




Programación Reactiva en Java



© JMA 2021. All rights reserved

PROGRAMACIÓN REACTIVA

© JMA 2021. All rights reserved

Introducción

- La Programación Reactiva (o Reactive Programming) es un paradigma de programación asíncrono enfocado en el trabajo con flujos de datos finitos o infinitos y la propagación de los cambios. Estos flujos se pueden observar y reaccionar en consecuencia.
- Es una evolución o especialización del paradigma la programación orientada a eventos o dirigida por eventos, donde solo cuenta el evento del cambio del valor de un dato (la aparición del siguiente valor).
- Se basa en el principio de Hollywood «No nos llames, Nosotros te llamamos», llamado así porque es la frase habitualmente empleada por los directores de casting en Hollywood para evitar la saturación de estar recibiendo llamadas de aspirantes preguntando si han sido aceptados.
- El modelo de programación reactiva ha evolucionado de manera significativa desde su concepción en 2010. Su concepción y evolución ha ido ligada a la publicación del [Reactive Manifiesto](#).

© JMA 2021. All rights reserved

Manifiesto de Sistemas Reactivos

- 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. 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. 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.

© JMA 2021. All rights reserved

Manifiesto de Sistemas Reactivos

- Creemos que se necesita un enfoque coherente para la arquitectura de sistemas, y creemos que todos los aspectos necesarios ya han sido identificados por separado: queremos sistemas Responsivos, Resilientes, Elásticos y Orientados a Mensajes. Nosotros les llamamos Sistemas Reactivos.
- Los sistemas contruidos 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.

© JMA 2021. All rights reserved

Manifiesto de Sistemas Reactivos

- **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. Los sistemas responsivos se enfocan en proveer tiempos de respuesta rápidos y consistentes, estableciendo límites superiores confiables para así proporcionar una calidad de servicio consistente. Este comportamiento consistente, a su vez, simplifica el tratamiento de errores, aporta seguridad al usuario final y fomenta una mayor interacción.
- **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, aislando cada componente de los demás, y asegurando así que cualquier parte del sistema pueda fallar y recuperarse sin comprometer el sistema como un todo. La recuperación de cada componente se delega en otro componente (externo) y la alta disponibilidad se asegura con replicación allí donde sea necesario. El cliente de un componente no tiene que responsabilizarse del manejo de sus fallos.

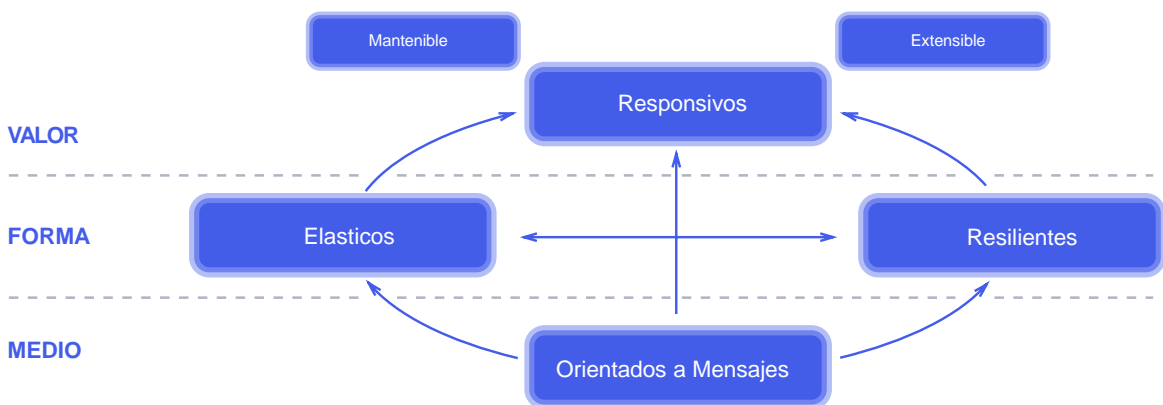
© JMA 2021. All rights reserved

Manifiesto de Sistemas Reactivos

- **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. Los Sistemas Reactivos soportan algoritmos de escalado predictivos, así como Reactivos, al proporcionar relevantes medidas de rendimiento en tiempo real. La elasticidad se consigue de forma rentable haciendo uso de plataformas con hardware y software genéricos.
- **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. La mensajería basada en ubicaciones transparentes como medio de comunicación permite que la gestión de fallos pueda trabajar con los mismos bloques y semánticas a través de un cluster o dentro de un solo nodo. La comunicación No-bloqueante permite a los destinatarios consumir recursos sólo mientras estén activos, llevando a una menor sobrecarga del sistema.

© JMA 2021. All rights reserved

Manifiesto de Sistemas Reactivos



© JMA 2021. All rights reserved

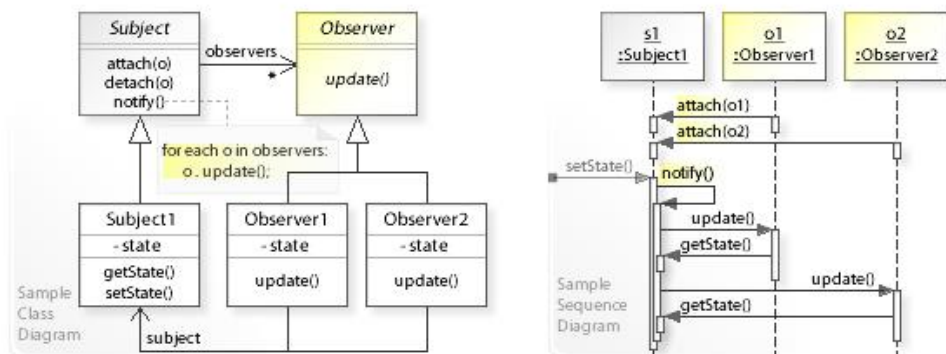
Fundamentos

- Un Observable o publicador es el propietario del dato y el mecanismo que, cada vez que cambia el dato, notifica los cambios producidos, creando un flujo de datos. Los Observables se basan en dos patrones de programación bien conocidos como son el patrón “Observer” y el patrón “Iterator”.
- Un Observador o suscriptor esta interesado en enterarse de los cambios que se producen en un flujo de datos. Los observadores se subscriben a los Observables para recibir notificaciones de los cambios. Por lo tanto, en lugar de sondear consultando si hay cambios, los eventos se realizan de forma asíncrona notificando cuando se producen los cambios para que los observadores puedan procesarlos.
- Un Observable puede tener tantos suscriptores como sean necesario.
- Para hacerlo simple, podríamos decir que un Observable es un objeto que guarda un valor y que emite un evento a todos sus suscriptores cada vez que ese valor se actualiza.

© JMA 2021. All rights reserved

Patrón Observer

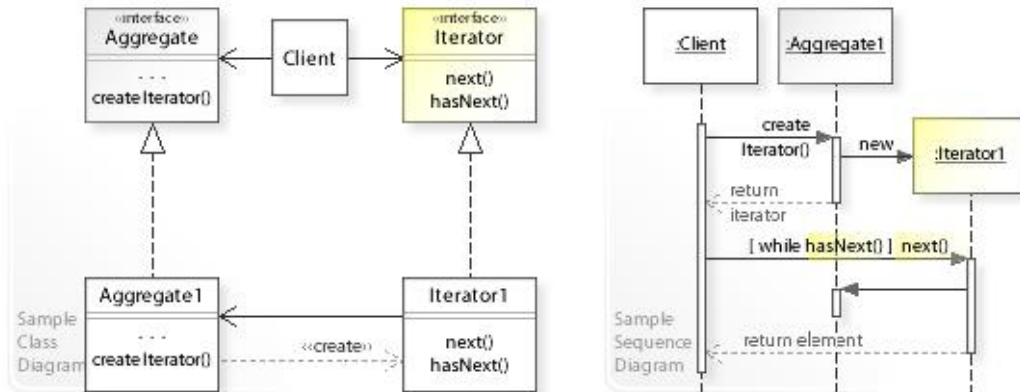
Observer es un patrón de diseño de comportamiento que permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



© JMA 2021. All rights reserved

Patrón Iterator

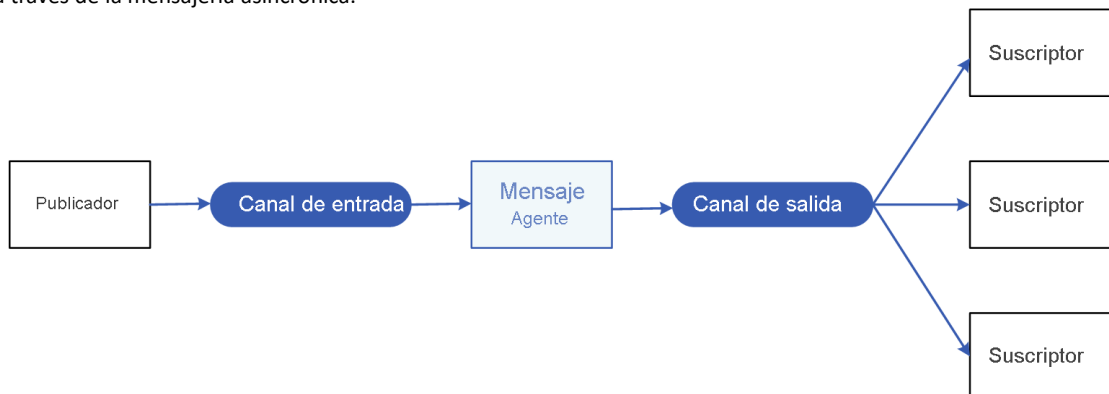
Iterator es un patrón de diseño de comportamiento que permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).



© JMA 2021. All rights reserved

Patrón de publicador y suscriptor

El patrón de publicador y suscriptor se basa en el patrón de observador al desacoplar los sujetos de los observadores a través de la mensajería asíncrona.



© JMA 2021. All rights reserved

Implementaciones

- El modelo de programación reactiva ha evolucionado de manera significativa desde su concepción en 2010. Su concepción y evolución ha ido ligada a la publicación del Reactive Manifesto.
- 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.
 - 1ª Generación: en 2010 Microsoft publica `RX.NET`, que en 2013 sería portado a entorno Java a través de la librería `RxJava`. Evolucionó el concepto de `Observable/Observer` incorporando la composición de operaciones, pero presentaba limitaciones arquitectónicas.
 - 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 reactivo 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`.

© JMA 2021. All rights reserved

Implementaciones

- Adicionalmente las librerías reactivas se han ocupado de mejorar los siguientes aspectos:
 - Composición y legibilidad: hasta Java 8 la única manera de trabajar con operaciones asíncronas consistían en el uso de `callbacks`, `Future` o `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.
 - `Backpressure`: debido al flujo `push-based`, puede que un `Publisher` genere más elementos de los que un `Subscriber` puede consumir. Para evitarlo:
 - 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.).

© JMA 2021. All rights reserved

PROGRAMACIÓN MULTITHREAD

© JMA 2021. All rights reserved

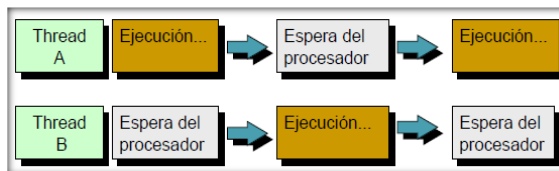
Introducción

- Multitarea, posibilidad de que un ordenador realice varias tareas a la vez.
- La Multitarea real se consigue cuando disponemos de múltiples CPU, sino será “emulada”.
- Esta emulación se consigue repartiendo tiempo y recursos entre distintos procesos.
- La palabra thread (hilo) hace referencia a un flujo de control dentro de un programa.
- Dentro de un programa pueden ejecutarse varios hilos a la vez.
- Los hilos comparten:
 - Los datos del programa (además pueden tener datos propios)
 - El entorno de ejecución
- Un Hilo puede ser el recolector de basura de Java que elimina los datos no deseados mientras el programa continúa con su ejecución normal. El uso de hilos es muy variado
 - Animación
 - Creación de servidores
 - Tareas de segundo plano
 - Programación paralela,...

