SPRING WebFlux

Germán Caballero Rodríguez
gcaballero@gmail.com

INDICE

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

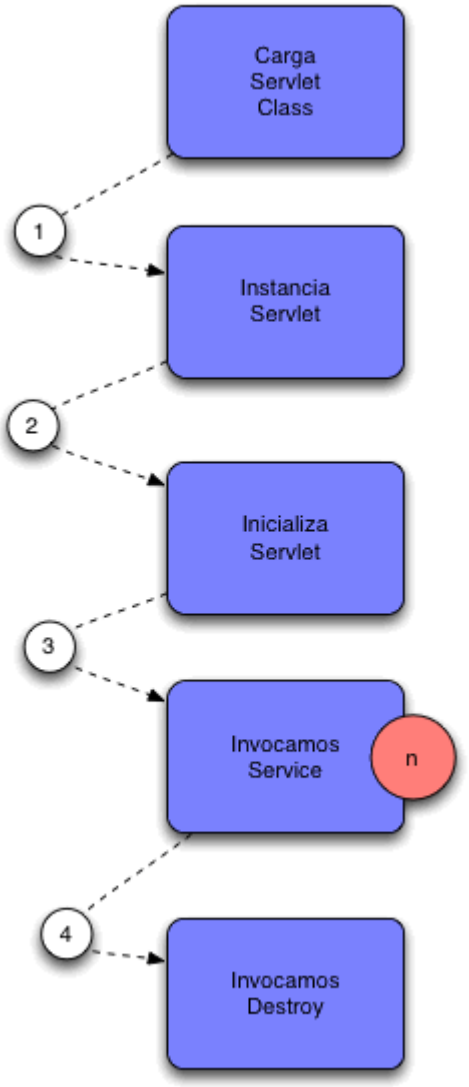
- Sólo unos pocos años atrás, una aplicación grande tenía decenas de servidores, segundos de tiempo de respuesta, horas de mantenimiento fuera de línea y gigabytes en datos.
- Hoy, las aplicaciones se despliegan en cualquier cosa, desde dispositivos móviles hasta clusters en la nube corriendo en miles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

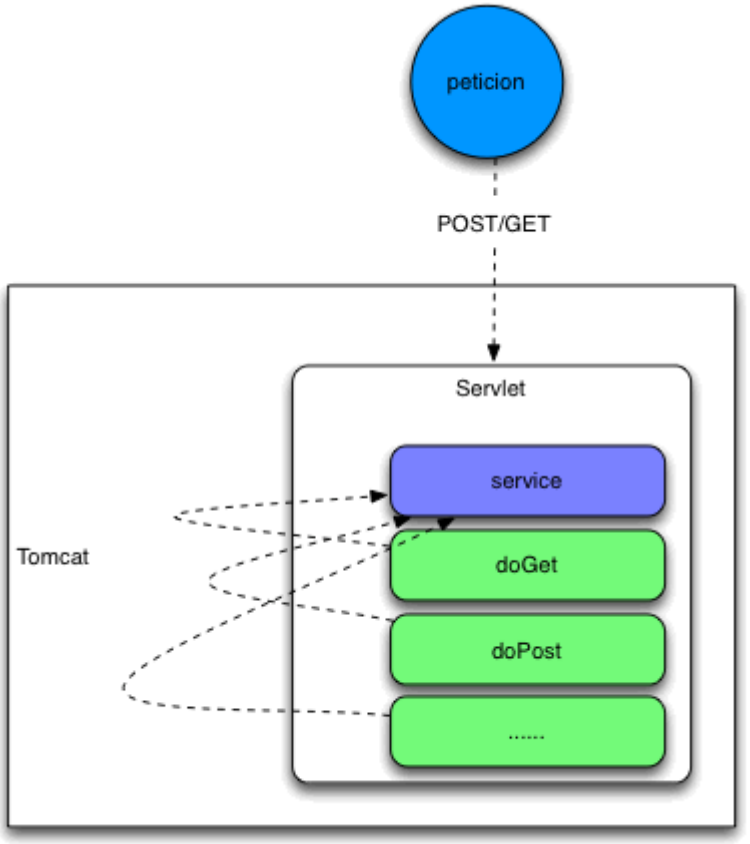
Introducción

- 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():



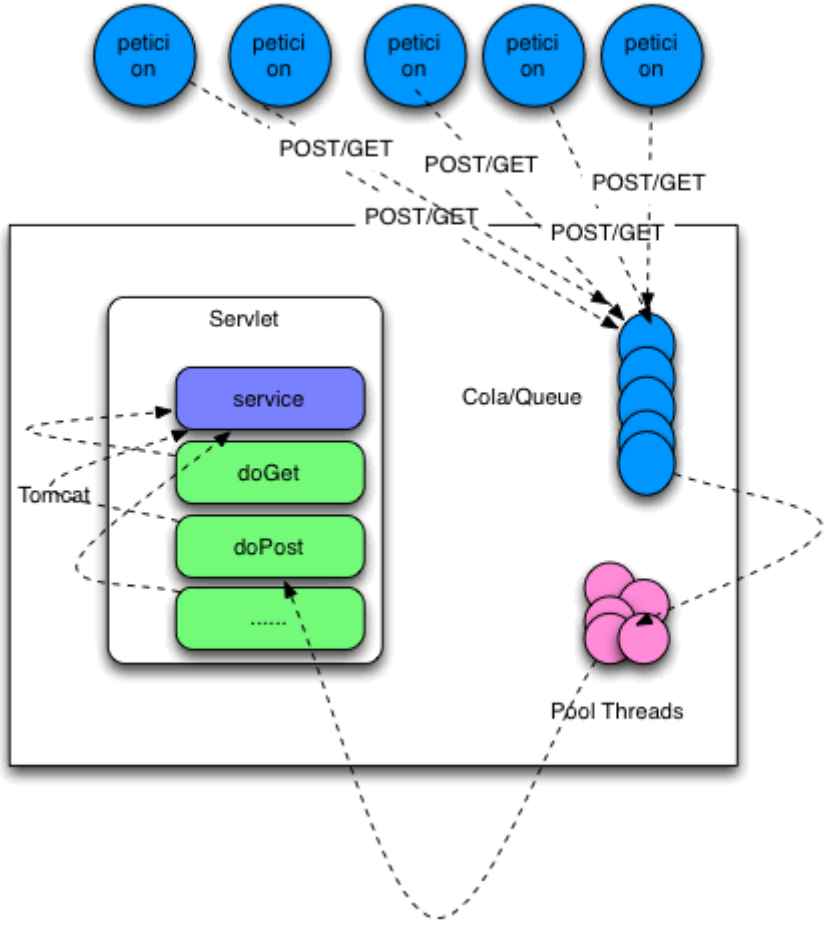
Introducción

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.



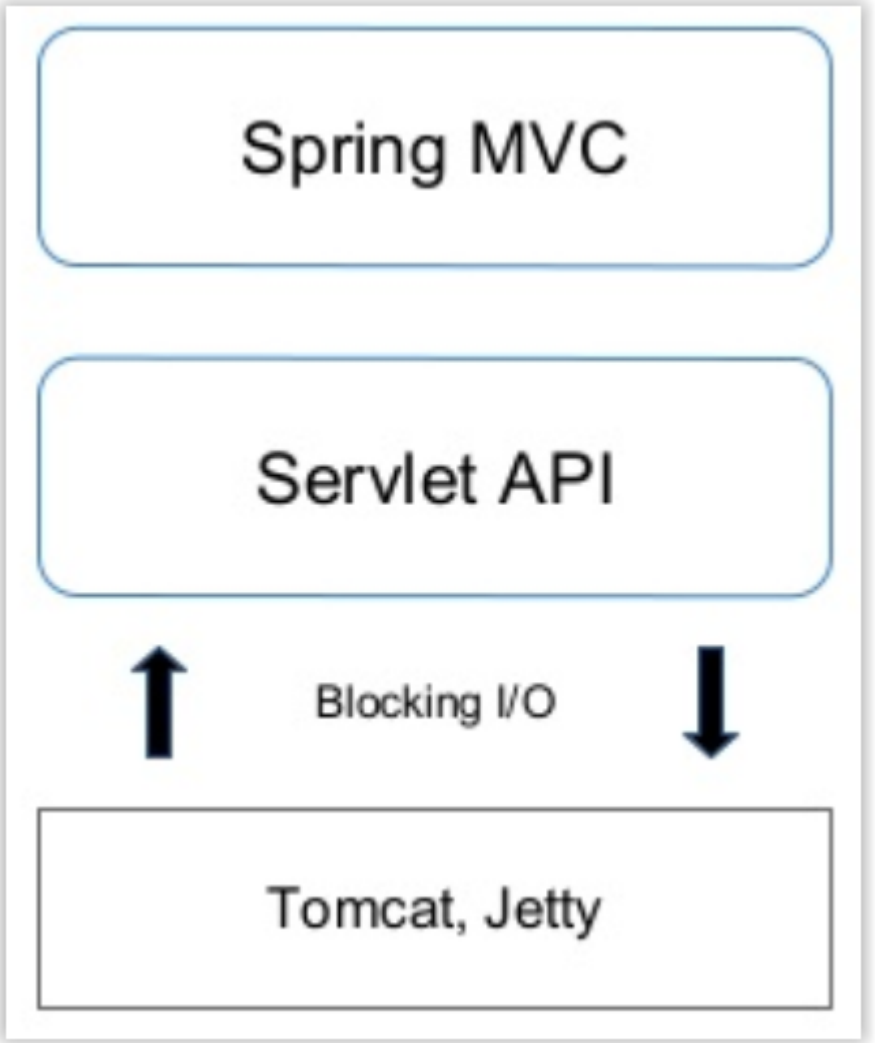
Introducción

- 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.



