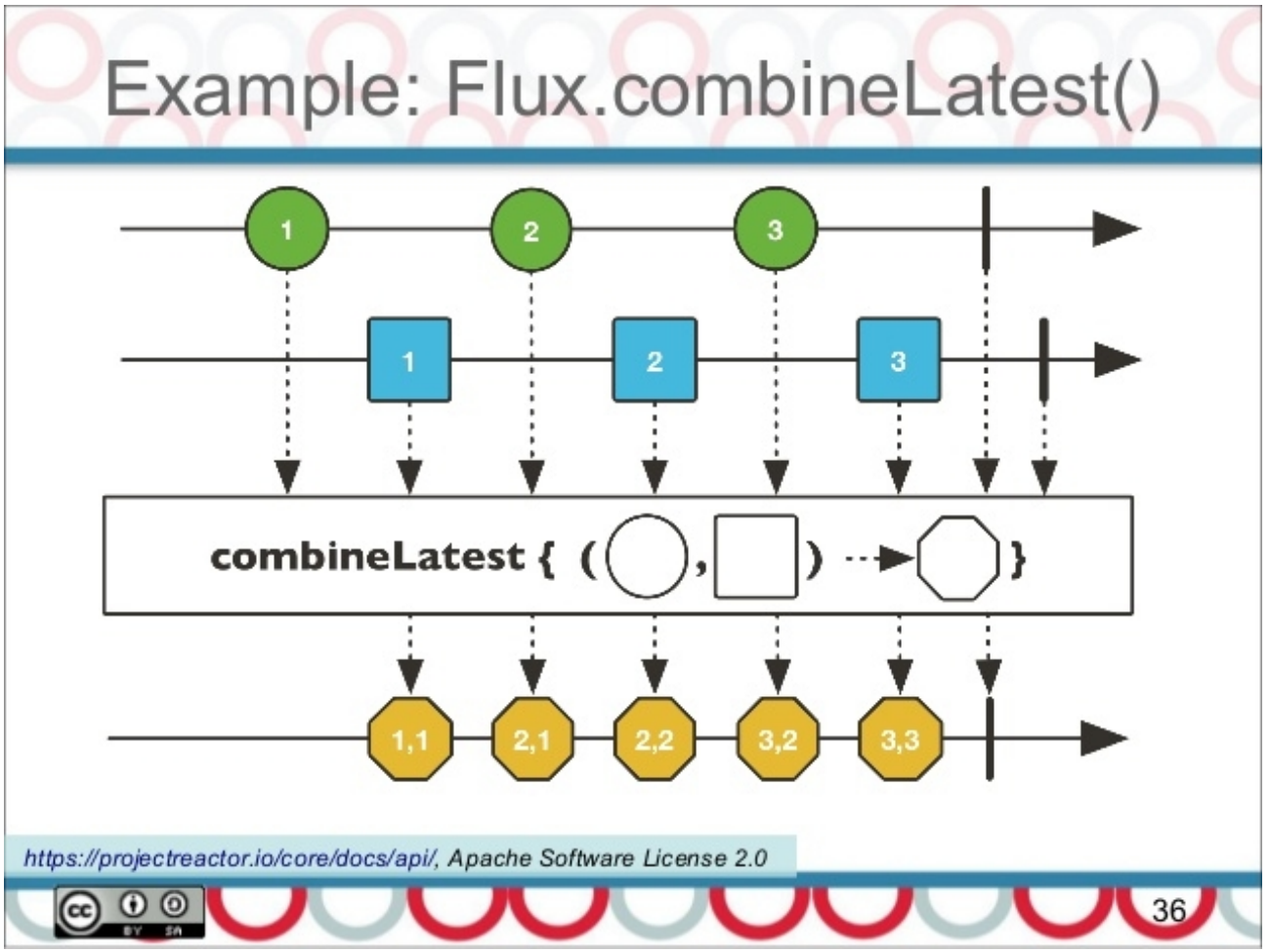
Merge



Veamos un ejemplo

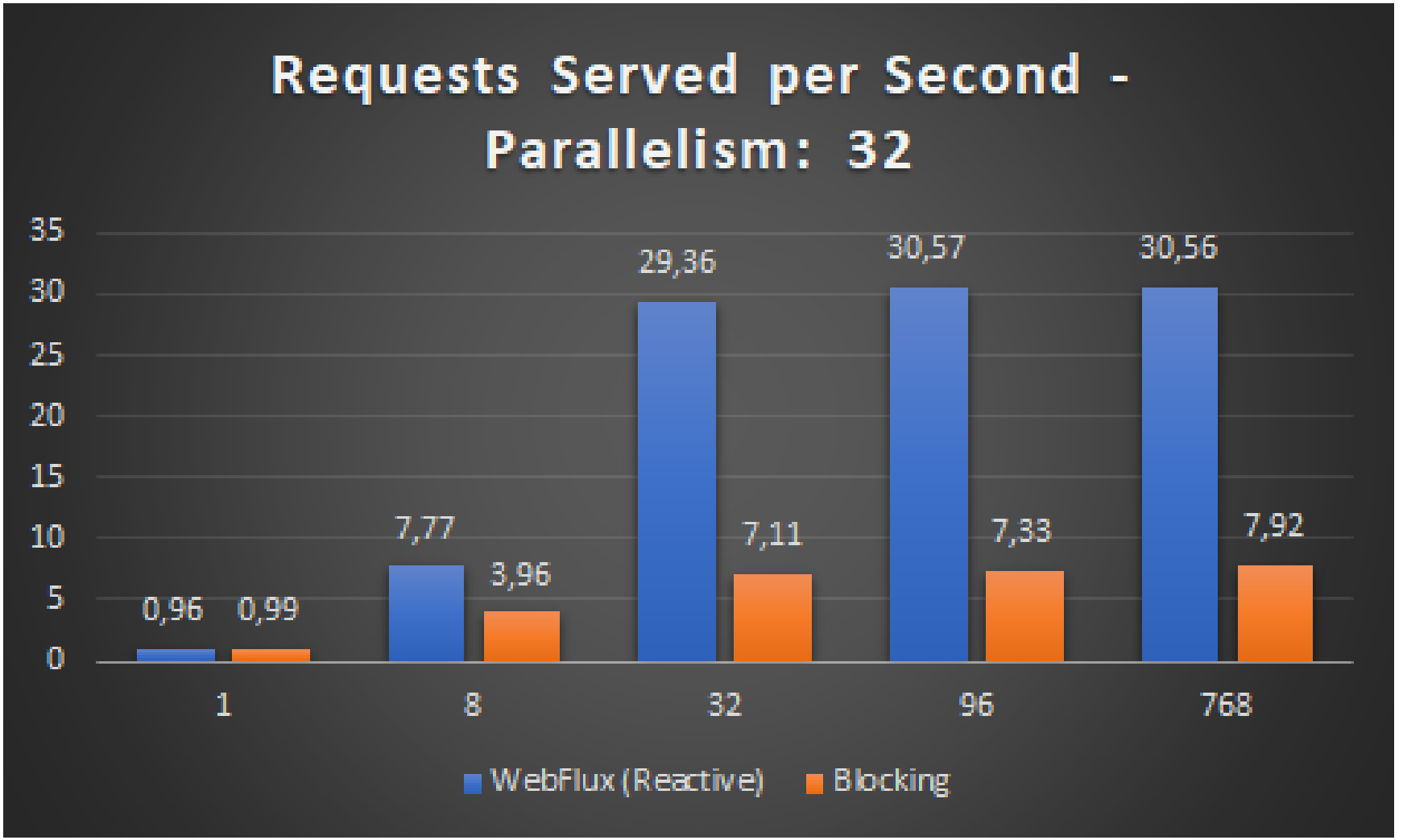
APIs Java de programación reactiva

- **Project Reactor**
  - Cominación de Flujos:



Arquitectura webflux

- Arquitecturas Spring MVC VS Reactiva:



Arquitectura webflux

- Arquitecturas Spring MVC VS Reactiva:

STATISTICS Servlet stack

Executions

	Total	OK	KO
	20000	20000	0
Mean req/s	714.286	714.286	-

Response Time (ms)

	Total	OK	KO
Min	202	202	-
50th percentile	495	495	-