© JMA 2021. All rights reserved

Introducción

- La máquina virtual Java implementa un entorno que permite a varias secuencias de instrucciones ejecutarse al mismo tiempo.
- En Java, los threads no se ejecutan necesariamente al mismo tiempo.
- En realidad, el interprete de Java puede ejecutar los threads por rodajas de tiempo (restricción impuesta en los sistemas monoprocesador).
- Este tipo de ejecución está basada en las posibilidades del planificador del Sistema Operativo.



© JMA 2021. All rights reserved

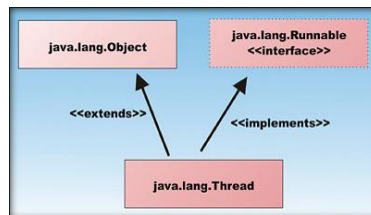
Introducción

- Cuando se ejecuta un programa Java, se lanza automáticamente el hilo PRINCIPAL.
- Es el hilo desde el cual se lanzan los demás hilos y debe de ser el último en terminar → Final del Programa.
- El hilo principal puede ser controlado a través de un Objeto de tipo Thread de la siguiente forma:
 - `Thread t = Thread.currentThread();`
- A partir de este momento este hilo se puede gestionar igual que los demás.

© JMA 2021. All rights reserved

Clase Thread y la interfaz Runnable

- La interfaz `java.lang.Runnable` permite definir las operaciones que realiza cada thread.
- Esta interfaz se define con un solo método público llamado `run` que puede contener cualquier código y que será el código que se ejecutará cuando se lance el thread.
- Para que una clase realice tareas concurrentes, basta con implementar `Runnable` y programar el método `run`.



© JMA 2021. All rights reserved

Clase Thread y la interfaz Runnable

- La clase `Thread`, implementa el interface `Runnable`, por lo que es posible crear objetos cuyo código se ejecute en un hilo aparte.
- Esta clase permite:
 - Iniciar
 - Controlar
 - Detener hilos de programa.
- Un nuevo thread se crea con un nuevo objeto de la clase `java.lang.Thread` o cualquier otra clase que implemente el interface `Runnable`.
 - `hilo = new Thread(objetoRunnable);`
 - `hilo.start();` //Se ejecuta el método `run` del objeto `Runnable`

© JMA 2021. All rights reserved

Creación con Runnable

- La interface Runnable proporciona un método alternativo a la herencia de la clase Thread, para los casos en los que no es posible hacer que nuestra clase extienda la clase Thread.
- Esto ocurre cuando nuestra clase, que deseamos correr en un thread independiente, deba extender alguna otra clase.
- Dado que no existe herencia múltiple, nuestra clase no puede extender a la vez la clase Thread y otra más.
- En este caso nuestra clase debe implantar la interface Runnable, variando ligeramente la forma en que se crean e inician los nuevos threads.

© JMA 2021. All rights reserved

Creación con Runnable

- Para poder ejecutar los threads necesitamos realizar los siguientes pasos:
 - Crear una clase que implemente Runnable
 - Definir el método run con las acciones que tomará el hilo de programa

```
class Movimiento implements Runnable{
    public void run() {
        while (true) {
            //código
        }
    }
}
```
 - Crear un objeto Thread tomando como parámetro un objeto de la clase anterior

```
Movimiento a1 = new Movimiento();
Thread thread1 = new Thread(a1,"Nombre");
```
 - Ejecutar el método start del Thread para ejecutar el método run

```
thread1.start();
```

© JMA 2021. All rights reserved

Creación con Runnable

```
public class HiloRunnable implements Runnable {  
    private int hijo = 0;  
    private int iter = 0;  
    private int sleep = 0;  
    private Thread t;  
  
    public HiloRunnable(int hijo, int iter, int sleep) {  
        super();  
        this.hijo = hijo;  
        this.iter = iter;  
        this.sleep = sleep;  
    }  
  
    public HiloRunnable(int hijo) {  
        this(hijo, 5, 500);  
    }  
}
```

→

© JMA 2021. All rights reserved

Creación con Runnable

←

```
@Override  
public void run() {  
    try {  
        for (int i = 0; i < iter; i++) {  
            System.out.printf("Hilo: %s Paso: %d de %d\n", Thread.currentThread().getName(), i+1, iter);  
            Thread.sleep(sleep);  
        }  
    } catch (Exception e) {  
        System.out.printf("Error en el hijo %d\n", hijo);  
    }  
    System.out.printf("Termina Hilo: %d\n", hijo);  
}  
  
public void lanzar() {  
    t = new Thread(this, "Hijo " + hijo);  
    System.out.println("Lanzo " + t.getName());  
    t.start();  
}  
}
```

© JMA 2021. All rights reserved

Pasos de creación mediante herencia

- Es la opción más utilizada, pues como la clase Thread implementa la interfaz Runnable, ya cuenta con el método run.
- Pasos:
 - Crear una clase basada en Thread
 - Definir el método run y en él las acciones que tomará el hilo de programa

```
class Movimiento extend Thread{
    public void run() {
        while (true) {
            //código
        }
    }
}
```
 - Crear un objeto de la clase

```
Movimiento a1 = new Movimiento();
```
 - Ejecutar el método start

```
a1.start();
```

© JMA 2021. All rights reserved

Creación mediante herencia

```
public class HiloHerencia extends Thread {
    private int iter = 0; private int sleep = 0;
    public HiloHerencia(String name) {
        this(name, 5, 500);
    }
    public HiloHerencia(String name, int iter, int sleep) {
        super(name);
        this.iter = iter; this.sleep = sleep;
    }
    @Override
    public void run() {
        try {
            for (int i = 0; i < iter; i++) {
                System.out.printf("Hilo: %s Paso: %d de %d\n", getName(), i+1, iter);
                Thread.sleep(sleep);
            }
        } catch (Exception e) { System.out.printf("Error en el hijo %s\n", getName()); }
        System.out.printf("Termina Hilo: %s\n", getName());
    }
}
```

© JMA 2021. All rights reserved

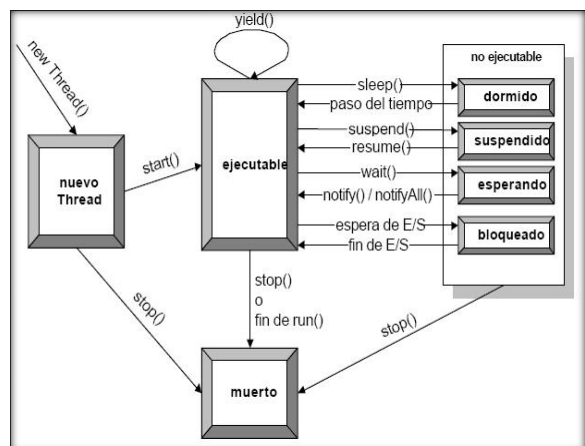
Ejecución simultanea

```
public static void main(String[] args) {  
    HiloRunnable h = new HiloRunnable(1);  
    HiloHerencia h2 = new HiloHerencia("Hijo 99", 50, 99);  
    try {  
        h.lanzar();  
        h2.start();  
        for (int i = 5; i > 0; i--) {  
            System.out.println("Hilo Padre: " + i);  
            Thread.sleep(1000);  
        }  
        System.out.println("Termina el padre");  
    } catch (Exception e) {  
        System.out.println("Error en el padre");  
    }  
}
```

© JMA 2021. All rights reserved

Ciclo de Vida

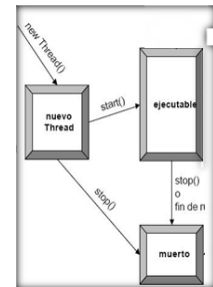
- Se entiende por “ciclo de vida” los diferentes estados en los que un hilo puede pasar desde que se inicia hasta que finaliza.



© JMA 2021. All rights reserved

Ciclo de Vida. Estados

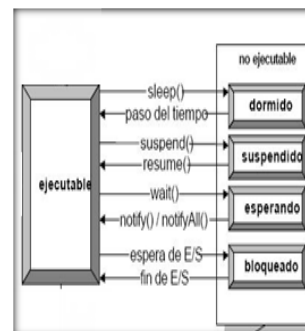
- **Estado Nuevo:**
 - Es el estado en el que se encuentra un thread en cuanto se crea.
 - En ese estado el thread no se está ejecutando, sólo se ha ejecutado el código del constructor del Thread.
- **Ejecutable**
 - Ocurre cuando se llama al método start.
 - No tiene por qué estar ejecutándose el thread, el thread está listo para ser ejecutado; se halla en la cola del planificador del sistema operativo.
- **Muerto**
 - Significa que el método finalizó. Esto puede ocurrir si:
 - El flujo de programa salió del método run
 - Por una excepción no capturada
 - Se invocó el método stop().
 - Se puede comprobar si un thread no está muerto con el método isAlive



© JMA 2021. All rights reserved

Ciclo de Vida. Estados

- **No Ejecutándose:**
 - El thread espera un evento que le permita volver a la cola del planificador.
 - Esto puede ocurrir si:
 - Se llamó al método sleep
 - Se llamó al método wait
 - Se llamó al método suspend
 - Se llamó a una operación de entrada o salida.
 - Se intento bloquear otro thread que ya estaba bloqueado
- **Dormido**
 - Se llamó al método sleep
- **Bloqueado**
 - Se llamó a una operación de entrada o salida.
- **Esperando**
 - Se llamó al método wait
- **Suspendido**
 - Se intento bloquear otro thread que ya estaba bloqueado



© JMA 2021. All rights reserved

Ciclo de Vida. Estados

- **Reanudación:**

- El thread vuelve al estado de Listo a Ejecutarse cuando ocurre alguna circunstancia:

- **Dormido**

- Se pasaron los milisegundos programados por el método sleep

- **Bloqueado**

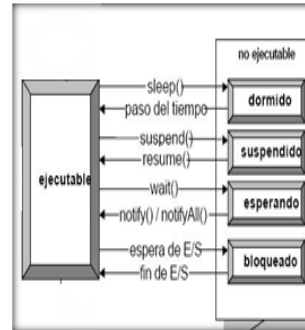
- Se terminó la operación de entrada o salida que había bloqueado al thread
- Se liberó al thread de un bloqueo.

- **Esperando**

- Se llamó a notify tras haber usado wait

- **Suspendido**

- Se ha invocado al método resume.



© JMA 2021. All rights reserved

Estructura

- **Creación:**

- `Hilo miHilo = new Hilo();`

- **Inicio:**

- `miHilo.start();` Implícitamente se invoca el método `run()`.