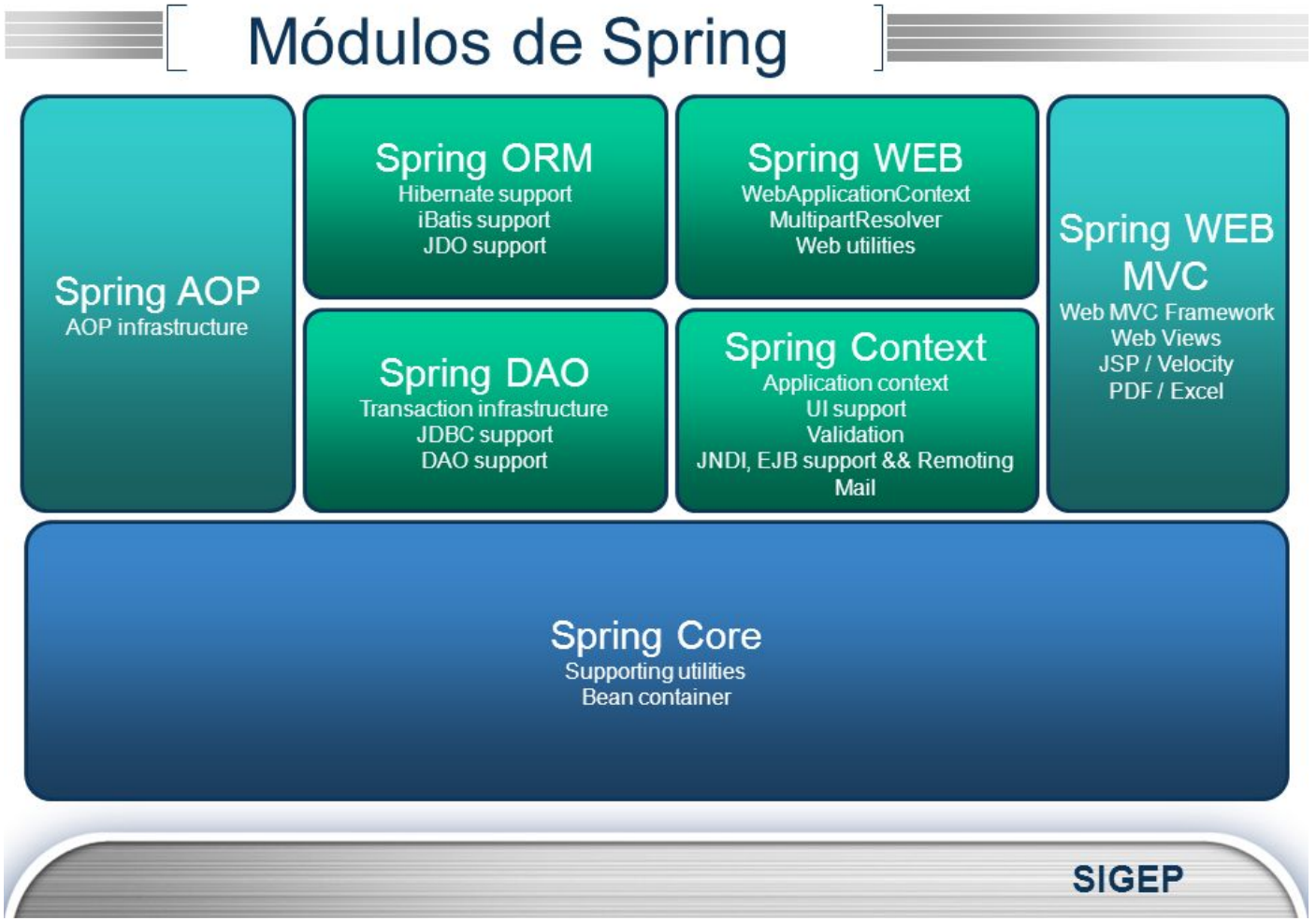
Introducción

- Arquitectura Spring MVC bloqueante



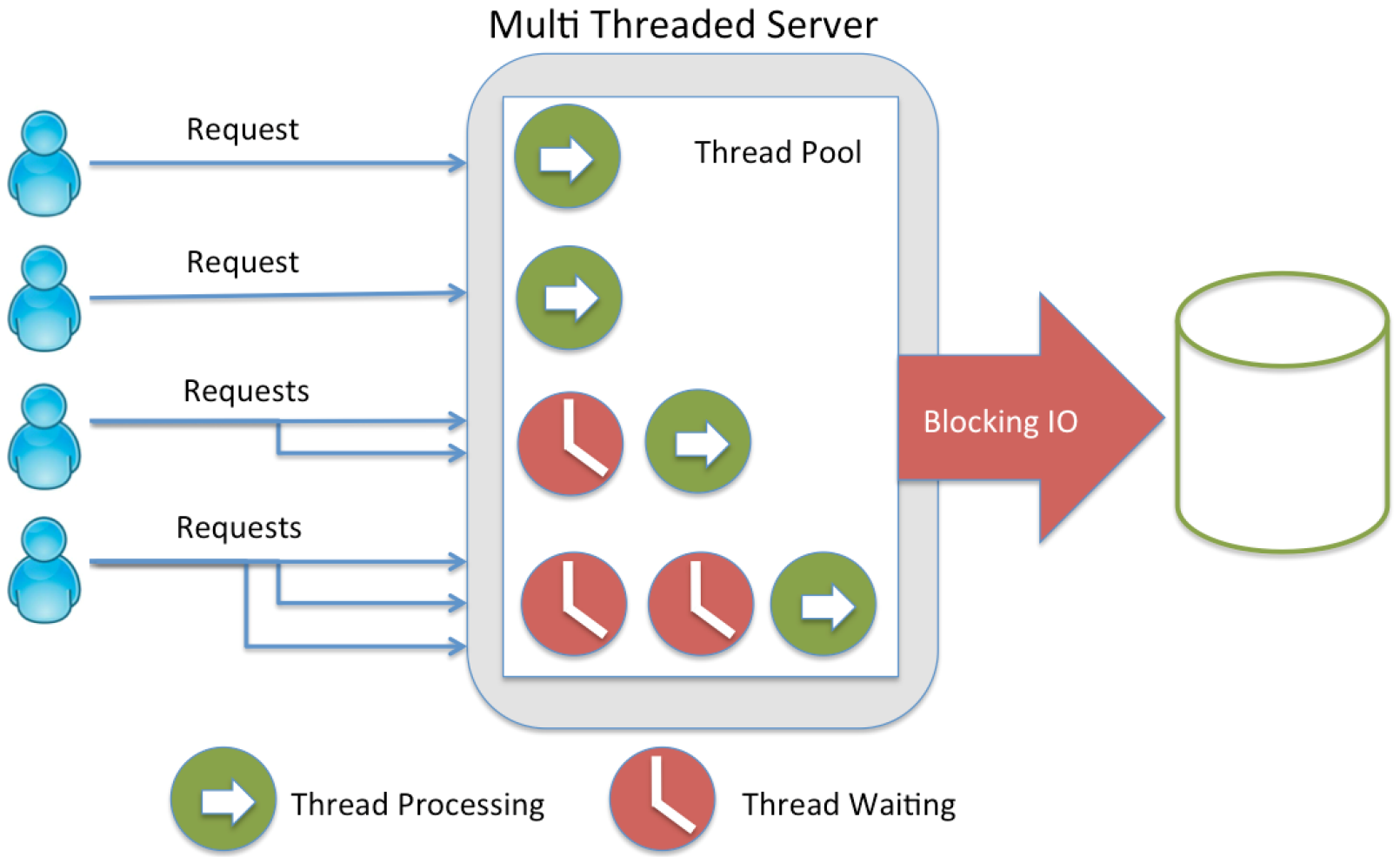
Introducción

- Arquitectura Spring MVC bloqueante



Introducción

- Arquitectura Spring MVC bloqueante