75th percentile	778	778	-
95th percentile	1129	1129	-
99th percentile	1215	1215	-
Max	1703	1703	-
Mean	578	578	-
Std Deviation	285	285	-

STATISTICS Reactive stack

Executions

	Total	OK	KO
	20000	20000	0
Mean req/s	833.333	833.333	-

Response Time (ms)

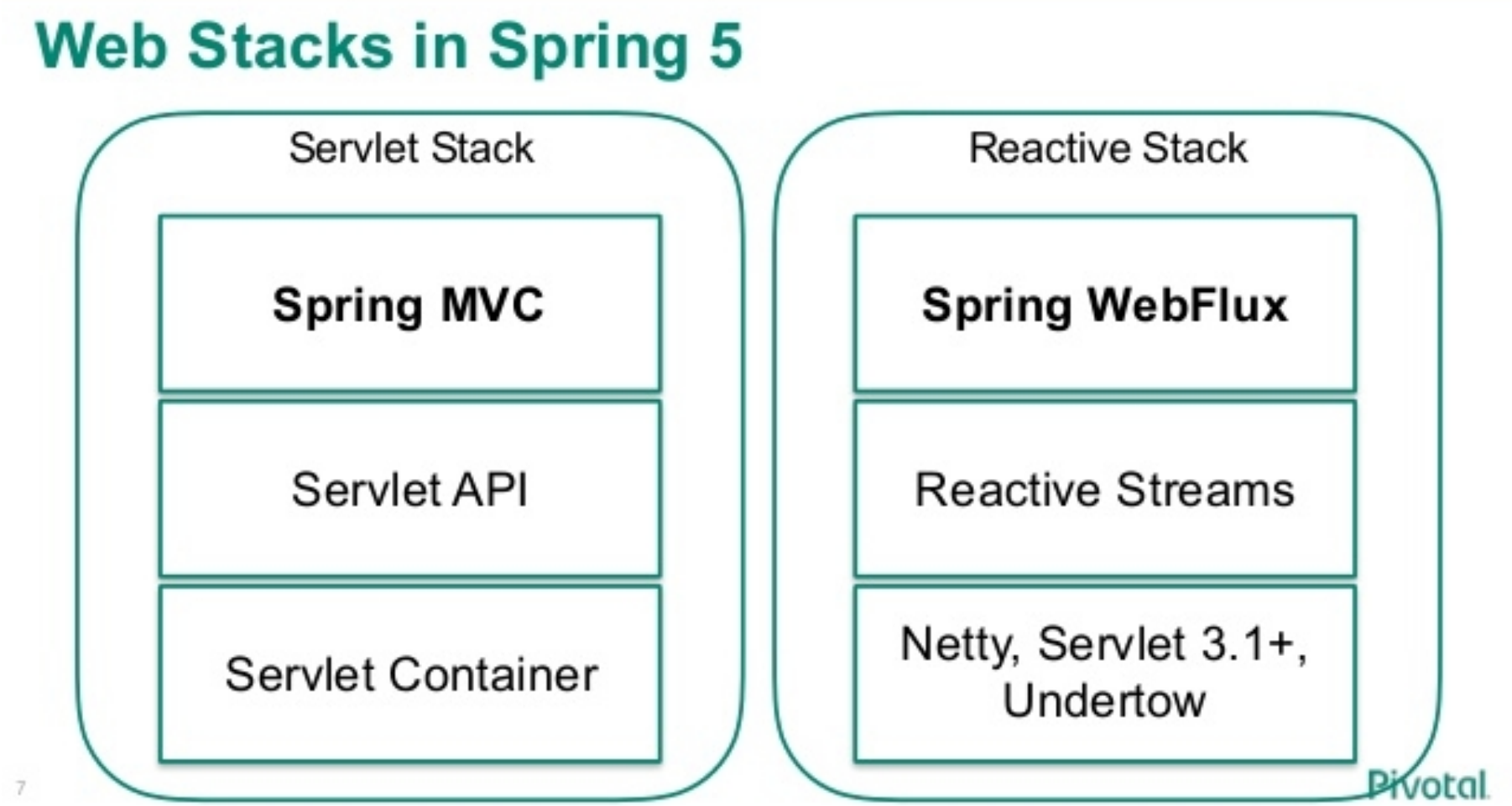
	Total	OK	KO
Min	201	201	-
50th percentile	260	260	-
75th percentile	317	317	-
95th percentile	480	480	-
99th percentile	716	716	-
Max	1111	1111	-
Mean	291	291	-
Std Deviation	100	100	-