- **Muerte:**

- El hilo finaliza cuando finaliza el método `run()`

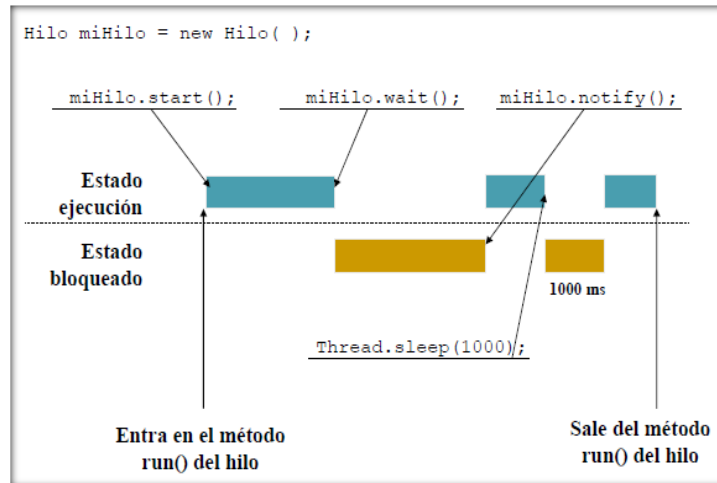
- **Bloqueo de un thread**

- Desbloqueo del thread cuando:**

- `Thread.sleep(10000);` ← Período de tiempo terminado
- `miHilo.wait();` ← `miHilo.notify()` o `miHilo.notifyAll()`
- `Espera de recurso(s)` ← `Recurso(s) liberado(s)`

© JMA 2021. All rights reserved

Estructura



© JMA 2021. All rights reserved

Control de los Threads

- **sleep**
 - Utilizado para que un thread se detenga durante un tiempo sin consumir CPU (`objeto.sleep(mseg)` o `Thread.sleep(mseg)`)
 - Tras consumir finalizar el tiempo indicado, continuará la ejecución.
 - En programación multitarea, dormir un Thread es obligatorio para que los otros se puedan ejecutar.
- **interrupt**
 - Los Threads no tienen ningún método de interrumpir un Hilo.
 - Usando `interrupt` indicamos al Thread que debe de finalizar su proceso cuando pueda.
 - Si el hilo no se está ejecutando, no se atiende la petición hasta que se ejecute

© JMA 2021. All rights reserved

Control de los Threads

```
public class SleepMessages {  
  
    public static void main(String args[]) throws InterruptedException {  
        String importantInfo[] = { "Mares eat oats", "Does eat oats", "Little lambs eat ivy",  
            "A kid will eat ivy too"    };  
  
        for (int i = 0; i < importantInfo.length; i++) {  
            //Pause for 4 seconds  
            Thread.sleep(4000);  
            //Print a message  
            System.out.println(importantInfo[i]);  
        }  
    }  
}
```

© JMA 2021. All rights reserved

Control de los Threads

- wait (tiempo máximo)
 - Establecen un mecanismo seguro para detener y recuperar la ejecución del thread.
 - Este método bloquea al thread actual en un entorno sincronizado para que el resto pueda ejecutarse.
 - Esto se puede hacer definiendo el método con este modificador o definiendo un bloque.

```
public synchronized void metodo()  
{  
    ...  
}
```

```
public void metodo()  
{  
    synchronized (this)  
    {  
        ...  
    }  
}
```

© JMA 2021. All rights reserved

Control de los Threads

- wait (tiempo maximo)

- Este método puede provocar una excepción del tipo InterruptedException si el thread es interrumpido durante la espera.

- Por eso los métodos synchronized deben propagar esta excepción a el thread que lo llamó mediante la palabra throws

```
public class Banco...{  
    ...  
    public synchronized void sacarDinero(int Cantidad) throws InterruptedException{  
        while (saldo<0){  
            wait(); //El thread se queda a la espera de que otro thread cambien  
            // el saldo  
        }  
    }  
}
```

- Se trata mas adelante en Sincronismo.

© JMA 2021. All rights reserved

Control de los Threads

```
public void detener () {  
    hiloDetenido = true;  
}  
  
public synchronized void reanudar() {  
    hiloDetenido = false;  
    this.notify();  
}  
  
public void run() {  
    while(!fin) {  
        txtCrono.setText(String.valueOf(++contador*1000));  
        try {  
            Thread.sleep(1000);  
            synchronized(this) {  
                while(hiloDetenido);  
                this.wait();  
            }  
        }  
        catch(InterruptedException e){}  
    }  
}
```

© JMA 2021. All rights reserved

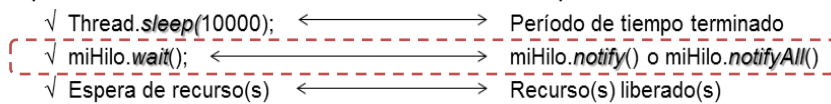
Control de los Threads

- notify y notifyAll

- Los métodos bloqueados con wait se quedan en espera y no salen hasta que sean liberados por otros threads o finalice el tiempo de espera.
- Se puede liberar un thread mediante el método notify para liberar al thread bloqueado o el método notifyAll para liberar a todos los threads bloqueados.

Bloqueo de un thread

Desbloqueo del thread cuando:



- Se trata mas adelante en Sincronismo.

© JMA 2021. All rights reserved

Control de los Threads

- join

- Espera a que finalice el hilo sobre el que se llama.
- Se utiliza habitualmente en el hilo padre para esperar a sus hijos.

- Hilo.isAlive

- Devuelve false si el hilo al que se refiere aún no se ha arrancado o está muerto.

- suspend

- Suspende indefinidamente un hilo.
- No es recomendado su uso → WAIT

- resume

- Activación de nuevo de un hilo suspendido
- No es recomendado su uso → NOTIFY

© JMA 2021. All rights reserved

Prioridades

- Cada hilo se crea con una prioridad, idéntica al proceso/hilo principal que lo lanza. (habitualmente 5)
- Los hilos con mayor prioridad se ejecutan más rápidamente que los de menos prioridad.
- Las prioridades se establecen de 1 a 10 a través del método `setPriority(int)`.
- También existen constantes definidas en la clase `Thread` que definen la prioridad máxima, mínima y normal:
 - `Thread.MIN_PRIORITY` (1)
 - `Thread.NORMAL_PRIORITY` (5)
 - `Thread.MAX_PRIORITY` (10)
- El método de clase `yield` hace que el intérprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible, lo que es una manera de asegurar que los hilos de menor prioridad no sufran inanición.

© JMA 2021. All rights reserved

La interfaz Callable

- La interfaz `Runnable` es una interfaz funcional y tiene un único método `run()` que no acepta ningún parámetro y no devuelve ningún valor. Esto es adecuado para situaciones en las que no buscamos un resultado de la ejecución del hilo. Dado que la firma del método no tiene la cláusula "throws" especificada, no hay forma de propagar más excepciones comprobadas.
- La interfaz `Callable` es una interfaz funcional genérica que contiene un único método `call()`, que devuelve un valor genérico.

```
public class MiCallable implements Callable<Integer> {  
    // ...  
    public Integer call() throws InvalidParamaterException {  
        if(number < 0)  
            throw new InvalidParamaterException("Number should be positive");  
        // ...  
        return number;  
    }  
}
```

© JMA 2021. All rights reserved

Ejecutar un Callable

- Las tareas Runnable se pueden ejecutar usando las clases Thread o ExecutorService, mientras que las Callable solo se pueden ejecutar usando la última.

```
try {
    ExecutorService servicio= Executors.newFixedThreadPool(1);
    Future<Integer> resultado= servicio.submit(new MiCallable());
    if(resultado.isDone())
        System.out.println(resultado.get());
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.getCause().printStackTrace();
} finally {
    servicio.shutdown();
}
```

© JMA 2021. All rights reserved

Future<V>

- Usado ExecutorService se crear un pool de Thread con el número de hilos indicados.
- Para enviar la tarea al pool utilizando el método submit.
- Al invocar el método submit recibiremos de forma automática una instancia de tipo Future.
- Los métodos isDone() e isCancelled() de Future informan del estado de la tarea y con cancel() se cancela.
- El método get() de Future permite recuperar el valor resultado del call() de la tarea, esperando si es necesario.
- En caso de producirse una excepción en la ejecución del método call(), la invocación a get() lanza una ExecutionException, que envuelve la excepción original. Para acceder a la excepción original, ExecutionException dispone del método getCause().

© JMA 2021. All rights reserved

ExecutorService

- El ExecutorService nos simplifica las tareas asíncronas proveyendo un pool de hilos que son manejados el, abstrayéndonos del trabajo de crearlos y asignarles tareas a estos hilos.
- Se puede instanciar:

```
ExecutorService servicio = new ThreadPoolExecutor(10, 10,  
    0L, TimeUnit.MILLISECONDS,  
    new LinkedBlockingQueue<Runnable>());  
ExecutorService servicio = Executors.newFixedThreadPool(10);  
ExecutorService servicio = Executors.newSingleThreadExecutor();
```
- Por defecto ExecutorService se queda 'escuchando' nuevas tareas, para que deje de hacerlo cuando termine la última tarea o inmediatamente indicando las tareas pendientes:

```
servicio.shutdown();  
List<Runnable> runnableList = servicio.shutdownNow();
```

© JMA 2021. All rights reserved

ExecutorService

- EjecutorService.execute es un método void que ejecuta cualquier implementación de Runnable, por lo que no puedes conocer el resultado :

```
servicio.execute(() -> doExecWork("hi 1"));
```
- Para ejecutar un Callable que devuelva un resultado se utiliza EjecutorService.submit:

```
Future<String> future = servicio.submit(() -> doFuncWork("hi with future 1"));
```
- ExecutorService.invokeAny recibe una lista de Callable, ejecuta todas las tareas devolviendo el resultado de una que se ha completado con éxito (es decir, sin lanzar una excepción), si alguna lo hace.

```
Collection<Callable<String>> callables = new ArrayList<>();  
callables.add(() -> doLongWork("hi! 1"));  
callables.add(() -> doLongWork("hi! 2"));  
String oneResult = servicio.invokeAny(callables);
```
- ExecutorService.invokeAll recibe una lista de Callable, ejecuta todas las tareas devolviendo una lista de futuros cuando todos se completa.

```
List<Future<String>> result = servicios.invokeAll(callables);
```

© JMA 2021. All rights reserved

REACTIVE STREAMS

© JMA 2021. All rights reserved

Flujos reactivos

- Reactive Streams es una iniciativa para proporcionar un estándar para el procesamiento de flujo asíncrono con contrapresión sin bloqueo. Esto abarca esfuerzos dirigidos a entornos de ejecución (JVM y JavaScript), así como a protocolos de red.
- Dicha iniciativa publicó en Julio de 2015 la especificación de Reactive Streams 1.0.0 for the JVM con equipos de entre otros de Netflix, Red Hat, Pivotal, Oracle y Typesafe.
- Java 9 (Septiembre de 2017) introdujo la clase `java.util.concurrent.Flow` en un intento de estandarizar de forma nativa la implementación de Reactive Streams. Las interfaces disponibles en `java.util.concurrent.Flow` eran semánticamente equivalentes a sus respectivas contrapartes de Reactive Streams, pero no coincidentes, que supuso la migración de las bibliotecas existentes para adoptar los nuevos tipos en el JDK.

© JMA 2021. All rights reserved

El problema

- El manejo de flujos de datos, especialmente datos "en vivo" cuyo volumen no está predeterminado, requiere un cuidado especial en un sistema asincrónico. El problema más importante es que el consumo de recursos debe controlarse de manera que una fuente de datos rápida no sobrepase el destino de la transmisión. La asincronía es necesaria para permitir el uso paralelo de los recursos informáticos y su optimización.
- El objetivo principal de Reactive Streams es gobernar el intercambio de flujos de datos a través de un límite asincrónico (pasar elementos a otro subproceso o grupo) al tiempo que garantiza que el lado receptor no se vea obligado a almacenar en búfer cantidades arbitrarias de datos. La contrapresión es una parte integral de este modelo para permitir que las colas que median entre hilos estén delimitadas. Los beneficios del procesamiento asincrónico se anularían si la comunicación de la contrapresión fuera síncrona, se debe exigir un comportamiento totalmente asíncrono y sin bloqueo en todos los aspectos.

© JMA 2021. All rights reserved

Reactive Streams

- Los Reactive Streams asemejan al patrón Observer, con un modelo de publicadores y suscriptores. Los publicadores generan una secuencia de elementos que envían a los suscriptores, pero teniendo además estos últimos la capacidad de limitar el número de elementos que quieren recibir.
- Reactive Streams es un estándar y una especificación de bibliotecas para:
 - procesar un número potencialmente ilimitado de elementos
 - en secuencia,
 - pasando los elementos de forma asincrónica entre componentes,
 - con contrapresión no bloqueante obligatoria.