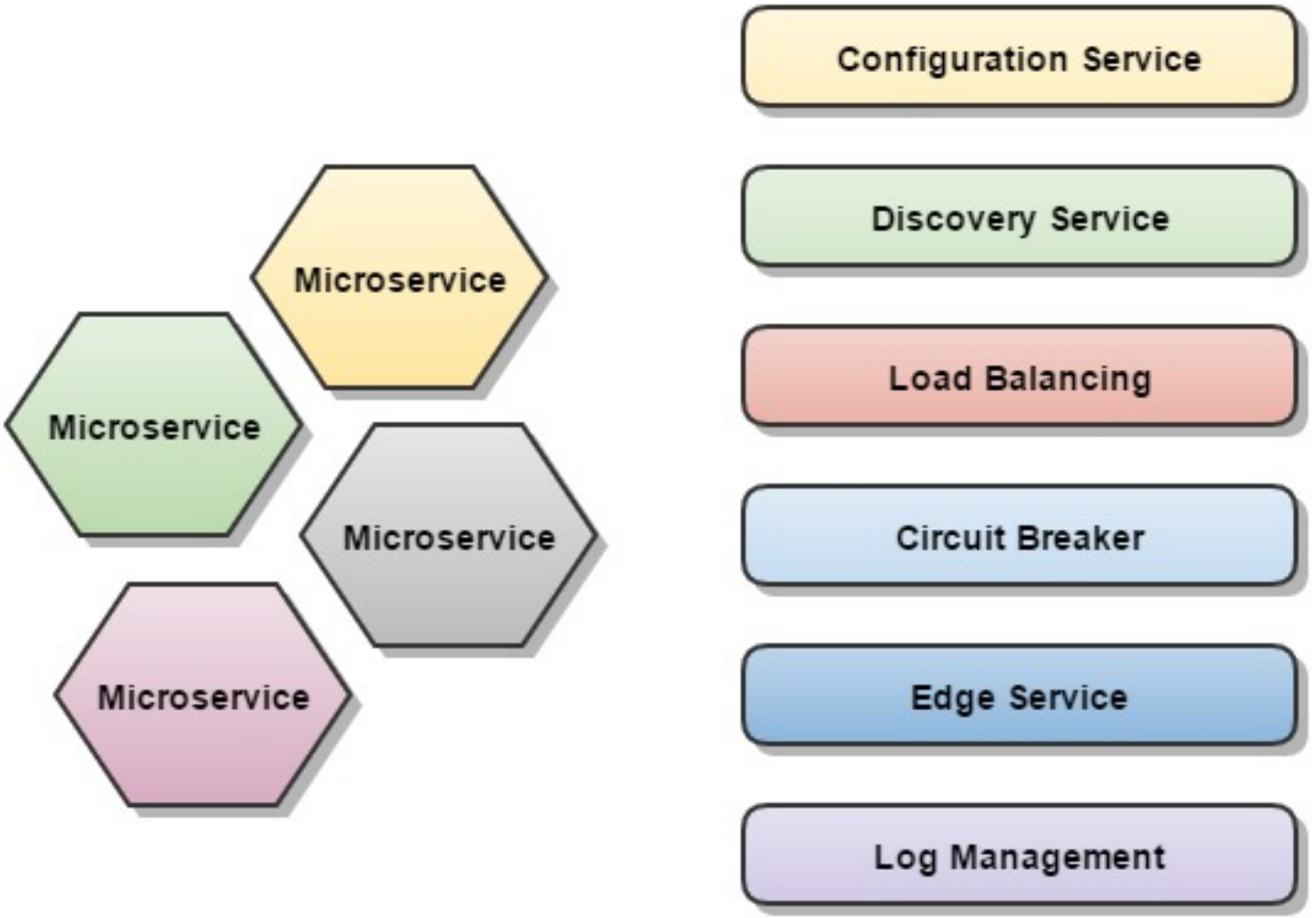


Introducción

- 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

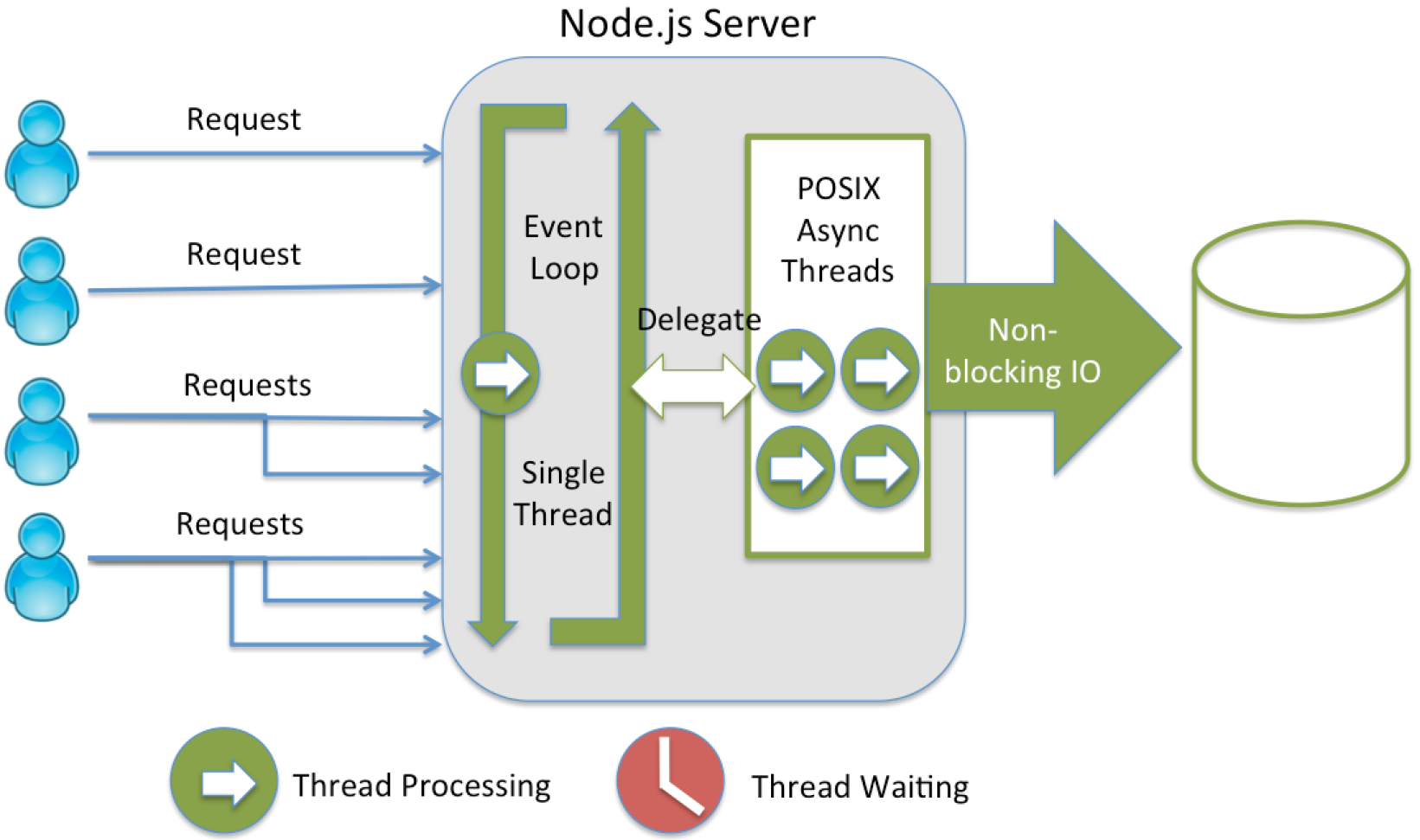
Introducción

- 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.