Arquitectura webflux

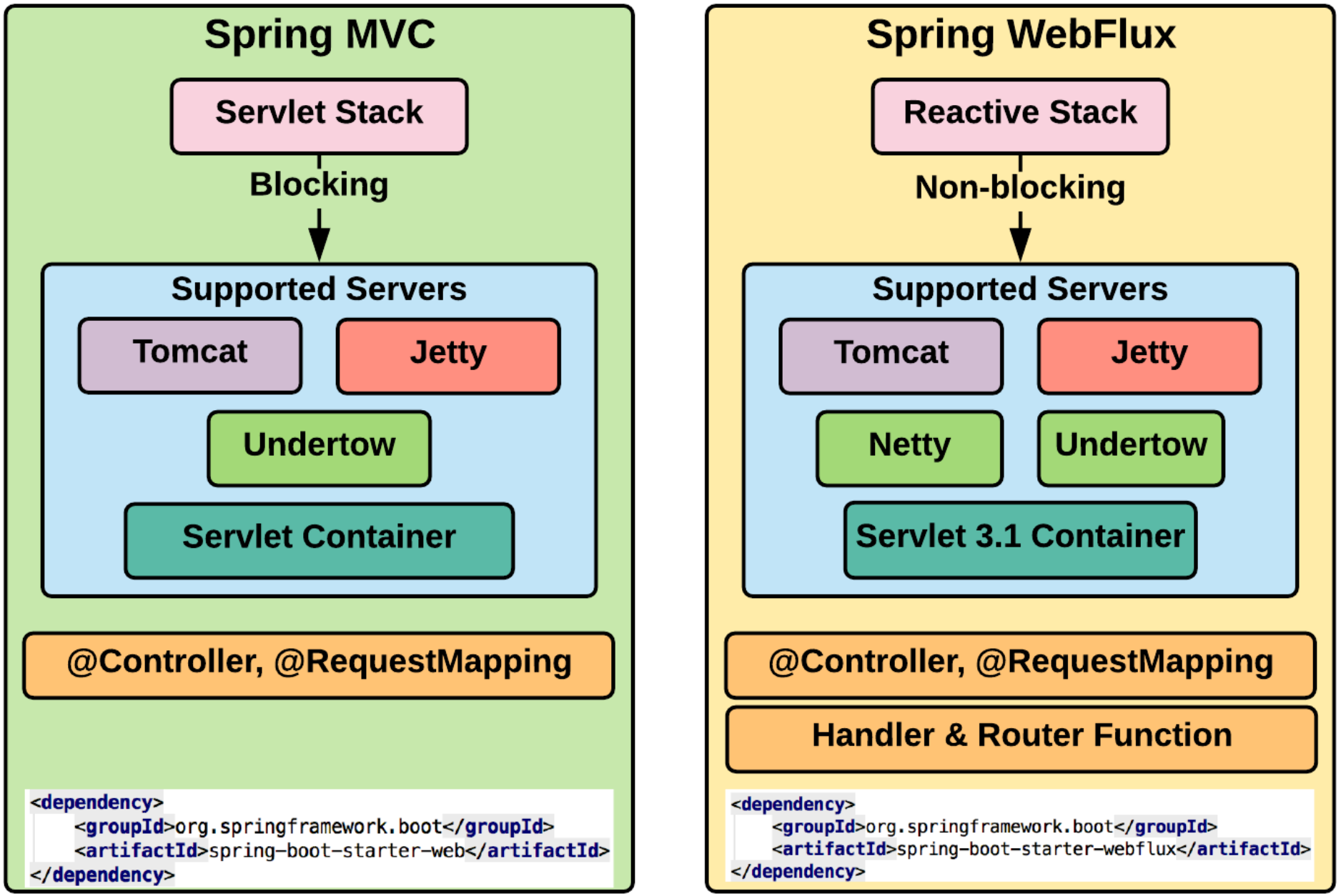
- Integración fácil con plantillas Thymeleaf y MongoDB y otras bases de datos NoSQL que implementan la reactividad de forma nativa (Cassandra, CouchBase...):