© JMA 2021. All rights reserved

Componentes API

- La API consta de los siguientes componentes que deben proporcionar las implementaciones de Reactive Stream:
 - Publicador
 - Suscriptor
 - Procesador
 - Suscripción
- Un Publicador es un proveedor de un número potencialmente ilimitado de elementos sucesivos, publicándolos de acuerdo con la demanda recibida de sus suscriptores.
- El Publicador admite la suscripción de Suscriptores que reciben un objeto Suscripción con el que demandar elementos o cancelar la demanda.
- El Suscriptor habilita los mecanismos para que el Publicador le indique el siguiente elemento cuando se genere, el error si hay un fallo o que se ha completado cuando no hay más elementos disponibles.
- La secuencia es:
 - `onSubscribe onNext* (onError | onComplete)?`

© JMA 2021. All rights reserved

Reactive Streams

- Java ha añadido interfaces con el mínimo número de métodos necesarios para implementar Reactive Streams, pero no implementaciones de dichas interfaces. Con la creación de estas interfaces dentro del propio JDK se unifican por parte de Java las distintas propuestas existentes actualmente para la implementación de Reactive Streams.
- Las 4 interfaces son:
 - `Flow.Publisher<T>`: Productor de artículos (y eventos de control) recibidos por los suscriptores.
 - `Flow.Subscriber<T>`: Receptor de artículos y mensajes.
 - `Flow.Processor<S, P>`: Componente que actúa como suscriptor y publicador.
 - `Flow.Subscription`: Objeto comunicación del del suscriptor con el publicador.
- Establecen mecanismos de comunicación en una única dirección, mediante procedimientos con un parámetro como máximo, sin acoplamiento.

© JMA 2021. All rights reserved

Flow.Publisher<T>

- La interface funcional Flow.Publisher<T> representa los publicadores.

@FunctionalInterface

```
public static interface Publisher<T> {  
    public void subscribe(Subscriber<? super T> subscriber);  
}
```

- El método subscribe permite que los suscriptores se suscriban a un publicador. El método recibe una instancia de Subscriber a través de la que el publicador se comunicará con el suscriptor. El publicador debe elevar una excepción de tipo NullPointerException si la instancia de Subscriber pasada como parámetro es nula.

© JMA 2021. All rights reserved

Flow.Subscriber<T>

- La interface Flow.Subscriber<T> representa los suscriptores. Es utilizada por los publicadores para notificar eventos a los suscriptores.

```
public static interface Subscriber<T> {  
    public void onSubscribe(Subscription subscription);  
    public void onNext(T item);  
    public void onError(Throwable throwable);  
    public void onComplete();  
}
```

- onSubscribe: una vez aceptada la suscripción (suscribe), se notifica al suscriptor la instancia Subscription a través de la podrá que controlar la suscripción.
- onNext: pasa al suscriptor los elementos generados por la suscripción.
- onComplete: se invoca cuando la suscripción termina, ya sea porque se han consumido el número de elementos solicitado por el suscriptor o el publicador no va a producir nuevos valores, después de lo cual no se invocará ningún método del Suscriptor.
- onError: se invoca tras un error irrecuperable encontrado por un publicador o una suscripción, después del cual el publicador no invoca ningún otro método de Suscriptor. La excepción se pasa como argumento por lo que no puede ser interceptada.
- De forma general el comportamiento no esta definido si se produce un error y se eleva una excepción durante la invocación de cualquiera de estos métodos por parte del suscriptor.

© JMA 2021. All rights reserved

Flow.Subscription

- La interface `Flow.Subscription` representa las suscripciones. Los generan los publicadores para que los suscriptores puedan controlar la suscripción.

```
public static interface Subscription {  
    public void request(long n);  
    public void cancel();  
}
```
- `request`: indica al publicador que está en disposición de recibir los siguientes `n` elementos de la suscripción, además de los ya solicitados (se añaden a los pendientes). Si es menor o igual a cero, se llamará al método `onError` con una `IllegalArgumentException`. Si se solicita `Long.MAX_VALUE` se interpretará que el suscriptor quiere recibir todos los elementos generados por la suscripción sin ningún tipo de restricción en cuanto a su volumen, cuando `n` tiende a infinito.
- `cancel`: hace que el suscriptor (eventualmente) deje de recibir mensajes. Debido a la naturaleza asíncrona de todo el proceso, se pueden recibir mensajes `onNext` adicionales después de invocar la cancelación, pero nunca `onComplete` o `onError`.

© JMA 2021. All rights reserved

Flow.Processor<S, P>

- La interface `Flow.Processor<S, P>` representa un componente que se comporta tanto como un publicador como un suscriptor.

```
public static interface Processor<S, P> extends Subscriber<S>, Publisher<P> {  
}
```
- El propósito principal de esta interface es permitir la implementación por componentes intermedios que se dediquen a realizar algún tipo de transformación sobre los datos recibidos de un publicador para publicarlos para otros componentes una vez transformados.
- Siendo `S` el tipo de origen, al que se suscribe (suscriptor), y `P` el tipo de destino, el que publica (publicador).



© JMA 2021. All rights reserved

Reglas para los Publisher

1. Un publicador DEBE enviar al suscriptor un número total de elementos igual o menor que el número total de elementos solicitados por dicho suscriptor.
2. Un publicador PODRÍA enviar menos elementos que los solicitados por un suscriptor, por ejemplo cuando ya no sea capaz de producir más elementos a partir de su fuente origen de información.
3. Un publicador DEBE invocar los métodos `onSubscribe`, `onNext`, `onError` y `onComplete` de un suscriptor de forma thread-safe y utilizando algún mecanismo de sincronización en caso de utilizar múltiples threads.
4. Un publicador DEBE comunicar sus errores invocando el método `onError` de sus suscriptores.
5. Un publicador DEBE comunicar que ya no puede generar más elementos invocando el método `onComplete` de sus suscriptores.
6. Un publicador que invoque los métodos `onError` u `onComplete` de un suscriptor DEBE considerar cancelada dicha suscripción.

© JMA 2021. All rights reserved

Reglas para los Publisher

7. Un publicador NO DEBERÍA invocar ningún otro método de un suscriptor después de haber invocado sus métodos `onError` u `onComplete`.
8. Un publicador DEBE dejar de invocar eventualmente métodos de un suscriptor si su suscripción ha sido cancelada. No obstante, debido a la naturaleza asíncrona del proceso, y la demora entre la petición de cancelación y su cancelación efectiva, podría llegar a invocar alguno de sus métodos.
9. Un publicador DEBE, dentro de la implementación de su método `subscribe`, invocar el método `onSubscribe` del suscriptor pasado la suscripción como parámetro antes de invocar cualquier otro de método de dicho suscriptor. El método debe terminar normalmente, excepto si el suscriptor pasado como parámetro es nulo, en cuyo caso debe elevar una excepción de tipo `NullPointerException`. Cualquier otro tipo de error debe realizarse invocando el método `onError` del suscriptor.
10. Un publicador NO DEBE permitir la suscripción de un mismo suscriptor más de una vez.
11. Un publicador PUEDE decidir el número de distintos suscriptores que soporta a un mismo tiempo y si la implementación de las suscripciones la realiza de forma unicast (transmisión de uno a uno) o multicast (transmisión de uno a muchos).

© JMA 2021. All rights reserved

Reglas para los Subscriber

1. Un suscriptor DEBE llamar al método request de Subscription para empezar a recibir elementos generados por la suscripción. Es decir, es responsabilidad del suscriptor indicar al publicador cuando quiere empezar a recibir y cuantos elementos está preparado para recibir.
2. Un suscriptor DEBERÍA procesar un elemento de forma asíncrona si estima que su tiempo de respuesta puede afectar la calidad de servicio del publicador.
3. Un suscriptor NO DEBE llamar a ningún método de Subscription ni Publisher dentro de sus métodos onComplete y onError para evitar ciclos y condiciones de carrera.
4. Un suscriptor DEBE considerar cancelada una suscripción cuando el publicador invoca su método onComplete u onError.
5. Un suscriptor DEBE llamar al método cancel de Subscription cuando un publicador invoca su método onSubscribe teniendo el suscriptor ya una suscripción activa.
6. Un suscriptor DEBE llamar al método cancel de Subscription cuando la suscripción ya no sea necesaria para permitir liberar los recursos asociados a dicha suscripción.
7. Un suscriptor DEBE asegurar que todas las llamadas a los métodos de Subscription se realizan desde un mismo thread, o utilizando algún método de sincronización en caso de que la suscripción se esté usando concurrentemente desde distintos threads.

© JMA 2021. All rights reserved

Reglas para los Subscriber

8. Un suscriptor DEBE estar preparado para recibir elementos a través de su método onNext, aún después de haber llamado al método cancel de Subscription, debido a la naturaleza asíncrona del proceso y la demora que se produce entre que el suscriptor solicita la cancelación y el publicador la cancela realmente.
9. Un suscriptor DEBE estar preparado para recibir la finalización de la suscripción a través de su método onComplete, incluso antes de haber llamado al método request de Subscription, debido a que el publicador tiene potestad para dar por terminada una suscripción.
10. Un suscriptor DEBE estar preparado para recibir la finalización de la suscripción a través de su método onError, incluso antes de haber llamado al método request de Subscription, debido a errores en el publicador.
11. Un suscriptor DEBE asegurar que el procesamiento asíncrono de sus métodos onSubscribe, onNext, onComplete y onError se realiza de manera thread-safe.
12. Un mismo suscriptor DEBE suscribirse a un mismo publicador una única vez.
13. Un suscriptor DEBE terminar las invocaciones a sus métodos onSubscribe, onNext, onComplete y onError sin elevar ninguna excepción. Excepto cuando algún parámetro recibido sea nulo, en cuyo caso deberá elevar una excepción de tipo NullPointerException. Cualquier otro error debe resolverse cancelando la suscripción. El publicador no puede tratar los errores del suscriptor, por lo que dará por cancelada la suscripción en caso de producirse alguno.

© JMA 2021. All rights reserved

Reglas para los Subscription

1. Una suscripción es el único componente a través del que se establece la comunicación entre un suscriptor y un publicador. Un objeto Subscription concreto pertenece de forma exclusiva a un Publisher y un Subscriber concretos. Sólo el suscriptor DEBE llamar a los métodos request y cancel de Subscription.
2. Una suscripción DEBE permitir que se llame a su método request desde los métodos onSubscribe y onNext de un suscriptor de manera síncrona. Es decir, la implementación del método request debe ser reentrante.
3. Una suscripción DEBE establecer un número máximo de llamadas recursivas entre los métodos request de Subscriber y onNext de Subscription. La recomendación a este respecto es limitar a uno el número de llamadas recursivas evitando la secuencia request -> onNext -> request -> onNext -> ...
4. Una suscripción DEBERÍA implementar su método request de forma que retorne lo antes posible para no afectar la calidad de servicio del suscriptor que lo invoque.
5. Una suscripción DEBE implementar su método cancel de forma idempotente, thread-safe y retornar lo antes posible para no afectar la calidad del servicio del suscriptor que lo invoque.
6. Una suscripción NO DEBE realizar ninguna acción cuando se llame a su método request una vez que la suscripción haya sido cancelada.
7. Una suscripción NO DEBE realizar ninguna acción cuando se llame a su método cancel una vez que la suscripción haya sido cancelada.
8. Una suscripción DEBE almacenar la suma de todas las cantidades de elementos solicitadas por todas las llamadas realizadas por el suscriptor a su método request.