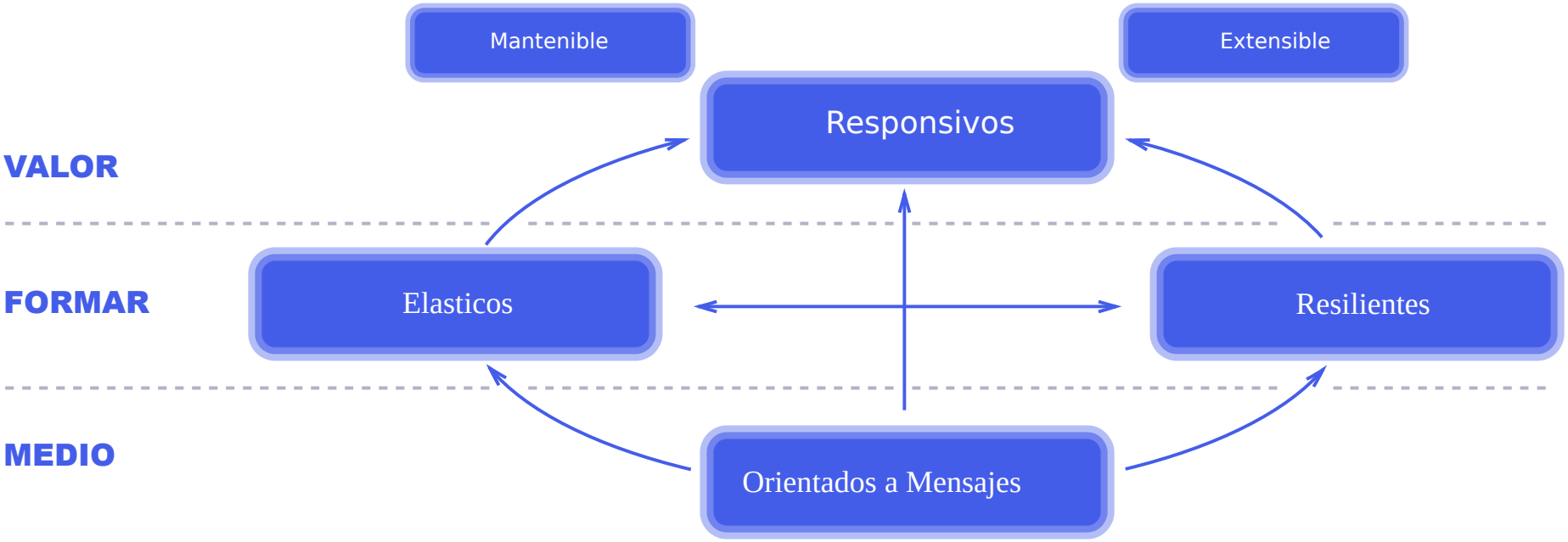
Manifiesto reactivo

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

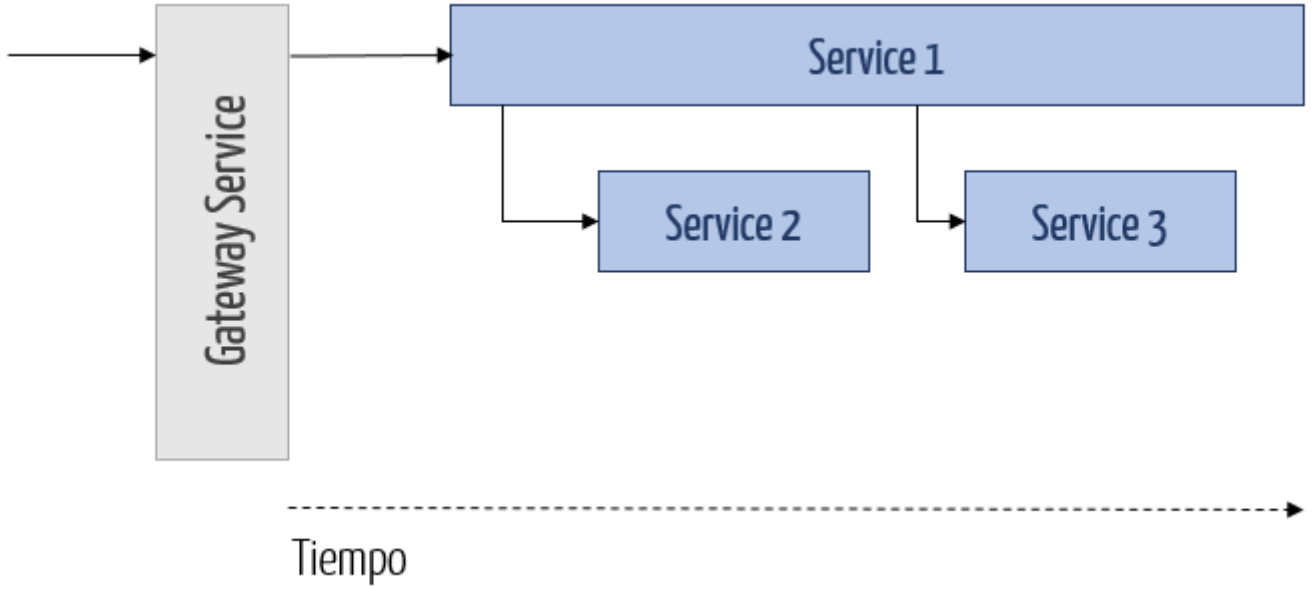
- La reactividad, al ser un paradigma (como la programación funcional), puede ser aplicada a más de un lenguaje.
- Tanto es así, que se han creado librerías para facilitar el desarrollo (y la vida de los desarrolladores), ahorrando tiempo, aumentando la productividad y construyendo sistemas más robustos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

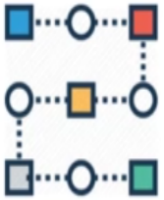
- Generación 0: java.util.Observable/Observer proporcionaban la base de uso del patrón Observer del Gang of Four. Tienen los inconvenientes de su simplicidad y falta de opciones de composición.

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

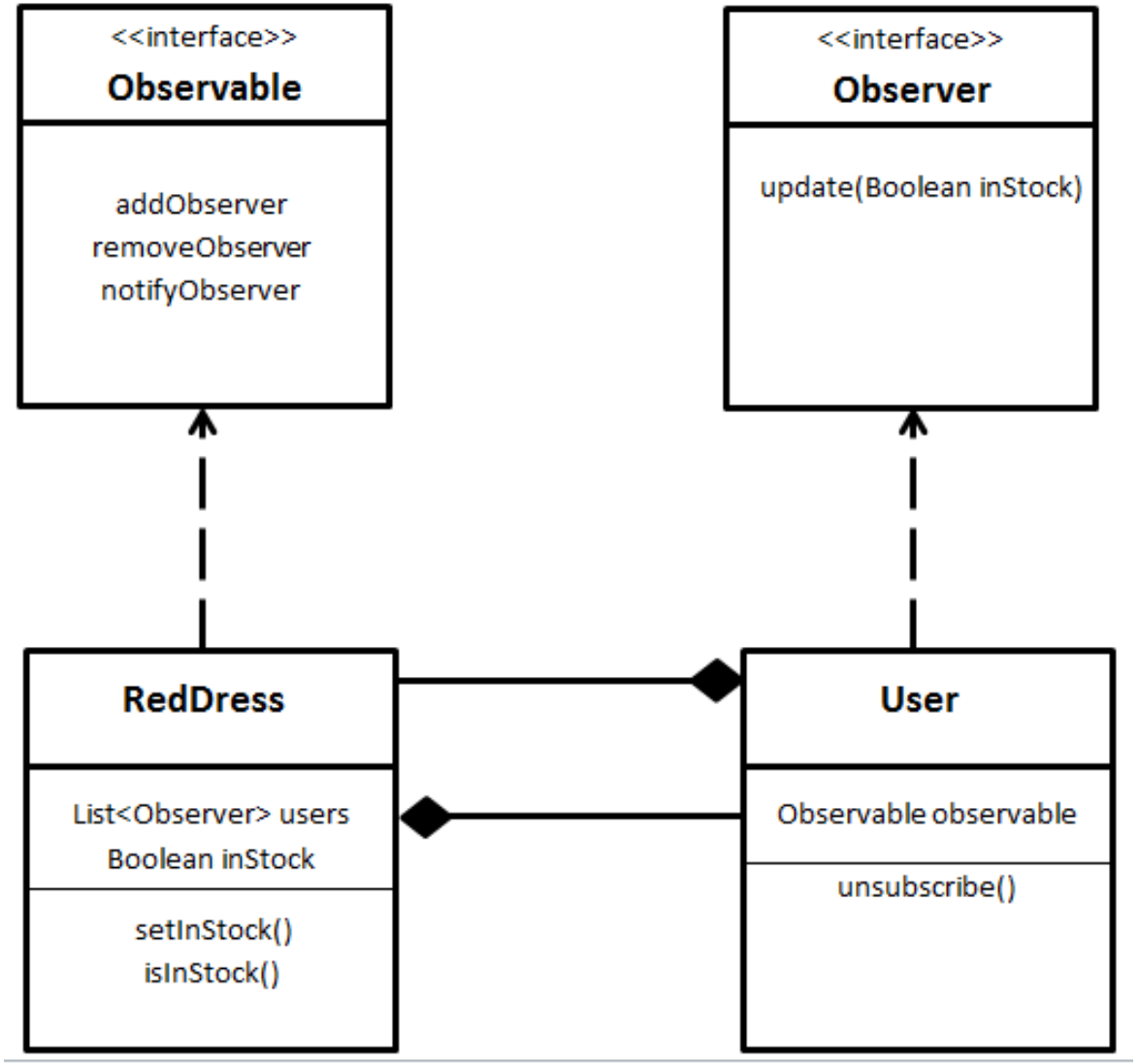
- La proramación reactiva toma lo mejor de patrones como el patrón observer, el patrón iterador, la arquitectura en pipeline y paradigmas como la programación funcional.
- Vamos a trabajar un poco con estos patrones para entenderlos bien, poder aplicarlos en otras situaciones antes de entrar de lleno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