Arquitectura webflux

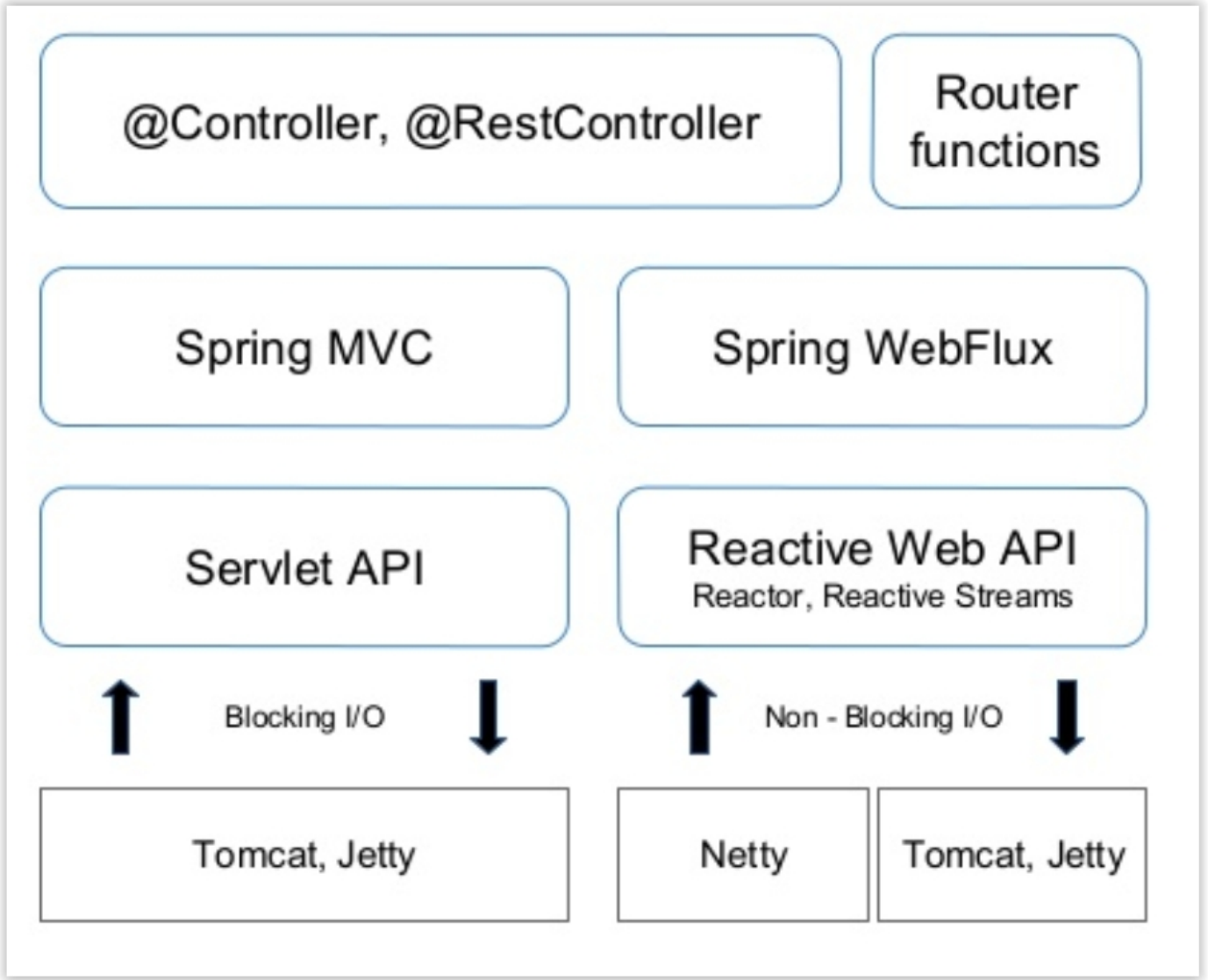
- Arquitecturas Spring MVC VS Reactiva:



Arquitectura webflux



Arquitectura webflux



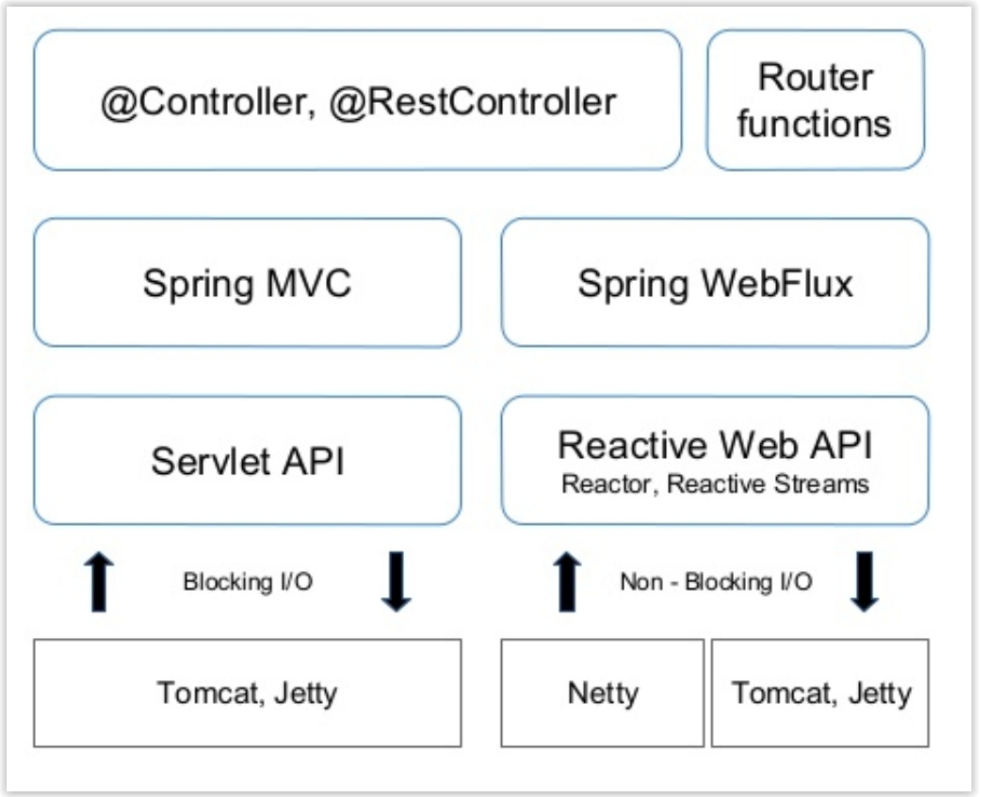
Arquitectura webflux

¿Pueden convivir Spring MVC y WebFlux?