© JMA 2021. All rights reserved

Reglas para los Subscription

9. Una suscripción DEBE invocar al método onError del suscriptor con una instancia de una excepción IllegalArgumentException si se solicita una cantidad de elementos igual o menor que cero en alguna llamada realizada por el suscriptor a su método request.
10. Una suscripción PODRÍA implementar en su método request invocaciones al método onNext del suscriptor de manera síncrona. Siempre y cuando la suscripción no haya sido ya cancelada.
11. Una suscripción PODRÍA implementar en su método request invocaciones al método onComplete del suscriptor de manera síncrona. Siempre y cuando la suscripción no haya sido ya cancelada.
12. Una suscripción DEBE requerir al Publisher que deje de invocar los métodos del Subscriber cuando el suscriptor llame a su método cancel. Siempre y cuando la suscripción no haya sido ya cancelada.
13. Una suscripción DEBE requerir del Publisher que libere cualquier referencia al Subscriber cuando el suscriptor llame a su método cancel. Siempre y cuando la suscripción no haya sido cancelada.
14. Una suscripción PODRÍA causar que el Publisher pase a un estado de finalizado cuando el suscriptor llame a su método cancel. Siempre y cuando la suscripción no haya sido ya cancelada.
15. Una suscripción NO DEBE elevar excepciones en la implementación de su método cancel .
16. Una suscripción NO DEBE elevar excepciones en la implementación de su método request.
17. Una suscripción DEBE permitir peticiones de una cantidad de elementos potencialmente infinita. Una petición de $2^{63}-1$ (java.lang.Long.MAX_VALUE) o más elementos se considera como una petición de una cantidad infinita de elementos.

© JMA 2021. All rights reserved

Reglas para los Processor

1. Un procesador DEBE cumplir tanto los requerimientos establecidos para los publicadores como para los suscriptores.
2. Un procesador PODRÍA elegir recuperarse de una invocación a su método `onError` por parte de un publicador, pero en ese caso DEBE considerar la suscripción cancelada, en caso contrario DEBE propagar el error inmediatamente invocando el método `onError` de sus suscriptores.

© JMA 2021. All rights reserved

SubmissionPublisher<T>

- `SubmissionPublisher` es una implementación de `Flow.Publisher` que emite elementos (no nulos) enviados de forma asíncrona a los suscriptores actuales hasta que se cierra. Cada suscriptor actual recibe los elementos recién enviados en el mismo orden a menos que se encuentren caídos o excepciones. Su permite que los generadores de elementos actúen como publicadores de flujos reactivos que se basan en el manejo caídas y/o bloqueo del control del flujo.
- Un `SubmissionPublisher` utiliza el `Executor` suministrado en su constructor para la entrega a los suscriptores. La mejor elección de `Ejecutor` depende del uso esperado.
- El almacenamiento en búfer permite a los productores y consumidores operar a diferentes velocidades. A cada suscriptor se le asigna un búfer independiente. Los búferes se crean con el primer uso y se expanden según sea necesario hasta el máximo dado.
- Los métodos de publicación admiten diferentes políticas sobre qué hacer cuando los búferes están saturados:
 - `submit`: Publica el elemento dado para cada suscriptor actual invocando de forma asíncrona su método `onNext`, bloqueando ininterrumpidamente los recursos para cualquier suscriptor mientras no están disponibles.
 - `offer`: A diferencia del `submit`, solo mantiene el bloqueo hasta el tiempo de espera especificado o hasta que se interrumpe el hilo, en cuyo caso la invocación del controlador de caída suministrado debe indicar si se reintentará.

© JMA 2021. All rights reserved

SubmissionPublisher<T>

- Si algún método del suscriptor genera una excepción, su suscripción se cancela. Si se proporciona un controlador de errores como un argumento del constructor, se invoca antes de la cancelación en caso de una excepción en el método onNext, pero excepciones en los métodos onSubscribe, onError y onComplete no se registran ni se maneja antes de la cancelación. Si el Ejecutor suministrado arroja RejectedExecutionException (o cualquier otra RuntimeException o Error) al intentar ejecutar una tarea, o un controlador de caída arroja una excepción al procesar un elemento eliminado, la excepción se vuelve a lanzar. En estos casos, no todos los suscriptores habrán recibido el artículo publicado. Por lo general, es una buena práctica en estos casos invocar a closeExceptionally que, a menos que ya esté cerrado, ejecuta los onError de los suscriptores actuales con el error dado y no permite los intentos posteriores de publicación.
- El método consume(Consumer) simplifica el soporte para un caso común en el que la única acción de un suscriptor es solicitar y procesar todos los elementos utilizando una función proporcionada.
- Esta clase también puede servir como una base conveniente para las subclasses que generan elementos y utilizar los métodos de esta clase para publicarlos.

© JMA 2021. All rights reserved

PrintSubscriber

```
private static class PrintSubscriber implements Flow.Subscriber<Integer> {
    private Flow.Subscription subscription;

    @Override
    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }
    @Override
    public void onNext(Integer item) {
        System.out.println("Received item: " + item);
        subscription.request(1);
        Sleeper.sleep(1000);
    }
    @Override
    public void onError(Throwable error) {
        error.printStackTrace();
    }
    @Override
    public void onComplete() {
        System.out.println("PrintSubscriber completed");
    }
}
```

© JMA 2021. All rights reserved

CacheSubscriber

```
private static class CacheSubscriber implements Flow.Subscriber<Integer> {
    private Flow.Subscription subscription;
    private List<Integer> cache = new ArrayList<Integer>();
    private int MAX_CACHE, count = 0;

    public CacheSubscriber(int max) { MAX_CACHE = max; }

    @Override
    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        subscription.request(MAX_CACHE);
    }
    @Override
    public void onNext(Integer item) {
        cache.add(item);
        if(++count >= MAX_CACHE) onComplete();
    }
    @Override
    public void onError(Throwable error) {
        error.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("CacheSubscriber completed: " + cache.size());
        for (Integer item : cache) {
            System.out.println("Cache item: " + item);
        }
    }
}
```

© JMA 2021. All rights reserved

GenericProcessor

```
public static class GenericProcessor<S, P> extends SubmissionPublisher<P> implements Flow.Processor<S, P> {
    private Flow.Subscription subscription;
    private Function<S, P> function;

    public GenericProcessor(Function<S, P> function) {
        super();
        this.function = function;
    }
    @Override
    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        subscription.request(Long.MAX_VALUE);
    }
    @Override
    public void onNext(S item) { submit(function.apply(item)); }
    @Override
    public void onError(Throwable error) { error.printStackTrace(); }
    @Override
    public void onComplete() {
        System.out.println("GenericProcessor completed");
        close();
    }
}
```

© JMA 2021. All rights reserved

SubmissionPublisher

```
private static class Sleeper {
    private static void sleep(int time) {
        try {
            Thread.sleep(time);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public static void ejecutar() throws Exception {
    SubmissionPublisher<Integer> publisher = new SubmissionPublisher<>();
    Flow.Processor<Integer, Integer> processor = new GenericProcessor<>{(item -> item * 2);
    Flow.Subscriber<Integer> display = new PrintSubscriber();
    Flow.Subscriber<Integer> cache = new CacheSubscriber(10);

    publisher.subscribe(cache);
    publisher.subscribe(processor);
    processor.subscribe(display);

    IntStream.range(0, 10).forEach(item -> {
        System.out.println("Generated item: " + item);
        publisher.submit(item);
        Sleeper.sleep(500);
    });

    publisher.close();
    System.out.println("Publisher completed and closed");
    Sleeper.sleep(6000);
}
```

© JMA 2021. All rights reserved

Executor

```
static class PeriodicPublisher<T> extends SubmissionPublisher<T> {
    final ScheduledFuture<?> periodicTask;
    final ScheduledExecutorService scheduler;
    PeriodicPublisher(Executor executor, int maxBufferCapacity, Supplier<? extends T> supplier, long period,
        TimeUnit unit) {
        super(executor, maxBufferCapacity);
        scheduler = new ScheduledThreadPoolExecutor(1);
        periodicTask = scheduler.scheduleAtFixedRate(() -> submit(supplier.get()), 0, period, unit);
    }
    public void close() {
        periodicTask.cancel(false);
        scheduler.shutdown();
        super.close();
    }
}

private static int value = 0;
public static void ejecutar() throws Exception {
    ExecutorService servicio = Executors.newFixedThreadPool(10);
    SubmissionPublisher<Integer> publisher = new PeriodicPublisher<>(servicio, 10, () -> value++, 1, TimeUnit.SECONDS);
    publisher.subscribe(new PrintSubscriber());
    servicio.awaitTermination(10, TimeUnit.SECONDS);
    publisher.close(); System.out.println("Publisher completed and closed");
}
```

© JMA 2021. All rights reserved

Implementaciones

- Akka Streams
 - Reactor
 - RxJava
 - MongoDB
 - Reactive Rabbit
 - Ratpack
 - Slick
 - Vert.x 3.0
-

© JMA 2021. All rights reserved

REACTOR PROJECT

© JMA 2021. All rights reserved

Reactor

- Project Reactor es una base totalmente no bloqueante para la JVM con soporte de contrapresión incluido, que implementa ampliamente la especificación Reactive Streams. Se integra directamente con las API de Java funcionales 8, en particular CompletableFuture, Stream y Duration.
- Proporciona los tipos Mono y Flux para trabajar en secuencias de datos 0..1 (Mono) y 0..N (Flux) a través de un rico conjunto de operadores alineados con el vocabulario de operadores de ReactiveX. Reactor es una biblioteca de Flujos reactivos y, por lo tanto, todos sus operadores admiten contrapresión sin bloqueo. Reactor esta fuertemente enfocado al Java del lado del servidor.
- Reactor también admite la comunicación entre procesos sin bloqueo con el proyecto reactor-netty. Adecuado para la arquitectura de microservicios, Reactor Netty ofrece motores de red preparados para contrapresión para HTTP (incluidos Websockets), TCP y UDP. La codificación y decodificación reactiva son totalmente compatibles.

© JMA 2021. All rights reserved

Reactivo no bloqueante con contrapresión

- El término "reactivo" se refiere a modelos de programación que se construyen en torno a la reacción al cambio: componentes de red que reaccionan a eventos de E / S, controladores de UI que reaccionan a eventos de ratón o teclado y otros. En ese sentido, el no bloqueo es reactivo, porque, en lugar de estar bloqueado para esperar, ahora estamos en el modo de reaccionar a las notificaciones cuando las operaciones se completan o los datos están disponibles.
- También hay otro mecanismo importante que asociamos con "reactivo" y es la contrapresión sin bloqueo, contra la presión de demasiadas invocaciones que produzcan una saturación. En el código imperativo sincrónico, el bloqueo de llamadas sirve como una forma natural de contrapresión que obliga a la persona que llama a esperar. En el código sin bloqueo, se vuelve importante controlar la tasa de eventos para que un origen rápido no abrume a su destino.