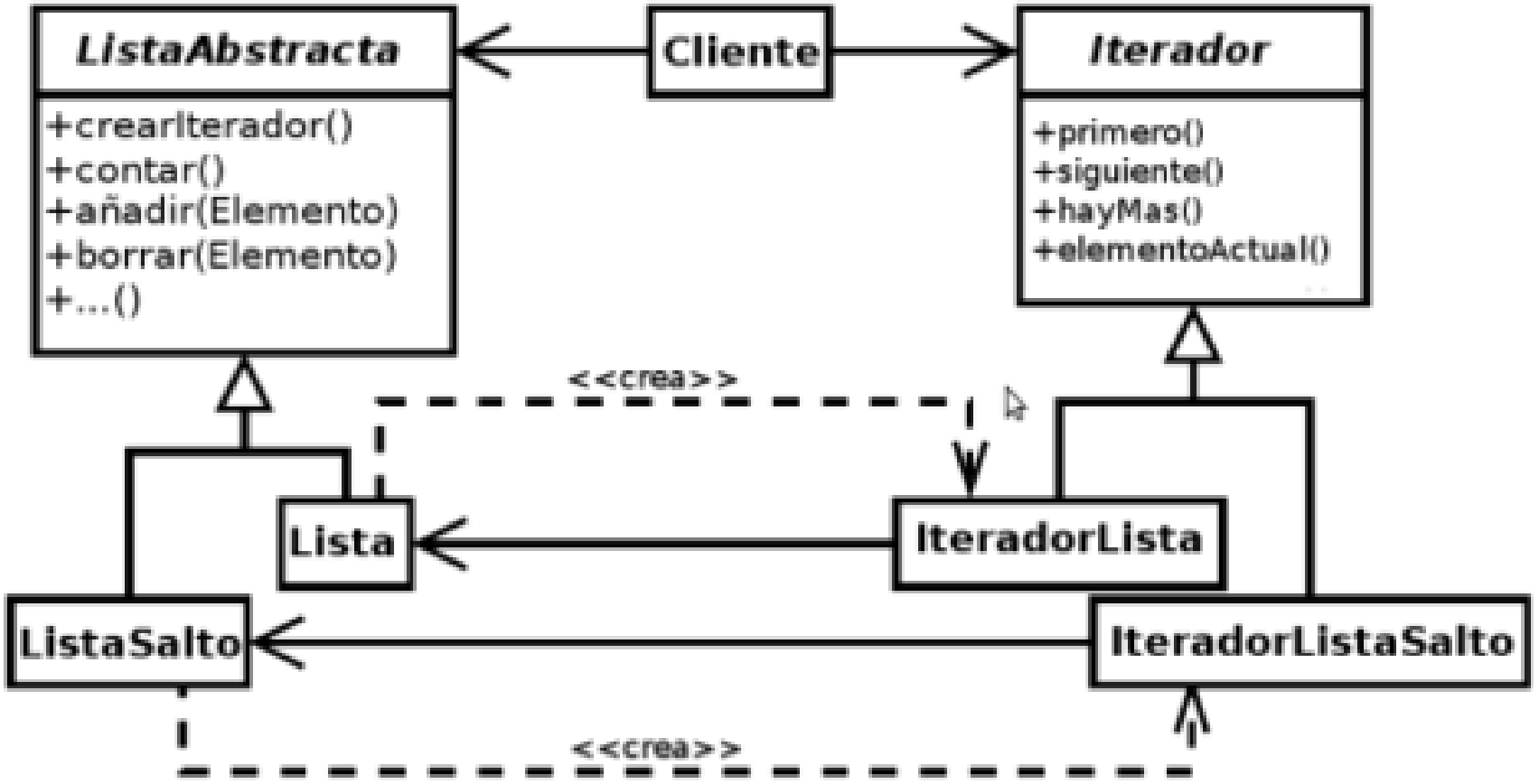
Veamos un ejemplo

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



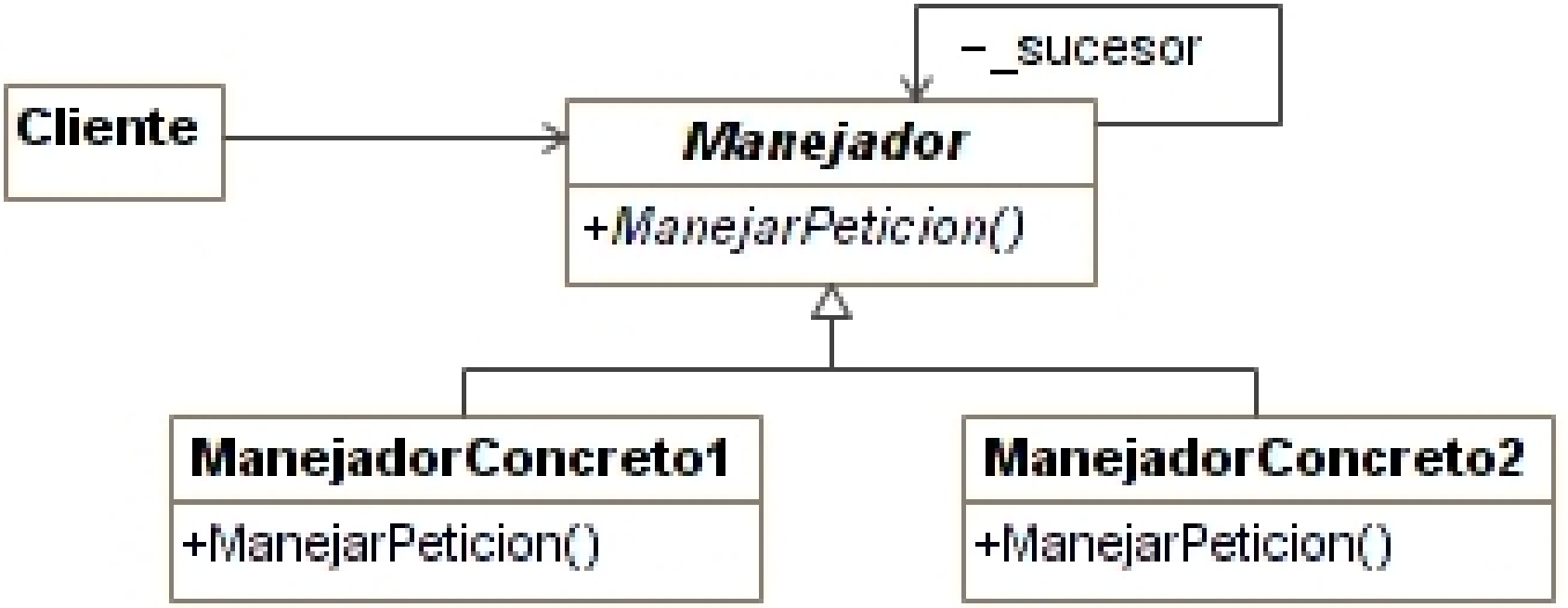
Veamos un ejemplo

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.

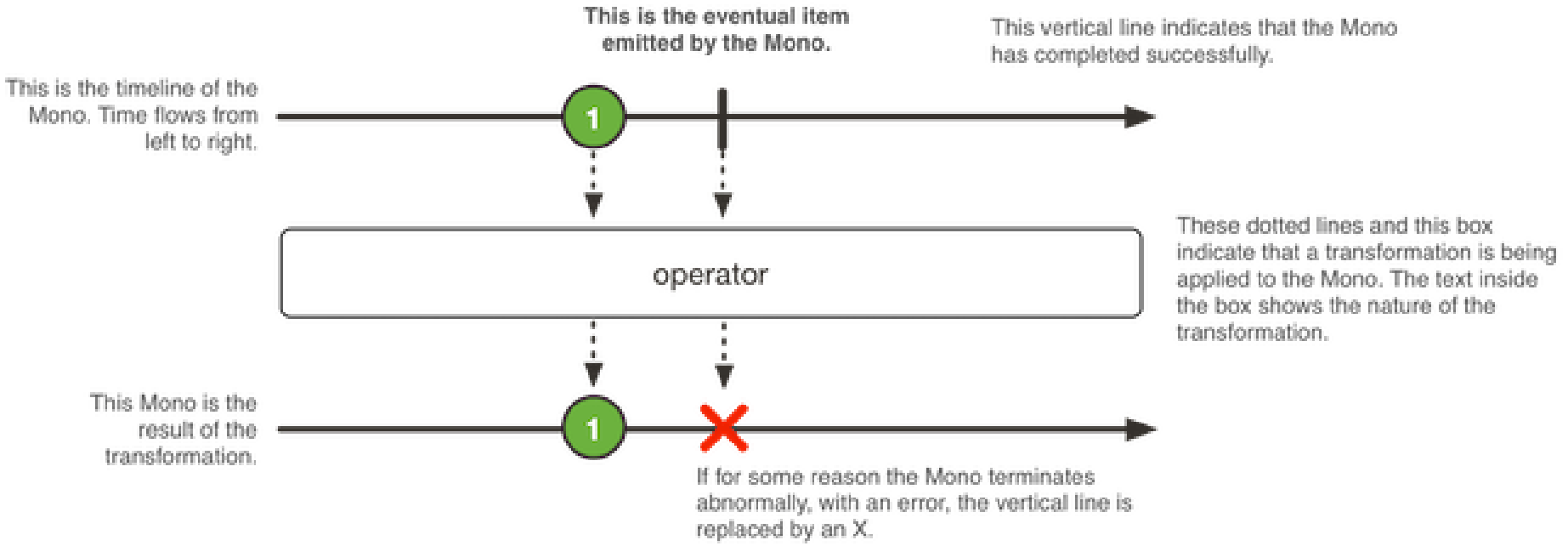
Veamos un ejemplo

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.