1 Thread gestiona toda llamada, quedando bloqueado hasta que esta termina produciendo la respuesta.

Trabaja con un *pool de threads* esperando a recibir peticiones

Idóneo si el '*sistema de almacenamiento*' es bloqueante



Llamadas no bloqueantes

Modelo de concurrencia '*event loop*'

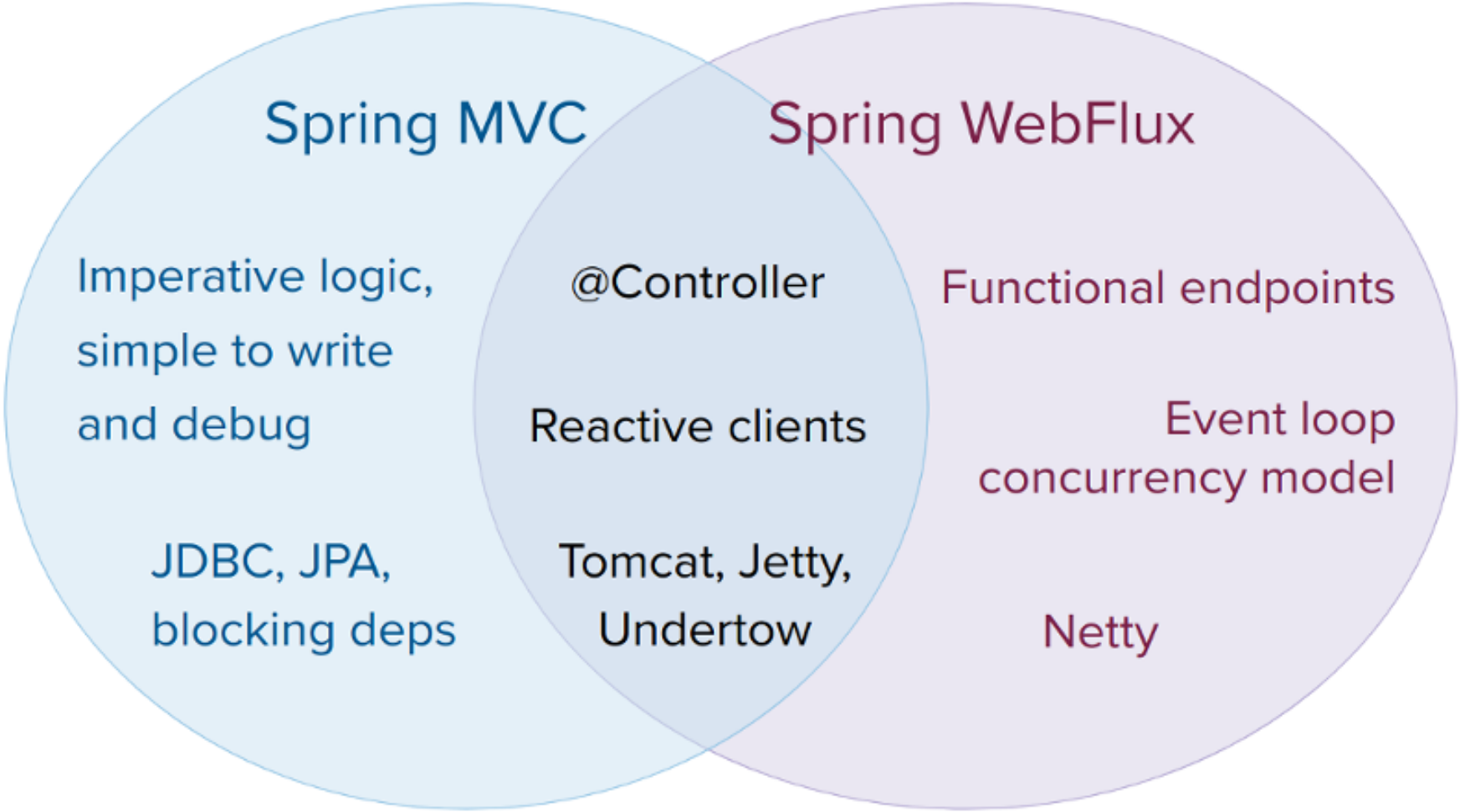
Endpoints funcionales

Idóneo si el '*sistema de almacenamiento*' es NO bloqueante

Ref: Juergen Hoeller

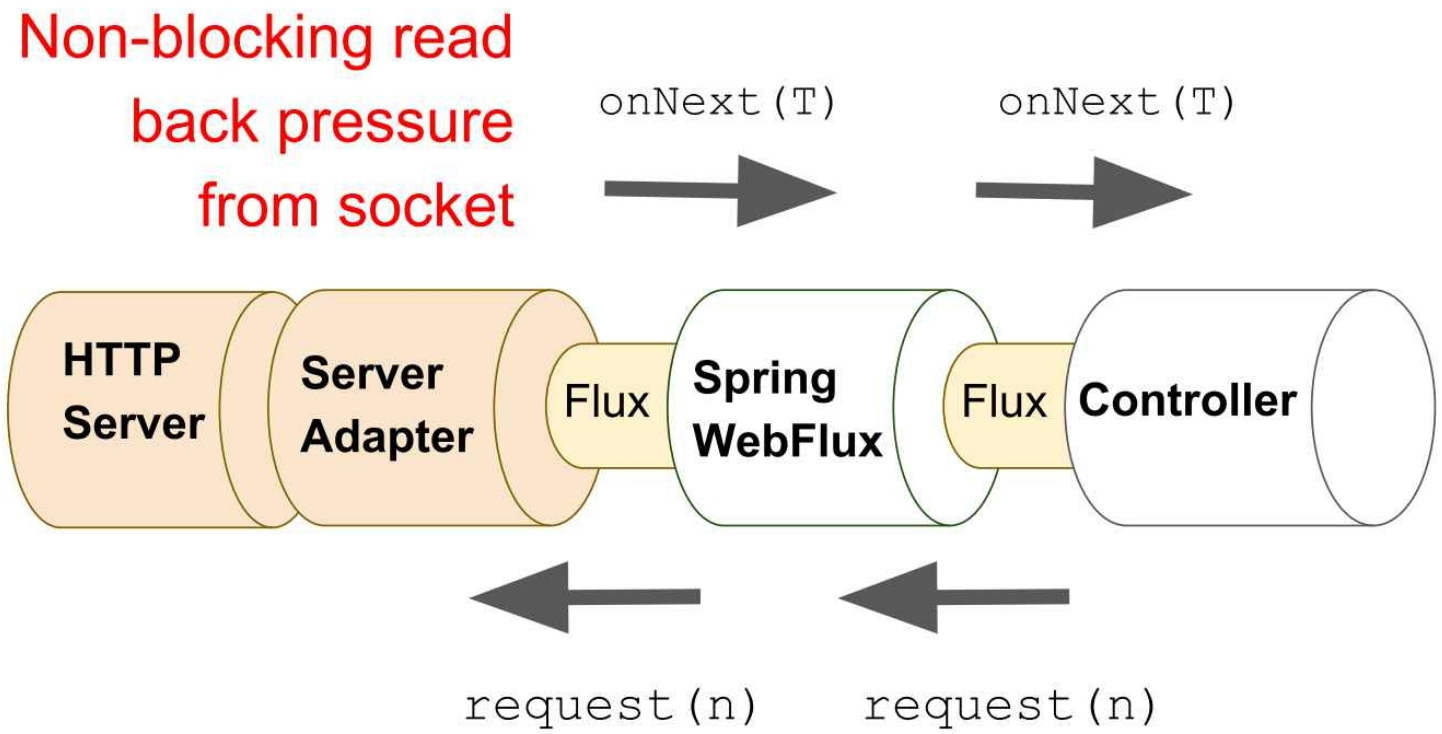
Veamos un ejemplo

Arquitectura webflux



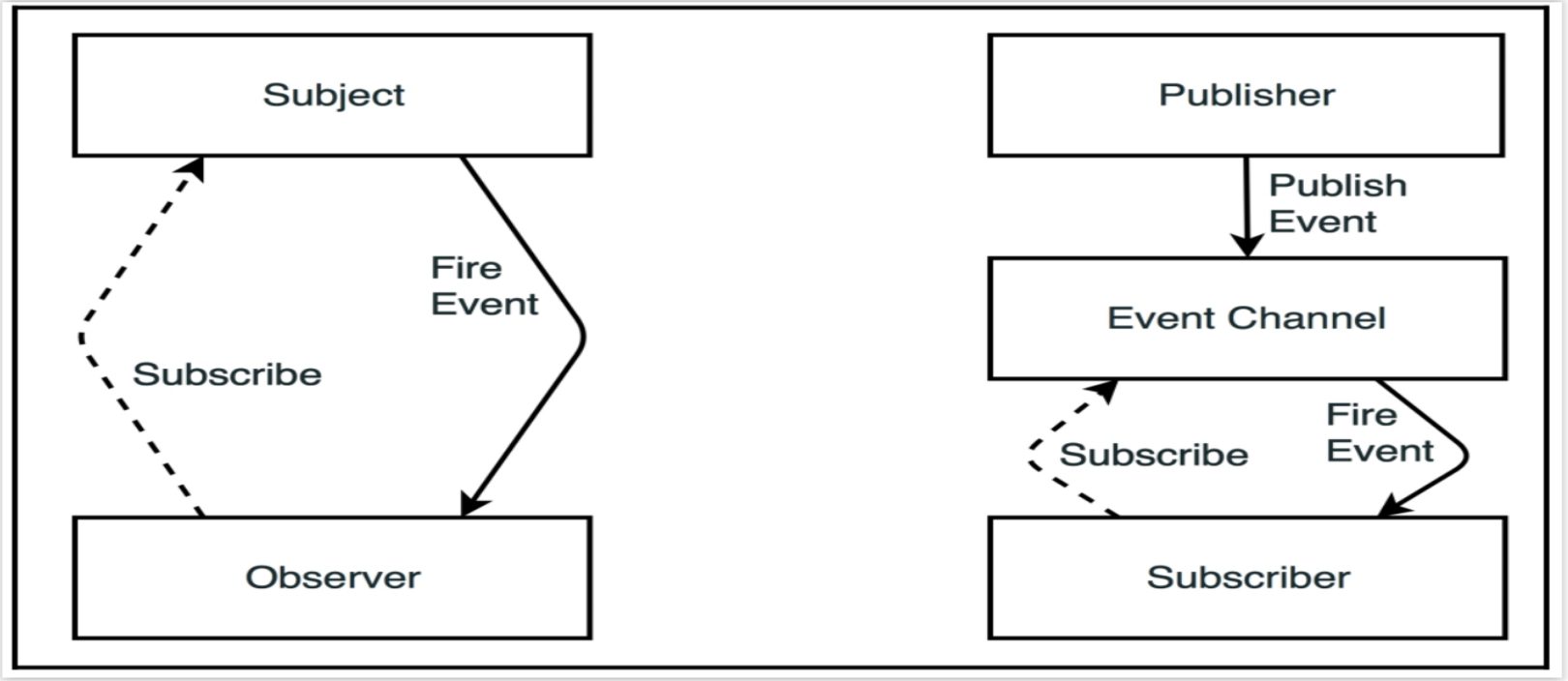
Arquitectura webflux

Data ingestion on reactive stack



Arquitectura webflux

- WebFlux emplea el patrón Publish-Subscribe pattern en vez del patrón Observer:
  - A diferencia del patrón Observer, en el patrón de Publicación-Suscripción, los editores y suscriptores no necesitan conocerse, como es el caso, representado en el siguiente diagrama::



Arquitectura webflux

- Un patrón de publicación-suscripción proporciona un nivel adicional de indirección entre los editores y los suscriptores.
- Los suscriptores conocen los canales de eventos que emiten notificaciones, pero generalmente no les importan las identidades de los publisers.
- Además, cada canal de eventos puede tener algunos publisers al mismo tiempo.