© JMA 2021. All rights reserved

Instalación de Maven

- Maven es compatible de forma nativa con el concepto de lista de materiales. Primero, se debe importar la lista de materiales agregando el siguiente fragmento al pom.xml:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.projectreactor</groupId>
      <artifactId>reactor-bom</artifactId>
      <version>2020.0.13</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```
- A continuación, se agregan las dependencias:

```
<dependencies>
  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
  </dependency>
  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

© JMA 2021. All rights reserved

De la programación imperativa a la reactiva

- Java ofrece dos modelos de programación asíncrona:
 - Devoluciones de llamada: los métodos asíncronos no tienen un valor de retorno, pero toman un parámetro callback adicional que se llama cuando el resultado está disponible.
 - Futuros: los métodos asíncronos devuelven inmediatamente a Future<T>. El proceso asíncrono calcula el valor T, pero el objeto Future envuelve el acceso a él. El valor no está disponible de inmediato y el objeto se puede sondear hasta que el valor esté disponible.
- Las devoluciones de llamada son difíciles de componer juntas, lo que lleva rápidamente a un código que es difícil de leer y mantener (el "Infierno de devolución de llamada"). Los objetos Future son un poco mejores que las devoluciones de llamada, pero todavía no les va bien en la composición.
- Las bibliotecas reactivas, como Reactor, tienen como objetivo abordar estos inconvenientes de los enfoques asíncronos "clásicos" en la JVM y, al mismo tiempo, se centran en algunos aspectos adicionales:
 - Componibilidad y legibilidad
 - Datos como un flujo manipulado con un rico vocabulario de operadores
 - No pasa nada hasta que te suscribes
 - Contrapresión o la capacidad del consumidor de indicarle al productor que está listo para el siguiente
 - Abstracción de alto nivel pero de alto valor que es independiente de la concurrencia

© JMA 2021. All rights reserved

Interfaz fluida

- Por "componibilidad", se refiere a la capacidad de orquestar múltiples tareas asincrónicas, en las que se utilizan los resultados de las tareas anteriores para alimentar las entradas de las siguientes. La capacidad de orquestar tareas está estrechamente ligada a la legibilidad y mantenibilidad del código. A medida que las capas de procesos asincrónicos aumentan tanto en número como en complejidad, poder componer y leer código se vuelve cada vez más difícil. Reactor ofrece opciones de composición ricas, en las que el código refleja la organización del proceso abstracto y, en general, todo se mantiene al mismo nivel (el anidamiento se minimiza).
- Cada operador agrega comportamiento a un Publisher y envuelve los pasos anteriores en una nueva instancia Publisher. De esta manera, toda la cadena está vinculada, de modo que los datos se originan desde el primero Publisher y se mueven hacia abajo en la cadena, transformados por cada eslabón. Finalmente, un Subscriber finaliza el proceso.
- En Reactor, cuando escribe una cadena Publisher, los datos no comienzan a bombearse a ella de forma predeterminada. En su lugar, crea una descripción abstracta del proceso asíncrono (que puede ayudar con la reutilización y la composición). Mediante el acto de suscribirse, se vincula una cadena Publisher a un Subscriber, lo que desencadena el flujo de datos en toda la cadena. Esto se logra internamente mediante una única señal request del Subscriber que se propaga en sentido ascendente, hasta la fuente Publisher. La propagación en sentido ascendente también se utiliza para la contrapresión, los operadores intermedios también pueden cambiar la solicitud en tránsito.

© JMA 2021. All rights reserved

Caliente vs frío

- La familia Rx de bibliotecas reactivas distingue dos amplias categorías de secuencias reactivas: caliente y fría .
- Esta distinción tiene que ver principalmente con cómo reacciona el flujo reactivo a los suscriptores:
 - Una secuencia fría no comienza hasta que tiene un suscriptor y comienza de nuevo para cada Subscriber, incluso en la fuente de datos. Por ejemplo, si la fuente envuelve una llamada HTTP, no se realiza la solicitud hasta disponer de una suscripción que procese la respuesta y se realiza una nueva solicitud HTTP para cada suscripción.
 - Una secuencia caliente puede comenzar desde el momento de su creación, puede incluso emitirse cuando ningún suscriptor está escuchando (una excepción a la regla de "no pasa nada antes de suscribirse") y no comienza desde cero para cada uno Subscriber, los suscriptores tardíos reciben las señales emitidas a partir del momento de suscribirse. Algunas corrientes reactivas calientes pueden almacenar en caché o reproducir el historial de emisiones total o parcialmente, permitiendo a los suscriptores recibir señales emitidas antes del momento de suscribirse.

© JMA 2021. All rights reserved

Flux

- A `Flux<T>` es un `Publisher<T>` estándar que representa una secuencia asíncrona de 0 a N elementos emitidos, opcionalmente terminado por una señal de finalización o un error. Al igual que en la especificación Reactive Streams, estos tres tipos de señales se traducen en llamadas a los métodos `onNext`, `onComplete` y `onError` de un suscriptor del flujo.
- Con esta amplia gama de posibles señales, Flux es el tipo reactivo de uso general. Hay que tener en cuenta que todos los eventos, incluso los que terminan, son opcionales: ningún evento `onNext`, pero un evento `onComplete` representa una secuencia finita vacía, pero si se elimina el `onComplete` se tendrá una secuencia vacía infinita (no es particularmente útil, excepto para las pruebas de cancelación).
- De manera similar, las secuencias infinitas no están habitualmente vacías. Por ejemplo, `Flux.interval(Duration)` produce un `Flux<Long>` infinito y emite tics regulares desde un reloj.

© JMA 2021. All rights reserved

Mono

- `Mono<T>` es una especialización de `Publisher<T>` que emite como máximo un elemento a través de la señal `onNext` y luego termina con una señal `onComplete` (éxito, con o sin valor), o solo emite una sola señal `onError` (fallido).
- En la mayoría de las implementaciones de Mono se espera que se llame inmediatamente a `onComplete` en su `Subscriber` después de haber llamado `onNext`. `Mono.never()` es un valor atípico: no emite ninguna señal (infinito), lo que técnicamente no está prohibido aunque no es muy útil fuera de las pruebas. Por otro lado, una combinación de `onNext` y `onError` está explícitamente prohibida.
- Mono ofrece solo un subconjunto de los operadores que están disponibles para Flux, y algunos operadores (en particular los que combinan el Mono con otro `Publisher`) lo cambian a un Flux. Por ejemplo, `Mono#concatWith(Publisher)` devuelve un Flux mientras `Mono#then(Mono)` devuelve otro Mono.
- Hay que tener en cuenta que se puede utilizar un Mono para representar procesos asíncronos sin valor, que solo tienen el concepto de finalización (similar a un `Runnable`). Para crear uno, se puede utilizar un `Mono<Void>`.

© JMA 2021. All rights reserved

Creación de tipos reactivos

- La forma más sencilla de empezar con Flux y Mono es utilizar uno de los numerosos métodos de fábrica que se encuentran en sus respectivas clases: just, empty, range, ...

```
Mono<String> noData = Mono.empty(), data = Mono.just("foo");  
Flux<Integer> numbers = Flux.range(5, 3);  
Flux<String> seq = Flux.fromIterable(list);
```
- La forma más simple de creación programática de un Flux (síncrono y uno por uno) es a través del método generate, que toma una función de generación.

```
Flux<String> flux = Flux.generate(  
    () -> 0, // Suministramos el valor de estado inicial de 0.  
    (estado, suscriptor) -> {  
        suscriptor.next("2 x " + estado + " = " + 2* estado); // suministramos el siguiente valor  
        if (estado == 10) suscriptor.complete(); // indicamos que se ha completado la secuencia.  
        return estado + 1;  
    });
```
- Los métodos push (asíncrono y monoproceso) y create (asíncrono y multiproceso) son formas más avanzada de creación programática de un Flux.

© JMA 2021. All rights reserved

Suscriptores

- Cuando se trata de suscribirse a Flux y Mono se recomienda hacer uso de las lambdas de Java 8. El método .subscribe() tiene una amplia variedad de sobrecargas para diferentes combinaciones de devoluciones de llamada:

```
.subscribe();  
.subscribe(Consumer<? super T> consumer);  
.subscribe(Consumer<? super T> consumer, Consumer<? super Throwable> errorConsumer);  
.subscribe(Consumer<? super T> consumer, Consumer<? super Throwable> errorConsumer,  
    Runnable completeConsumer);
```



```
Flux<Integer> pares = Flux.just(2, 4, 6, 8, 10);  
pares.subscribe(i -> System.out.println(i),  
    error -> System.err.println("Error " + error),  
    () -> System.out.println("Done"));
```
- Todas estas variantes basadas en lambda devuelven Disposable. La interfaz Disposable representa el hecho de que la suscripción se puede cancelar llamando a su método dispose().

```
pares.dispose()
```

© JMA 2021. All rights reserved

Suscriptores

- Como alternativa a las lambdas se pueden utilizar instancias de herederos de `BaseSubscriber`, que implementen como métodos las diferentes acciones. Las instancias de `BaseSubscriber` (o subclases de él) son de un solo uso, lo que significa que si un `BaseSubscriber` cancela su suscripción al primer `Publisher` también lo hace a un segundo `Publisher` si está suscrito.

```
public class SampleSubscriber<T> extends BaseSubscriber<T> {  
    public void hookOnSubscribe(Subscription subscription) {  
        System.out.println("Subscribed");  
        request(1);  
    }  
    public void hookOnNext(T value) {  
        System.out.println(value);  
        request(1);  
    }  
}  
SampleSubscriber<Integer> mySubscriber = new SampleSubscriber<Integer>();  
Flux<Integer> ints = Flux.range(1, 4);  
ints.subscribe(mySubscriber);
```

© JMA 2021. All rights reserved

Suscripción

- La clase `FluxSink` envuelve la suscripción descendente para emitir cualquier número de señales `onNext` seguidas de una o ninguna señal `onError` o `onComplete`. Para controlar la suscripción y realizar limpieza en la cancelación o terminación, suministra los métodos:
 - `next(T t)`, `error(Throwable e)`, `complete()`, `isCancelled()`, `onCancel(Disposable d)`, `onDispose(Disposable d)`, `onRequest(LongConsumer consumer)`, `currentContext()`, `requestedFromDownstream()`.
- El método `create` es una forma más avanzada de creación programática de un `Flux` asíncrono y multiproceso, que es adecuada para múltiples emisiones por ronda, incluso de múltiples subprocesos. El método no paraleliza el código ni lo hace asíncrono, lo asume, aunque se puede usar con API asíncrona.
`Flux<String> flux = Flux.create(sink -> {});`
- El método `push` es un término medio entre `generate` y `create` que es adecuado para procesar eventos de un solo productor. Es similar a `create` en el sentido de que también puede ser asíncrono y puede gestionar la contrapresión utilizando cualquiera de las estrategias de desbordamiento admitidas por `create`. Sin embargo, solo se pueden invocar `next`, `complete` o `error` en el hilo productor.

© JMA 2021. All rights reserved

Create

```
public static class MessageService {
    List<FluxSink> sinks = new ArrayList<FluxSink>();
    public void add(FluxSink sink) {
        sinks.add(sink);
        sink.onDispose(() -> sinks.remove(sink));
    }
    public void emit(List<String> chunk) {
        chunk.forEach(item -> sinks.forEach(sink -> sink.next(item)));
    }
    public void processComplete() { List.copyOf(sinks).forEach(FluxSink::complete); }
}

var srv = new MessageService();
Flux<String> flux = Flux.create(sink -> srv.add(sink));
srv.emit(List.of("uno", "dos", "tres"));
```

© JMA 2021. All rights reserved

Manejo de errores

- En Reactive Streams, los errores son eventos terminales. Tan pronto como ocurre un error, detiene la secuencia y se propaga por la cadena de operadores hasta el último paso, el Subscriber definido y su método onError.
s.subscribe(value -> System.out.println("RECEIVED " + value),
error -> System.err.println("CAUGHT " + error)
);
- Estos errores aún deben tratarse en el nivel de la aplicación. Por ejemplo, puede mostrar una notificación de error en una interfaz de usuario o enviar una carga útil de error significativa en un punto final REST. Por esta razón, siempre se debe definir el método onError del suscriptor .
- Si no está definido onError y hubiese un error, se lanzaría UnsupportedOperationException.
- Reactor también ofrece medios alternativos para lidiar con errores en el medio de la cadena, como operadores de manejo de errores:
Flux.just(1, 2, 0).map(i -> "100 / " + i + " = " + (100 / i)) //this triggers an error with 0
.onErrorReturn("Divided by zero :("); // error handling example

© JMA 2021. All rights reserved

Manejo de errores

- "Atrapar y volver a lanzar" permite interceptar y envolver una excepción:

```
Flux.just("timeout1").flatMap(k -> callExternalService(k))  
    .onErrorResume(original -> Flux.error(new BusinessException("oops, SLA exceeded", original)));  
Flux.just("timeout1").flatMap(k -> callExternalService(k))  
    .onErrorMap(original -> new BusinessException("oops, flux error", original));
```
- `onErrorResume` tiene variantes que permiten filtrar a qué excepciones recurrir, en función de la clase de la excepción o de un Predicate. El hecho de que se necesite Function también permite elegir una secuencia alternativa diferente para cambiar, según el error encontrado.