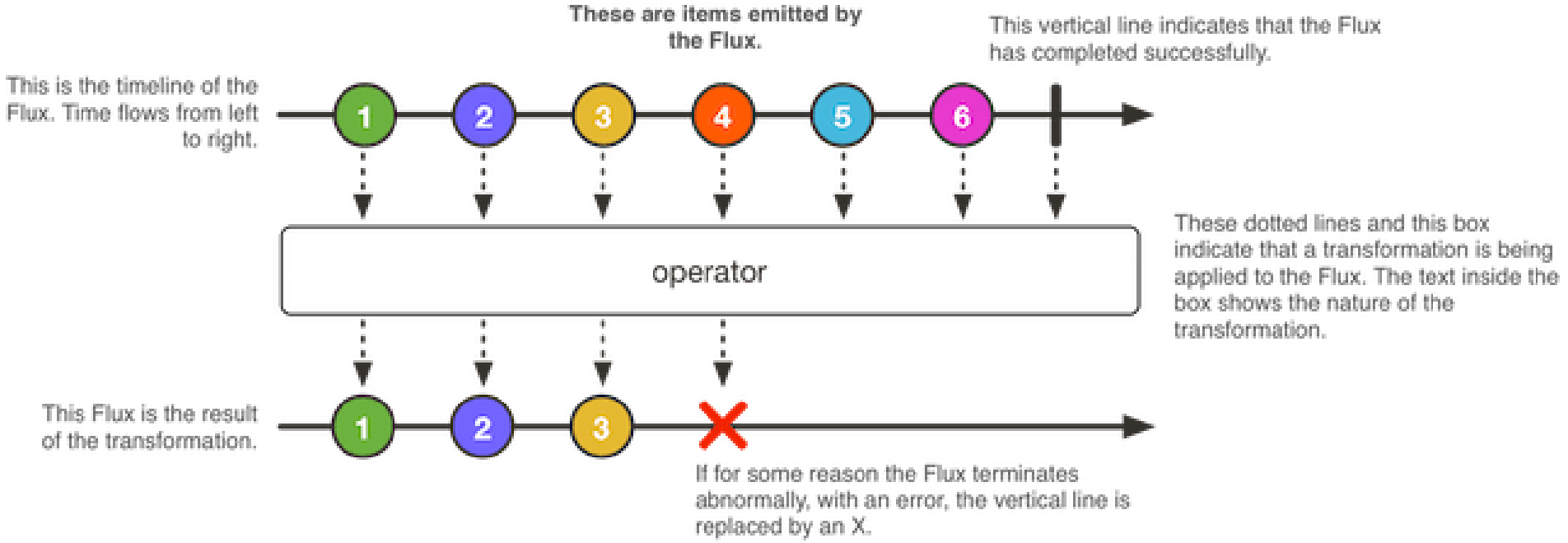
APIs Java de programación reactiva

- **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()
```

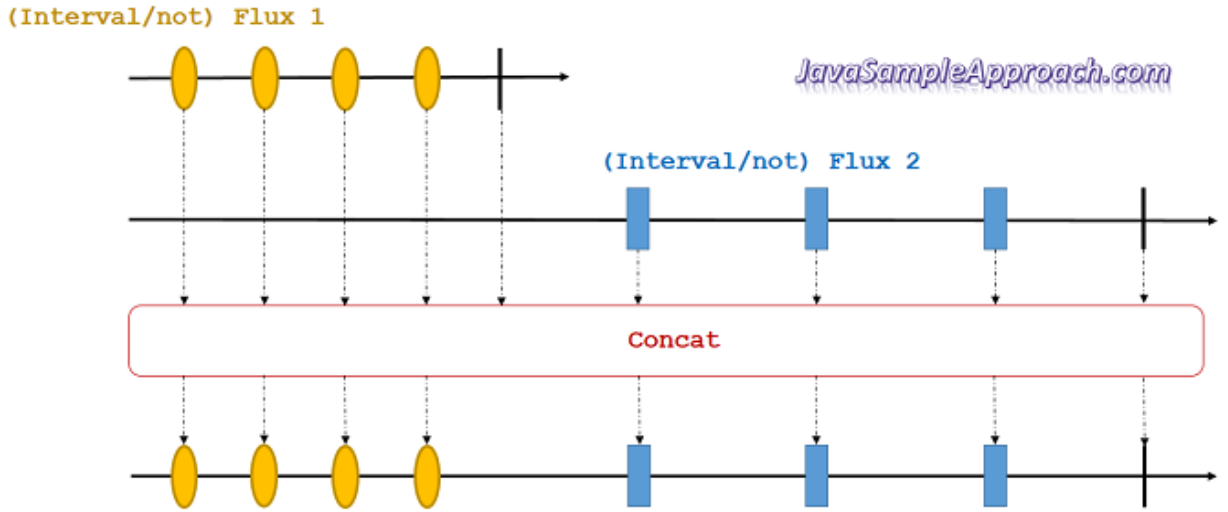
Veamos un ejemplo

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:

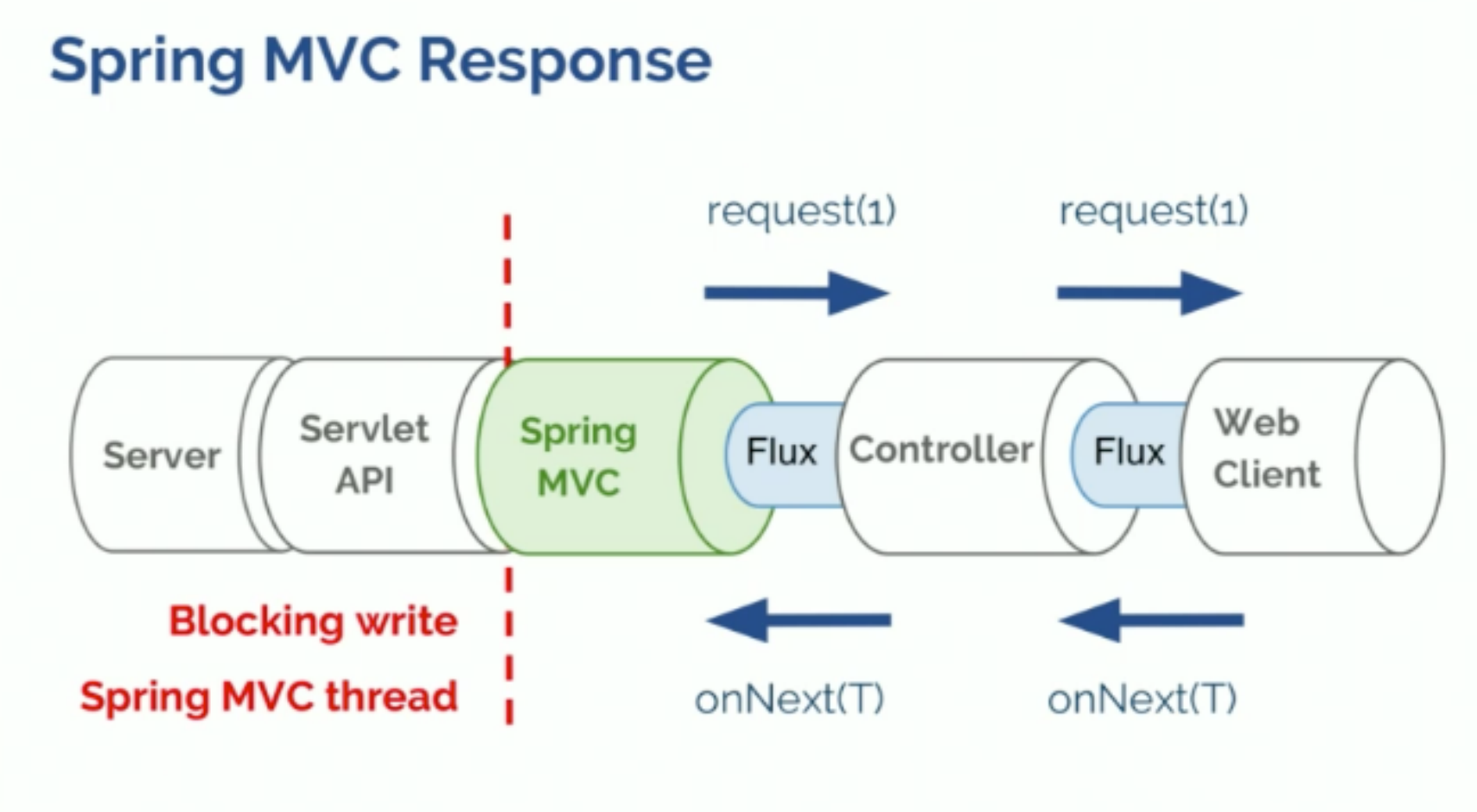