- El diagrama anterior debería ayudar a detectar la diferencia entre los patrones de Observador y Publicación-Suscripción.

Arquitectura webflux

- El canal de eventos (también conocido como intermediario de mensajes o bus de eventos) puede filtrar adicionalmente los mensajes entrantes y distribuirlos entre los suscriptores.
- El filtrado y el enrutamiento pueden realizarse según el contenido del mensaje, el tema del mensaje o, a veces, incluso ambos.
- En consecuencia, los suscriptores en un topic-based system recibirán todos los mensajes publicados sobre los temas de interés.:

Arquitectura webflux

- Tipos de Filtrado de mensajes:
  - Sistema basado en temas **(topic-based system):**
    - Los mensajes se publican en "temas" o se denominan canales lógicos.
    - Los suscriptores en un sistema basado en temas recibirán todos los mensajes publicados sobre los temas a los que se suscriben, y todos los suscriptores de un tema recibirán los mismos mensajes.
    - El publicador es responsable de definir las clases de mensajes a los que se pueden suscribir los suscriptores.
  - Sistema basado en contenido **(content-based system):**
    - Los mensajes solo se entregan a un suscriptor si los atributos o el contenido de esos mensajes coinciden con las restricciones definidas por el suscriptor.
    - El suscriptor es responsable de clasificar los mensajes.