```
Flux.just("timeout1", "unknown", "key2").flatMap(k -> callExternalService(k))  
    .onErrorResume(error -> {  
        if (error instanceof TimeoutException) return getFromCache(k);  
        else if (error instanceof UnknownKeyException)  
            return registerNewEntry(k, "DEFAULT");  
        else return Flux.error(error);  
    })  
);
```

© JMA 2021. All rights reserved

Operadores

- Flux y Mono suministran métodos adicionales, denominados operadores, para permitir la manipulación sofisticada de los flujos.
- Cada operador agrega comportamiento a un Publisher y envuelve los pasos anteriores en una nueva instancia Publisher.
- Los operadores se pueden emplear para:
 - Crear una nueva secuencia
 - Transformar una secuencia existente
 - Filtrar una secuencia
 - Echar un vistazo a una secuencia
 - Manejar errores
 - Trabajar con el tiempo
 - Dividir o reunir un flujo
 - Regresar al modo síncrono
 - Pasar a multidifusión de un flujo a varios suscriptores

© JMA 2021. All rights reserved

Crear una nueva secuencia

- Que emite un T, que ya tengo: `just(Flux|Mono)`, o es opcional : `Mono#justOrEmpty (<T> opcional | null)` o que es devuelto por un método: `just(Flux|Mono)`, `Mono#fromSupplier` o un `just(Flux|Mono)` envuelto dentro `defer(Flux|Mono)`
- Que emite varios T que puedo enumerar explícitamente: `Flux#just (T...)` o iterar sobre: una matriz: `Flux#fromArray`, una colección o iterable: `Flux#fromIterable`, un rango de enteros: `Flux#range`, una secuencia suministrada para cada suscripción: `Flux#fromStream (Supplier<Stream>)`
- Que se emite desde varias fuentes de valor único como: un proveedor: `Mono#fromSupplier`, una tarea: `Mono#fromCallable` , `Mono#fromRunnable`, `Mono#fromFuture`
- Que ya está completa pero vacía: `empty(Flux|Mono)`, errónea: `error(Flux|Mono)`, que nunca hace nada: `never(Flux|Mono)`, que se decide en la suscripción: `defer(Flux|Mono)` o que depende de un recurso disponible: `using(Flux|Mono)`
- Que genera eventos programáticamente (puede usar el estado): síncronamente y uno por uno: `Flux#generate`, o asincrónicamente con múltiples emisiones posibles en una sola pasada: `Flux#create (Mono#create también, pero con emisión única)`

© JMA 2021. All rights reserved

Transformar una secuencia existente

- Para transformar los datos existentes 1 a 1 cambiando el tipo: `cast(Flux|Mono)`, usando un `Function`: `map(Flux|Mono)`, en su índice: `Flux#index`
- Para transformar los datos existentes 1 a n usando un método de fábrica `flatMap(Flux|Mono)` o con `BiConsumer` programático para cada elemento de origen y/o estado: `handle(Flux|Mono)`
- Para agregar elementos preestablecidos a una secuencia existente al inicio: `Flux#startWith (T...)` o al final: `Flux#concatWithValues (T...)`
- Para transformar un Flux en una lista: `collectList` , `collectSortedList`, en un mapa: `collectMap` , `collectMultiMap`, en un contenedor arbitrario: `collect`, en un entero con el tamaño de la secuencia: `count`, en un valor aplicando una función: `reduce`, pero emitiendo cada valor intermedio: `scan`, en un valor booleano: `all`, `any`, `hasElements`.
- Para combinar publishers en orden secuencial: `Flux#concat` o `.concatWith(other)(Flux|Mono)`, `Flux#concatDelayError`, `Flux#mergeSequential`, o en orden de emisión: `Flux#merge`, `.mergeWith(other)(Flux|Mono)`, `Flux#zip`, `Flux#zipWith`, por pares de valores: `Mono#zipWith`, `Mono#zip`, coordinando su terminación: `Mono#and`, `Mono#when`, `Flux#combineLatest`, `firstWithValue(Flux|Mono)`, `firstWithSignal(Flux|Mono)`, `switchMap`, `switchOnNext`
- Para repetir una secuencia existente: `repeat(Flux|Mono)` pero a intervalos de tiempo: `Flux.interval(duration).flatMap(tick → myExistingPublisher)`

© JMA 2021. All rights reserved

Transformar una secuencia existente

- Para transformar una secuencia vacía en un valor: `defaultIfEmpty(Flux|Mono)` o en otra secuencia: `switchIfEmpty(Flux|Mono)`
- Para una secuencia de la que no interesan los valores: `ignoreElements()` | `Mono.ignoreElement()`
- ... y que la finalización se represente como `Mono<Void>`: `then(Flux|Mono)`
- ... y esperar a que se complete otra tarea al final: `thenEmpty(Flux|Mono)`
- ... y cambiar a otro `Mono` al final: `Mono#then(mono)`
- ... y emitir un solo valor al final: `Mono#thenReturn(T)`
- ... y cambiar a un `Flux` al final: `thenMany(Flux|Mono)`
- Para aplazar la finalización de un `Mono` hasta que otro publishers, que se deriva de este valor, se haya completado: `Mono#delayUntil(Función)`
- Para expandir elementos de forma recursiva en un grafo de secuencias y emitir la combinación expandiendo primero la amplitud del grafo (horizontal): `expand(Flux|Mono)` o la profundidad del grafo primero (vertical): `expandDeep(Flux|Mono)`

© JMA 2021. All rights reserved

Filtrar una secuencia

- Para filtrar una secuencia basado en un criterio arbitrario: `filter(Flux|Mono)` que se calcula de forma asincrónica: `filterWhen(Flux|Mono)`, restringiendo el tipo de objetos emitidos: `ofType(Flux|Mono)`, ignorando los valores por completo: `ignoreElements(Flux|Mono)` o ignorando los duplicados: en toda la secuencia: `Flux#distinct` o entre elementos emitidos posteriormente: `Flux#differentUntilChanged`
- Para mantener solo un subconjunto de la secuencia:
 - tomando N elementos al comienzo de la secuencia: `Flux#take` o solo el primer elemento, como `Mono`: `Flux#next()`, usando la solicitud (N) en lugar de la cancelación: `Flux#limitRequest` o `Flux#limitRate`, al final de la secuencia: `Flux#takeLast`, hasta que se cumpla un criterio (incluido): `Flux#takeUntil` (basado en predicado), `Flux#takeUntilOther` (basado en un publicador complementario), mientras se cumple un criterio (exclusivo): `Flux#takeWhile`, tomando como máximo 1 elemento en una posición específica: `Flux#elementAt` o al final: `.takeLast(1)` o `Flux#last()`
 - saltando elementos al comienzo de la secuencia: `Flux#skip`, al final de la secuencia: `Flux#skipLast`, hasta que se cumpla un criterio (incluido): `Flux#skipUntil` (basado en predicado), `Flux#skipUntilOther` (basado en publicador complementario), mientras se cumple un criterio (exclusivo): `Flux#skipWhile`
 - tomando una muestra de elementos: `Flux#sample`, `Flux#sampleFirst`, `Flux#sampleTimeout`
- Para obtener como máximo 1 elemento (error si hay más de uno) y si la secuencia está vacía un error: `Flux#single()` o una secuencia vacía: `Flux#singleOrEmpty`

© JMA 2021. All rights reserved

Echar un vistazo a una secuencia

- Sin modificar la secuencia final, para recibir notificaciones y poder ejecutar comportamientos adicionales (efectos secundarios):
 - `doOnNext(Flux|Mono)`, `Flux#doOnComplete`, `Mono#doOnSuccess`, `doOnError(Flux|Mono)`, `doOnCancel(Flux|Mono)`, `doFirst(Flux|Mono)`, `doOnSubscribe(Flux|Mono)`, `doOnRequest(Flux|Mono)`,
 - finalización o error: `doOnTerminate(Flux|Mono)`, pero después de que se haya propagado corriente abajo: `doAfterTerminate(Flux|Mono)`
 - cualquier tipo de señal, representada como una señal : `doOnEach(Flux|Mono)`
 - cualquier condición de terminación (completa, error, cancelar): `doFinally(Flux|Mono)`
 - registrar lo que sucede internamente: `log(Flux|Mono)`
- Para saber de todos los eventos cada uno representado como objeto de señal :
 - en una devolución de llamada fuera de la secuencia: `doOnEach(Flux|Mono)`
 - en lugar de las emisiones originales `onNext`: `materialize(Flux|Mono)`
 - ... y volver a los `onNexts`: `dematerialize(Flux|Mono)`

© JMA 2021. All rights reserved

Manejar errores

- Para crear una secuencia errónea: `error(Flux|Mono)`
 - ... y reemplazar la finalización de un Flux exitoso : `concat(Flux.error(e))`
 - ... y reemplazar la emisión de un Mono exitoso : `then(Mono.error(e))`
 - ... si transcurre demasiado tiempo entre `onNexts`: `timeout(Flux|Mono)`
- El equivalente `try / catch` capturando una excepción:
 - ... y devolver un valor predeterminado: `onErrorReturn(Flux|Mono)`
 - ... y devolver otro Flux o Mono : `onErrorResume(Flux|Mono)`
 - ... y envolver y volver a lanzar: `.onErrorMap(t → new RuntimeException(t))(Flux|Mono)`
 - el bloque finalmente: `doFinally(Flux|Mono)`
 - el patrón de uso de Java 7: `using(Flux|Mono)` método de fábrica

© JMA 2021. All rights reserved

Trabajar con el tiempo

- Par asociar las emisiones con un tiempo medido
 - ... con la mejor precisión y versatilidad disponibles de los datos proporcionados: `timed(Flux|Mono)`
 - ... desde el último `onNext`: `elapsed(Flux|Mono)`
 - ... desde una marca temporal: `timestamp(Flux|Mono)`
- Para que se interrumpa la secuencia si hay demasiado retraso entre las emisiones: `timeout(Flux|Mono)`
- Para obtener tics de un reloj, intervalos de tiempo regulares: `Flux#interval`
- Para emitir un simple 0 después de un retraso inicial: `static Mono.delay`.
- Para introducir un retraso:
 - entre cada señal `onNext`: `Mono#delayElement` , `Flux#delayElements`
 - antes de que ocurra la suscripción: `delaySubscription(Flux|Mono)`

© JMA 2021. All rights reserved

Regresar al modo síncrono

- Para en un `Flux<T>`:
 - bloquear hasta que pueda obtener el primer elemento: `Flux#blockFirst`
 - bloquear hasta que pueda obtener el último elemento (o nulo si está vacío): `Flux#blockLast`
 - cambiar a un `Iterable<T>` síncrono: `Flux#toIterable`
 - cambiar a un Java 8 `Stream<T>` síncrono: `Flux#toStream`
- Para en un `Mono<T>`:
 - bloquear hasta que pueda obtener el valor: `Mono#block`
 - cambiar a un `CompletableFuture<T>`: `Mono#toFuture`

© JMA 2021. All rights reserved

Dividir un flujo

- Para dividir un Flux <T> en un Flux<Flux<T>>, según un criterio de límite de tamaño, duración o predicado: `window`, `windowTimeout`, `windowUntil`, `windowWhile`, `windowWhen`.
- Para dividir un Flux <T> en búfer de elementos de dentro de los límites juntos en una lista o colección por límite de tamaño, duración o predicado: `buffer`, `bufferTimeout`, `bufferUntil`, `bufferWhile`, `bufferWhen`.
- Para dividir un Flux <T> para que el elemento que comparte una característica termine en el mismo subflujo: `groupBy`

© JMA 2021. All rights reserved

Pasar a multidifusión de un flujo a varios suscriptores

- Para conectar varios suscriptores a un flujo:
 - y decidir cuándo activar la fuente con `connect()`: `publish()` (devuelve un `ConnectableFlux`)
 - y activar la fuente inmediatamente (los suscriptores tardíos ven datos posteriores): `share() (Flux | Mono)`
 - y conectar permanentemente la fuente cuando se hayan registrado suficientes suscriptores: `.publish().autoConnect (n)`
 - y conectar y cancelar automáticamente la fuente cuando los suscriptores superen o bajen del umbral: `.publish().refCount (n)`
- Para almacenar en caché los datos de un publicador y reproducirlos para suscriptores posteriores hasta `n` elementos o de los últimos elementos vistos dentro de una duración: `cache(Flux|Mono)`, pero sin activar inmediatamente la fuente: `Flux#replay` (devuelve un `ConnectableFlux`)

© JMA 2021. All rights reserved

Operador transform

- Desde una perspectiva de código limpio, la reutilización de código es generalmente algo bueno. El operador transform permite encapsular una parte de una cadena de operadores en una función.
- Esa función se aplica a una cadena de operadores original en el momento del ensamblaje para aumentarla con los operadores encapsulados. Hacerlo aplica las mismas operaciones a todos los suscriptores de una secuencia y es básicamente equivalente a encadenar a los operadores directamente.