```
2  Flux.concat(mono1, mono3, mono2)
3          .subscribe(System.out::print);
4
5  Flux.concat(flux2, flux1)
6          .subscribe(System.out::print);
```

Veamos un ejemplo

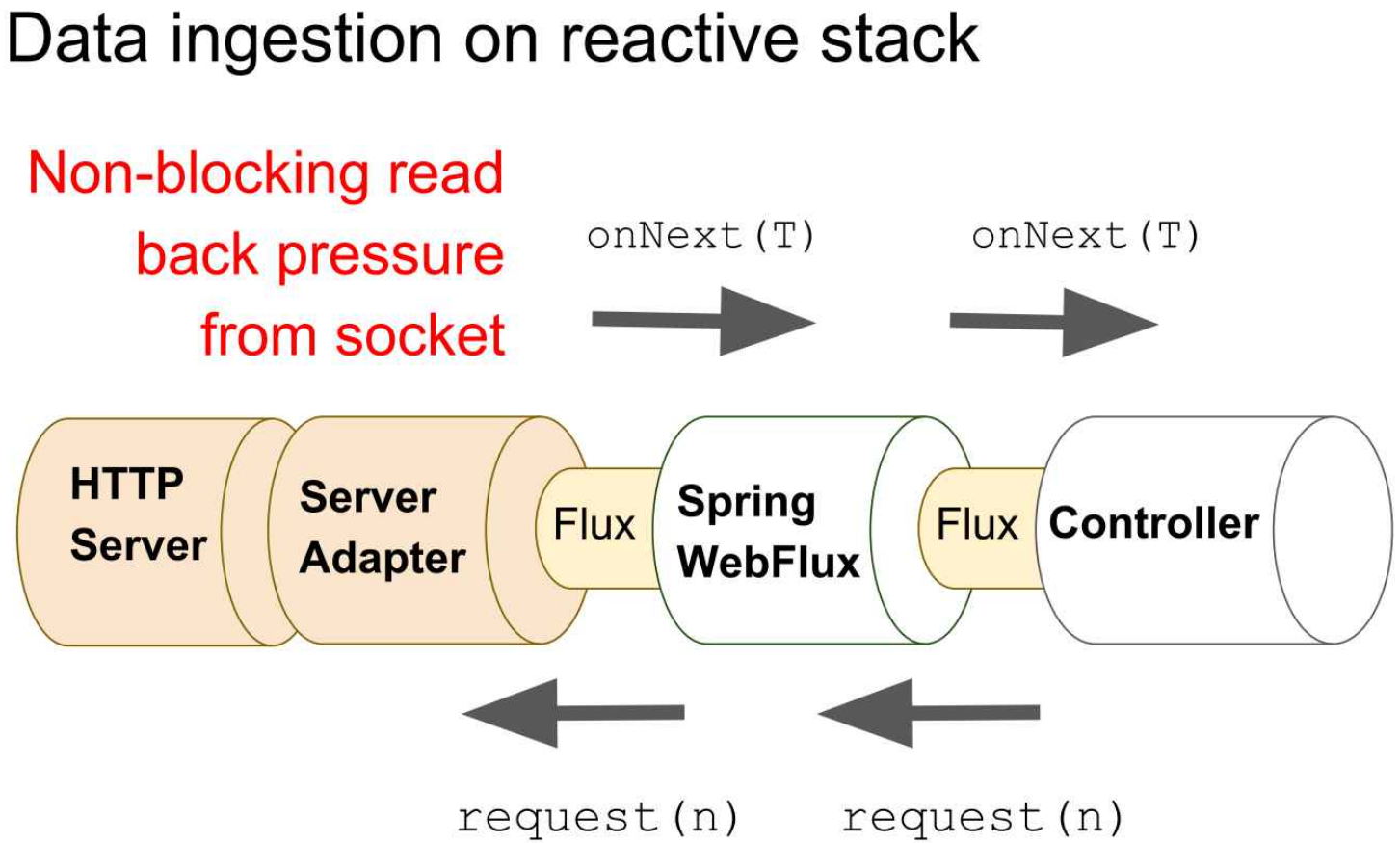
Arquitectura webflux

- Como era la arquitectura MVC Spring:

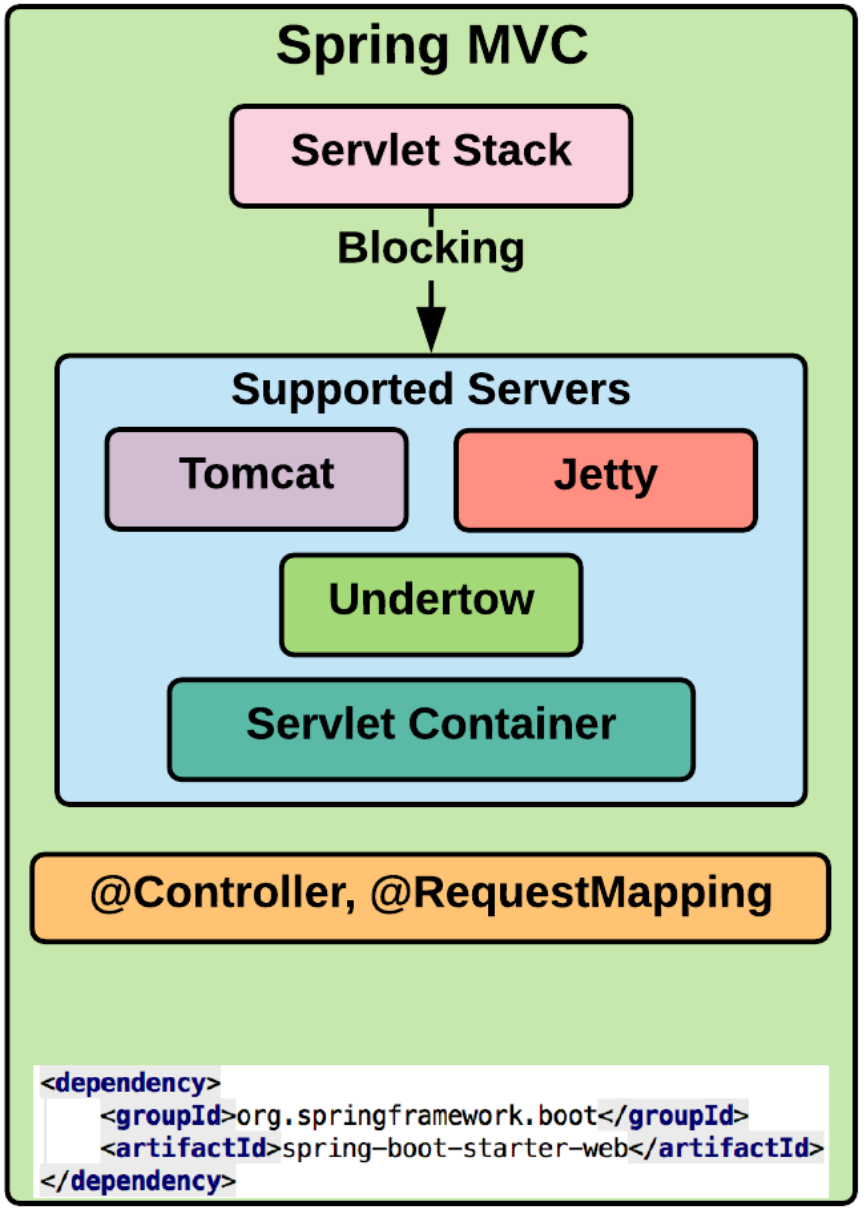


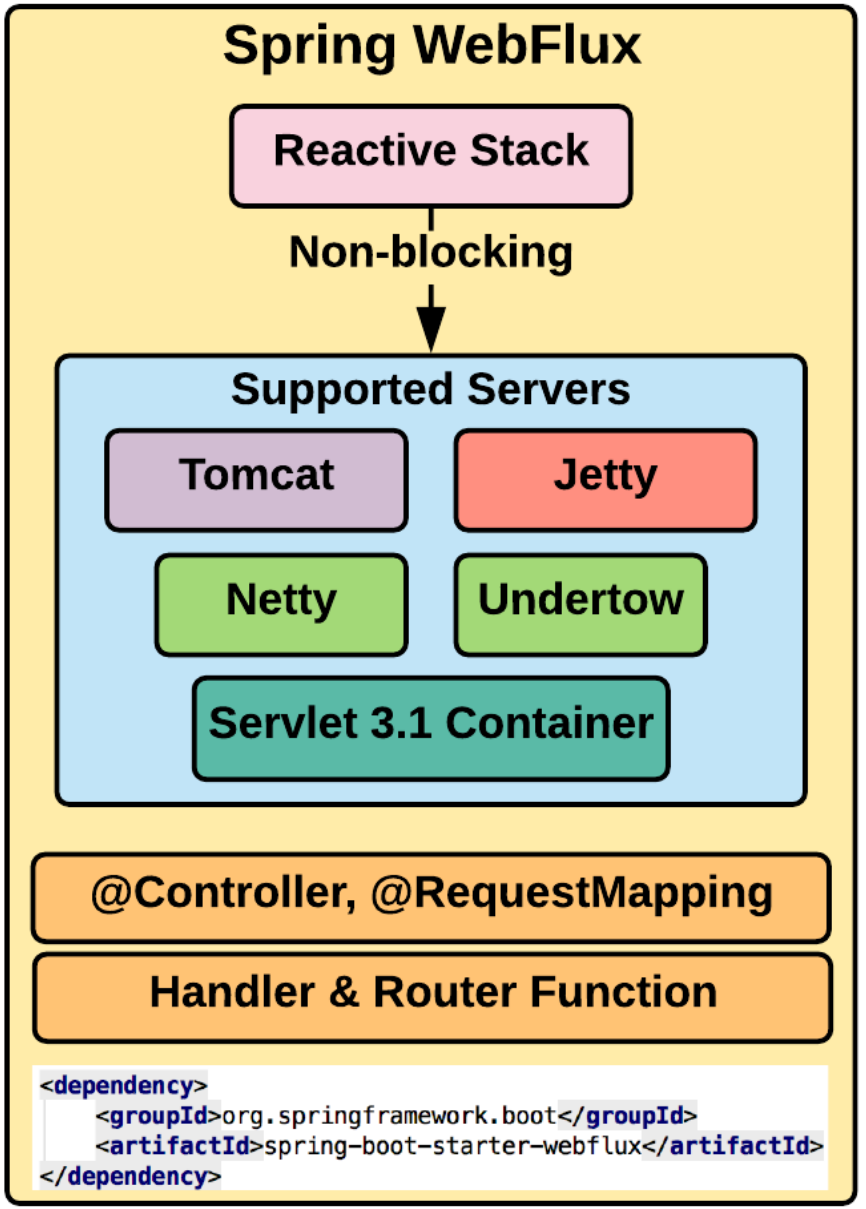
Arquitectura webflux

- Como es la arquitectura Spring WebFlux:

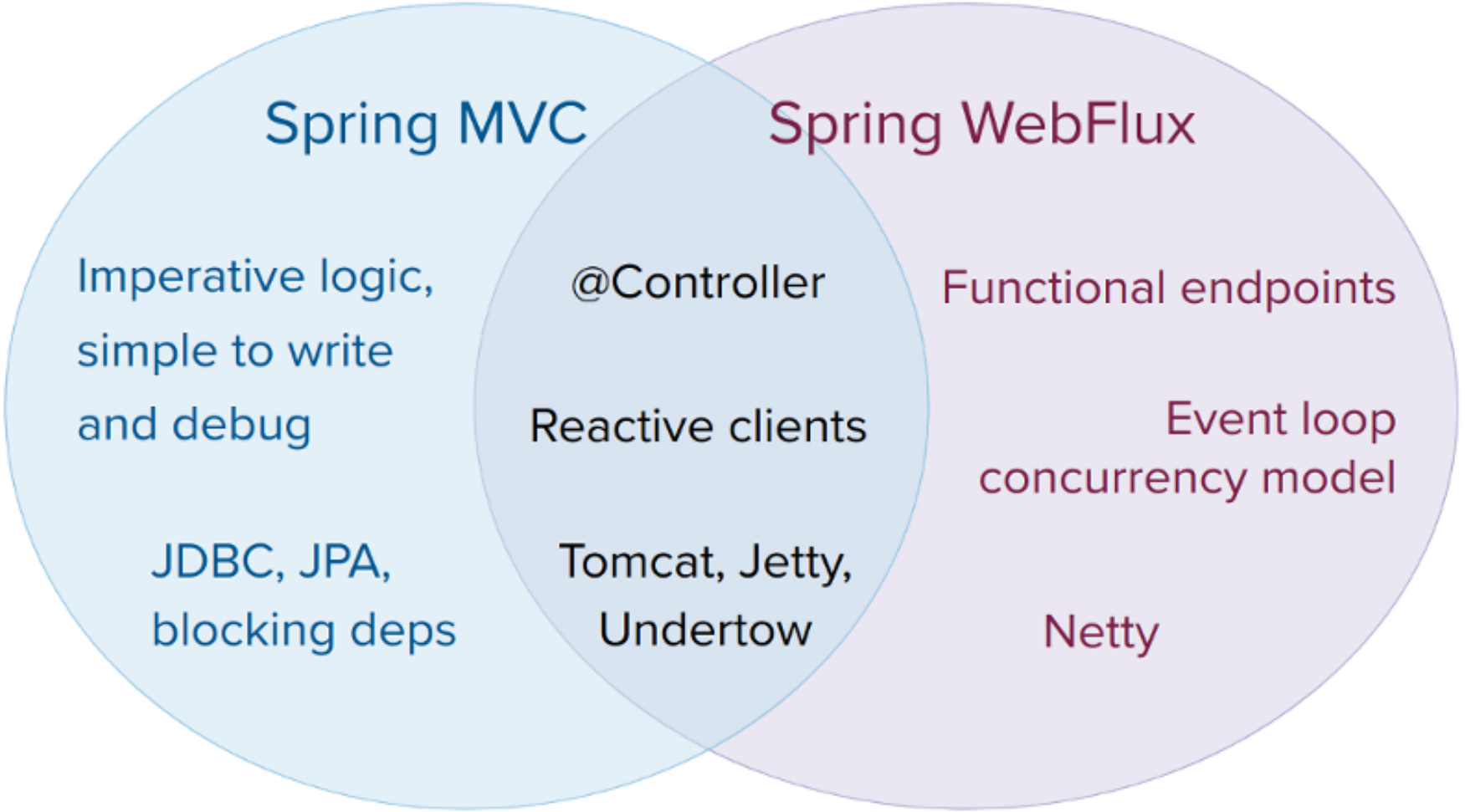


Arquitectura webflux

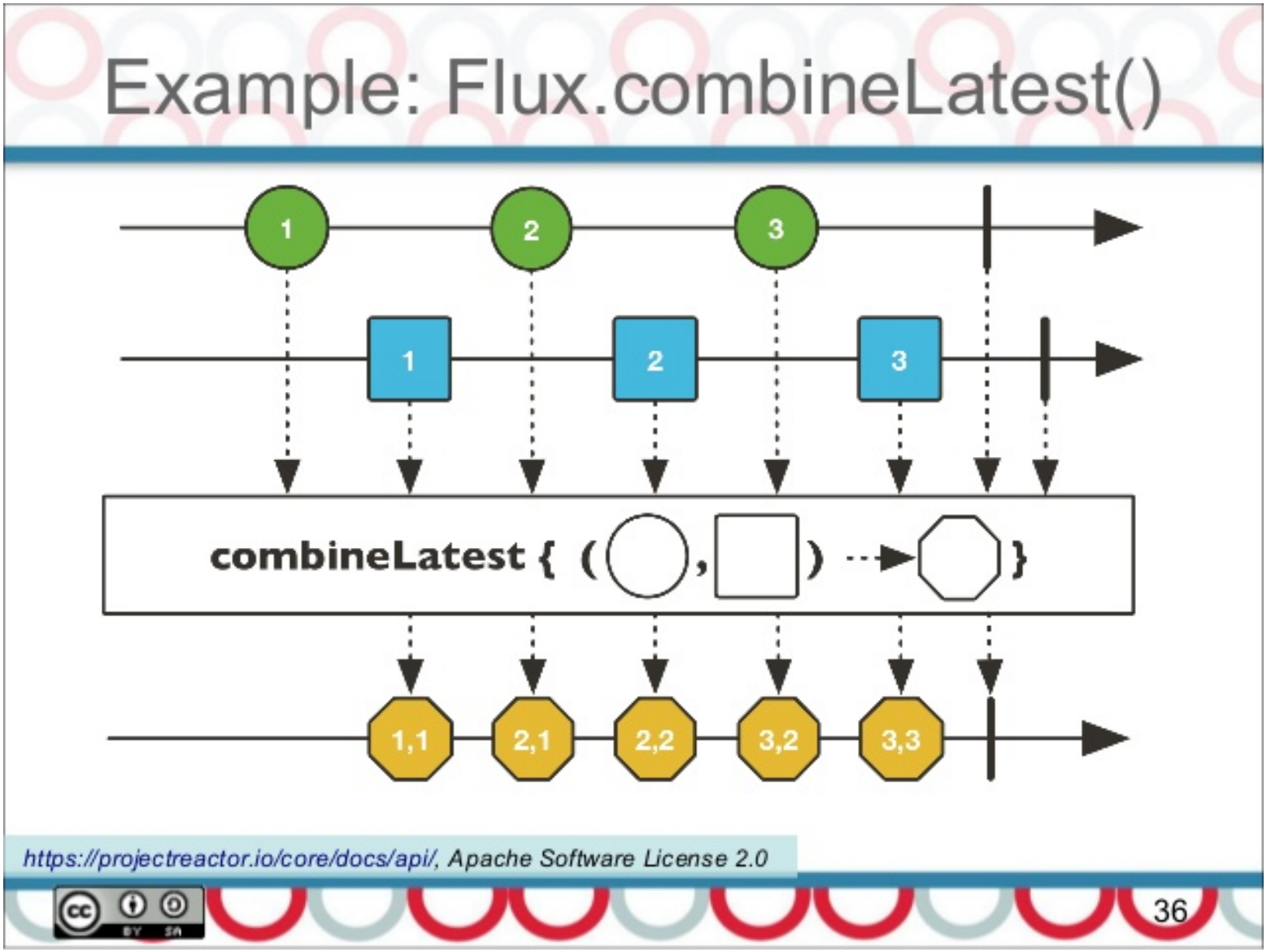




Arquitectura webflux

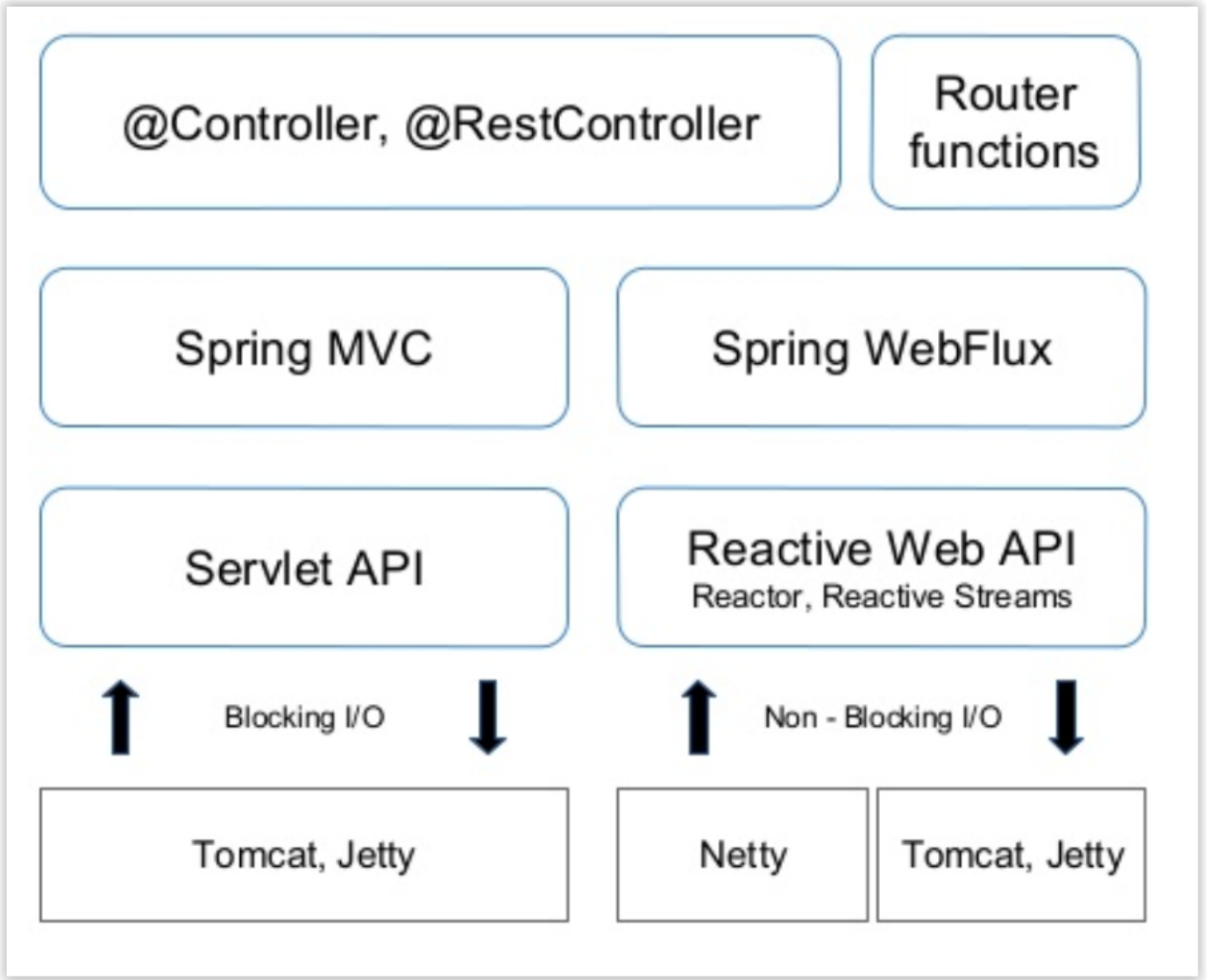


Arquitectura webflux



Arquitectura webflux

¿Pueden convivir Spring MVC y WebFlux?

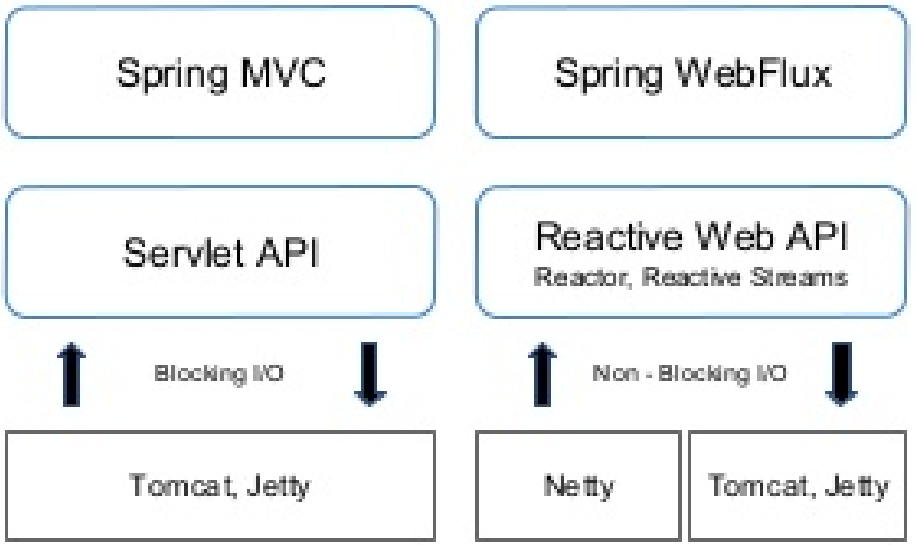


Arquitectura webflux

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

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
-