Arquitectura webflux

- Algunos sistemas soportan un híbrido de los dos; los publicadores publican mensajes a un tema mientras los suscriptores registran suscripciones basadas en contenido a uno o más temas.
- La anotación @EventListener de Spring Framework hace posible aplicar enrutamiento tanto basado en temas como en contenido.
- Los tipos de mensajes podrían desempeñar el papel de los temas; el atributo de condición habilita el manejo de eventos de enrutamiento basado en contenido basado en Spring Expression Language (SpEL).

Arquitectura webflux

- Para entender el patrón de publicación-suscripción en Spring Framework, haremos un ejemplo.
- Supongamos que tenemos que implementar un servicio web simple que muestre la temperatura actual en la habitación.
- Para este propósito, tenemos un sensor de temperatura, que envía eventos con la temperatura actual en grados Celsius de vez en cuando.
- La simulación del sensor de temperatura será un generador de números aleatorios.

Arquitectura webflux

- Para hacer que nuestra aplicación siga el diseño reactivo, no podemos usar un viejo modelo de extracción para la recuperación de datos.
- Afortunadamente, hoy en día tenemos algunos protocolos bien adoptados para la propagación asíncrona de mensajes de un servidor a un cliente, a saber, WebSockets y eventos enviados por el servidor (SSE o Server Sended Events).
- En el ejemplo actual, utilizaremos el último.
- El SSE permite que un cliente reciba actualizaciones automáticas de un servidor, y se usa comúnmente para enviar actualizaciones de mensajes o flujos de datos continuos a un navegador.

Arquitectura webflux

- Con el inicio de HTML5, todos los navegadores modernos tienen una API de JavaScript llamada EventSource, que es utilizada por los clientes que solicitan una URL particular para recibir un flujo de eventos.
- EventSource también se conecta automáticamente de forma predeterminada en el caso de problemas de comunicación.
- Enlaces de interés:
  - https://hpbn.co/server-sent-events-sse/
  - https://www.sitepoint.com/real-time-apps-websockets-server-sentevents/

HTTP asíncrono con Spring Web MVC

- Ahora podemos delinear el diseño de nuestro sistema en el siguiente diagrama:

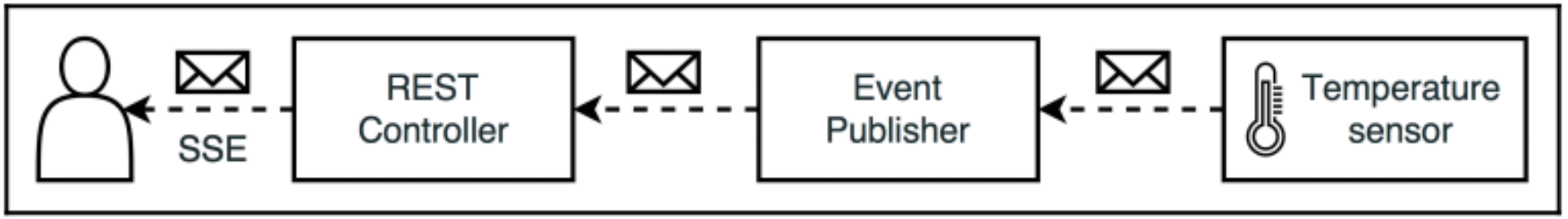


Diagram 2.5 Events flow from a temperature sensor to a user

- Para simular el sensor, implementemos la clase TemperatureSensor y decorémosla con una anotación @Component para registrar el Spring Bean

HTTP asíncrono con Spring Web MVC

```
@Component
public class TemperatureSensor {
    private final ApplicationEventPublisher publisher; // (1)
    private final Random rnd = new Random(); // (2)
    private final ScheduledExecutorService executor = // (3)

    public TemperatureSensor(ApplicationEventPublisher publisher) {

    public void startProcessing() { // (4)

    private void probe() { // (5)
}
```

```
final class Temperature {
    private final double value;

    public Temperature(double value) {

    public double getValue() {

}
```

Veamos un ejemplo

HTTP asíncrono con Spring Web MVC

- Por lo tanto, nuestro sensor de temperatura simulado solo depende de la clase ApplicationEventPublisher (1), proporcionada por Spring Framework.
- Esta clase hace posible publicar eventos en el sistema. Es un requisito tener un generador aleatorio (2) para crear temperaturas con algunos intervalos aleatorios.
- Un proceso de generación de eventos ocurre en un ScheduledExecutorService (3) separado, donde la generación de cada evento programa la siguiente ronda de la generación de un evento con un retraso aleatorio (5.1).
- Toda esa lógica se define en el método de probe() (5).
- A su vez, la clase mencionada tiene el método startProcessing () anotado con @PostConstruct (4), que Spring Framework llama cuando el bean está listo y activa toda la secuencia de valores de temperatura aleatorios

HTTP asíncrono con Spring Web MVC

- HTTP asíncrono con Spring Web MVC El soporte asíncrono introducido en Servlet 3.0 amplía la capacidad de procesar una solicitud HTTP en subprocesos que no son contenedores.
- Tal característica es bastante útil para tareas de larga ejecución.
- Con esos cambios, en Spring Web MVC podemos devolver no solo un valor de tipo T en @Controller, sino también un Callable <T> o un DeferredResult <T>
- El Callable <T> se puede ejecutar dentro de un subproceso que no es contenedor, pero aún así, sería una llamada de bloqueo.
- En contraste, DeferredResult <T> permite una generación de respuesta asíncrona en un subproceso no contenedor llamando al método setResult (resultado T) para que pueda usarse dentro del bucle de eventos.
- A partir de la versión 4.2, Spring Web MVC hace posible devolver ResponseBodyEmitter, que se comporta de manera similar a DeferredResult, pero puede usarse para enviar múltiples objetos, donde cada objeto se escribe por separado con una instancia de un convertidor de mensajes (definido por la interfaz HttpMessageConverter)

HTTP asíncrono con Spring Web MVC

- El SseEmitter extiende ResponseBodyEmitter y hace posible enviar muchos mensajes salientes para una solicitud entrante de acuerdo con los requisitos de protocolo de SSE.
- Junto con ResponseBodyEmitter y SseEmitter, Spring Web MVC también respeta la interfaz StreamingResponseBody.
- Cuando se devuelve desde @Controller, nos permite enviar datos sin procesar (bytes de carga útil) de forma asíncrona.
- StreamingResponseBody puede ser muy útil para transmitir archivos grandes sin bloquear los subprocesos de Servlet.

HTTP asíncrono con Spring Web MVC

- **Exponer el punto final SSE.**
- El siguiente paso requiere agregar la clase TemperatureController con la anotación @RestController, lo que significa que el componente se utiliza para la comunicación HTTP, como se muestra en el siguiente código:

```
@RestController
public class TemperatureController {
    private final Set<SseEmitter> clients = // (1)
            new CopyOnWriteArraySet<>();

    public SseEmitter events(HttpServletRequest request) { // (3)

    public void handleMessage(Temperature temperature) { // (11)
}
```

HTTP asíncrono con Spring Web MVC

- Ahora, para comprender la lógica de la clase TemperatureController, necesitamos describir el SseEmitter.
- Spring Web MVC proporciona esa clase con el único propósito de enviar eventos SSE.
- Cuando un método de manejo de solicitudes devuelve la instancia de SseEmitter, el procesamiento de la solicitud real continúa hasta que SseEnitter.complete (), se produce un error o se produce un tiempo de espera.

HTTP asíncrono con Spring Web MVC

- El TemperatureController proporciona un controlador de solicitudes (3) para el URI / temperature-stream (2) y devuelve el SseEmitter (8).
- En el caso de que un cliente solicite ese URI, creamos y devolvemos la nueva instancia de SseEmitter (4) con su registro anterior en la lista de clientes activos (5).
- Además, el constructor SseEmitter puede consumir el parámetro de tiempo de espera.

HTTP asíncrono con Spring Web MVC

- Para la colección de los clientes, podemos usar la clase CopyOnWriteArraySet del paquete java.util.concurrent (1).
- Dicha implementación nos permite modificar la lista e iterar sobre ella al mismo tiempo.
- Cuando un cliente web abre una nueva sesión SSE, agregamos un nuevo emisor a la colección de los clientes.
- El SseEmitter se elimina a sí mismo de la lista de clientes cuando ha terminado de procesar o ha alcanzado el tiempo de espera (6) (7).

HTTP asíncrono con Spring Web MVC

- Ahora, tener un canal de comunicación con los clientes significa que debemos poder recibir eventos sobre los cambios de temperatura.
- Para ese propósito, nuestra clase tiene un método handleMessage () (11).
- Está decorado con la anotación @EventListener (10) para recibir eventos de Spring.
- Este marco invocará el método handleMessage () solo cuando reciba eventos de temperatura, ya que este tipo de argumento del método se conoce como temperatura.
- La anotación @Async (9) marca un método como candidato para la ejecución asíncrona, por lo que se invoca en el grupo de subprocesos configurado manualmente.

HTTP asíncrono con Spring Web MVC

- El método handleMessage () recibe un nuevo evento de temperatura y lo envía de forma asincrónica a todos los clientes en formato JSON en paralelo para cada evento (13).
- Además, al enviar a emisores individuales, rastreamos todos los fallidos (14) y los eliminamos de la lista de clientes activos (15).
- Este enfoque permite detectar clientes que ya no están operativos.
- Desafortunadamente, SseEmitter no proporciona ninguna devolución de llamada para el manejo de errores, y puede hacerse manejando los errores generados por el método send () solamente.

HTTP asíncrono con Spring Web MVC

- **Construyendo una UI con soporte reactivo**
  - Haremos dos ejemplos HTML yJS con soporte SSE y WebFlux
  - Lo último que necesitamos para completar nuestro caso de uso es una página HTML con algún código JavaScript para comunicarnos con el servidor.
  - En aras de la concisión, eliminaremos todas las etiquetas HTML y dejaremos solo el mínimo requerido para lograr un resultado, de la siguiente manera:

HTTP asíncrono con Spring Web MVC

```
<body>
    <ul id="events"></ul>
    <script type="application/javascript">

function add(message) {
    const el = document.createElement("li");
    el.innerHTML = message;
    document.getElementById("events").appendChild(el);
}
var eventSource = new EventSource("/temperature-stream");

eventSource.onmessage = e => { // (2)
    const t = JSON.parse(e.data);
    const fixed = Number(t.value).toFixed(2);
    add('Temperature: ' + fixed + ' C');
}
eventSource.onopen = e => add('Connection opened'); // (3)
eventSource.onerror = e => add('Connection closed'); //
    </script>
</body>
```

Referencias y enlaces de interés

- https://start.spring.io/
- https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html
- https://www.arquitecturajava.com/entendiendo-el-servlet-lifecycle/
- https://www.reactivemanifesto.org/es
- https://anotherdayanotherbug.wordpress.com/2016/01/26/futuros-en-java-parte-1-introduccion/
- https://www.callicoder.com/reactive-rest-apis-spring-webflux-reactive-mongo/
- https://danielggarcia.wordpress.com/category/ingenieria/patrones-de-diseno/
- https://profile.es/blog/que-es-la-programacion-reactiva-una-introduccion/
- https://www.arquitecturajava.com/programacion-funcional-java-8-streams/
- https://anotherdayanotherbug.wordpress.com/2016/01/26/futuros-en-java-parte-1-introduccion/
- https://www.oscarblancarteblog.com/2016/12/08/java-8-interfaces-funcionales/
- https://grokonez.com/reactive-programming/reactor/reactor-how-to-combine-flux-mono-reactive-programming
-