```
Function<Flux<String>, Flux<String>> filterAndMap =  
    flujo -> flujo.filter(color -> !color.equals("orange")).map(String::toUpperCase);  
  
Flux.fromIterable(Arrays.asList("blue", "green", "orange", "purple"))  
    .doOnNext(System.out::println)  
    .transform(filterAndMap)  
    .subscribe(d -> System.out.println("Subscriber to Transformed MapAndFilter: "+d));
```

© JMA 2021. All rights reserved

<http://spring.io>

SPRING CON SPRING BOOT

© JMA 2021. All rights reserved

Spring

- Inicialmente era un ejemplo hecho para el libro “J2EE design and development” de Rod Johnson en 2003, que defendía alternativas a la “visión oficial” de aplicación JavaEE basada en EJBs.
- Actualmente es un framework open source que facilita el desarrollo de aplicaciones java JEE & JSE (no está limitado a aplicaciones Web, ni a java pueden ser .NET, Silverlight, Windows Phone, etc.)
- Provee de un contenedor encargado de manejar el ciclo de vida de los objetos (beans) para que los desarrolladores se enfoquen a la lógica de negocio. Permite integración con diferentes frameworks.
- Surge como una alternativa a EJB's
- Actualmente es un framework completo compuesto por múltiples módulos/proyectos que cubre todas las capas de la aplicación, con decenas de desarrolladores y miles de descargas al día.

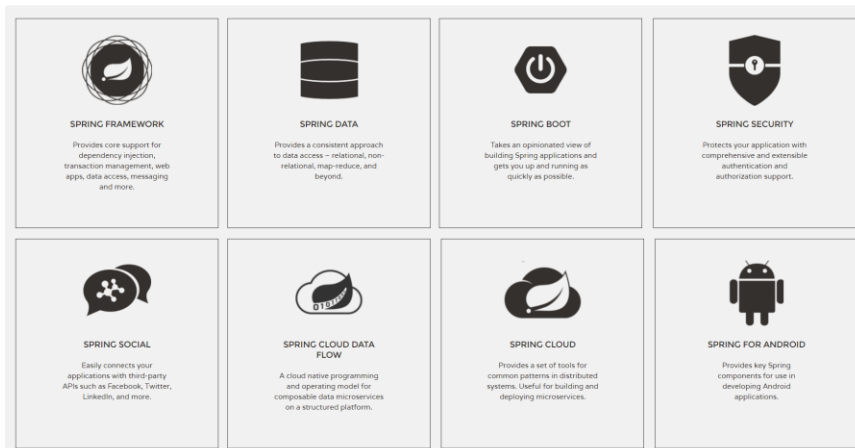
© JMA 2021. All rights reserved

Características

- Ligero
 - No se refiere a la cantidad de clases sino al mínimo impacto que se tiene al integrar Spring.
- No intrusivo
 - Generalmente los objetos que se programan no tienen dependencias de clases específicas de Spring
- Flexible
 - Aunque Spring provee funcionalidad para manejar las diferentes capas de la aplicación (vista, lógica de negocio, acceso a datos) no es necesario usarlo para todo. Brinda la posibilidad de utilizarlo en la capa o capas que queramos.
- Multiplataforma
 - Escrito en Java, corre sobre JVM

© JMA 2021. All rights reserved

Proyectos



© JMA 2021. All rights reserved

Módulos necesarios

- Spring Framework
 - Spring Core
 - Contenedor IoC (inversión de control) - inyector de dependencia
 - Spring MVC
 - Framework basado en MVC para aplicaciones web y servicios REST
- Spring Data
 - Simplifica el acceso a los datos: JPA, bases de datos relacionales / NoSQL, nube
- Spring Boot
 - Simplifica el desarrollo de Spring: inicio rápido con menos codificación

© JMA 2021. All rights reserved

Spring Boot

- Spring Boot es una herramienta que nace con la finalidad de simplificar aun más el desarrollo de aplicaciones basadas en el framework Spring Core: que el desarrollador solo se centre en el desarrollo de la solución, olvidándose por completo de la compleja configuración que actualmente tiene Spring Core para poder funcionar.
 - Configuración: Spring Boot cuenta con un complejo módulo que autoconfigura todos los aspectos de nuestra aplicación para poder simplemente ejecutar la aplicación, sin tener que definir nada.
 - Resolución de dependencias: Con Spring Boot solo hay que determinar que tipo de proyecto estaremos utilizando y el se encarga de resolver todas las librerías/dependencias para que la aplicación funcione.
 - Despliegue: Spring Boot se puede ejecutar como una aplicación Stand-alone, pero también es posible ejecutar aplicaciones web, ya que es posible desplegar las aplicaciones mediante un servidor web integrado, como es el caso de Tomcat, Jetty o Undertow.
 - Métricas: Por defecto, Spring Boot cuenta con servicios que permite consultar el estado de salud de la aplicación, permitiendo saber si la aplicación está encendida o apagada, memoria utilizada y disponible, número y detalle de los Beans creados por la aplicación, controles para el prendido y apagado, etc.
 - Extensible: Spring Boot permite la creación de complementos, los cuales ayudan a que la comunidad de Software Libre cree nuevos módulos que faciliten aún más el desarrollo.
 - Productividad: Herramientas de productividad para desarrolladores como LiveReload y Auto Restart, funcionan en su IDE favorito: Spring Tool Suite, IntelliJ IDEA y NetBeans.

© JMA 2021. All rights reserved

Dependencias: starters

- Los starters son un conjunto de descriptores de dependencias convenientes (versiones compatibles, ya probadas) que se pueden incluir en la aplicación.
- Se obtiene una ventanilla única para el módulo de Spring y la tecnología relacionada que se necesita, sin tener que buscar a través de códigos de ejemplo y copiar/pegar cargas de descriptores de dependencia.
- Por ejemplo, si desea comenzar a utilizar Spring con JPA para un acceso CRUD a base de datos, basta con incluir la dependencia `spring-boot-starter-data-jpa` en el proyecto.

© JMA 2021. All rights reserved

Spring Tools

- <https://spring.io/tools>
- Spring Tool Suite (STS) es un IDE basado en la versión Java EE de Eclipse, pero altamente personalizable para trabajar con Spring Framework.
 - IDE gratuito, personalización del Eclipse
- Spring Tools 4 es la próxima generación de herramientas Spring para los entornos de codificación favoritos. Proporciona soporte de primera clase para el desarrollo de aplicaciones empresariales basadas en Spring, ya sea que se prefiera Eclipse, Visual Studio Code o Theia IDE.
 - Help → Eclipse Marketplace ...
 - Spring Tools 4 for Spring Boot

© JMA 2021. All rights reserved

Crear proyecto

- Desde web:
 - <https://start.spring.io/>
 - Descomprimir en el workspace
 - Import → Maven → Existing Maven Project
- Desde Eclipse:
 - New Project → Spring Boot → Spring Started Project
- Dependencias
 - Spring Reactive Web
 - Spring Boot DevTools
 - Lombok
 - Spring Data Reactive MongoDB
 - Validation

© JMA 2021. All rights reserved

Application

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(ApiHrApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        // Opcional: Procesar los args una vez arrancado SprintBoot
    }
}
```

© JMA 2021. All rights reserved

Configuración

- **@Configuration**: Indica que esta es una clase usada para configurar el contenedor Spring.
- **@ComponentScan**: Escanea los paquetes de nuestro proyecto en busca de los componentes que hayamos creado, ellos son, las clases que utilizan las siguientes anotaciones: **@Component**, **@Service**, **@Controller**, **@Repository**.
- **@EnableAutoConfiguration**: Habilita la configuración automática, esta herramienta analiza el classpath y el archivo `application.properties` para configurar nuestra aplicación en base a las librerías y valores de configuración encontrados, por ejemplo: al encontrar el motor de bases de datos H2 la aplicación se configura para utilizar este motor de datos, al encontrar Thymeleaf se crearan los beans necesarios para utilizar este motor de plantillas para generar las vistas de nuestra aplicación web.
- **@SpringBootApplication**: Es el equivalente a utilizar las anotaciones: **@Configuration**, **@EnableAutoConfiguration** y **@ComponentScan**

© JMA 2021. All rights reserved

Configuración

- Editar `src/main/resources/application.properties`:
server.port=\${PORT:8080}
Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=hr
spring.datasource.password=hr
spring.datasource.driver-class=oracle.jdbc.driver.OracleDriver

MySQL settings
spring.datasource.url=jdbc:mysql://localhost:3306/sakila
spring.datasource.username=root
spring.datasource.password=root

MongoDB
spring.data.mongodb.uri=mongodb://localhost:27017/curso

logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n
logging.level.org.hibernate.SQL=debug
- Repetir con `src/test/resources/application.properties`
- Eclipse: Run Configurations → Arguments → VM Arguments: `-DPORT=8888`

© JMA 2021. All rights reserved

Instalación con Docker

- Docker Toolbox
 - Windows 10 Pro ++: <https://docs.docker.com/docker-for-windows/install/>
 - Otras: <https://github.com/docker/toolbox/releases>
- Ejecutar Docker QuickStart
- Para crear el contenedor de MySQL con la base de datos Sakila:
 - `docker run -d --name mysql-sakila -e MYSQL_ROOT_PASSWORD=root -p 3306:3306 1maa/sakila:latest`
- Para crear el contenedor de MongoDB:
 - `docker run -d --name mongodb -p 27017:27017 mongo`
- Para crear el contenedor de Redis:
 - `docker run --name redis -p 6379:6379 -d redis`
 - `docker run -d --name redis-commander -p 8081:8081 rediscommander/redis-commander:latest`
- Para crear el contenedor de RabbitMQ:
 - `docker run -d --hostname rabbitmq --name rabbitmq -p 4369:4369 -p 5671:5671 -p 5672:5672 -p 15671:15671 -p 15672:15672 -p 25672:25672 rabbitmq:3-management`

© JMA 2021. All rights reserved

Spring Reactive Web

SPRING WEBFLUX

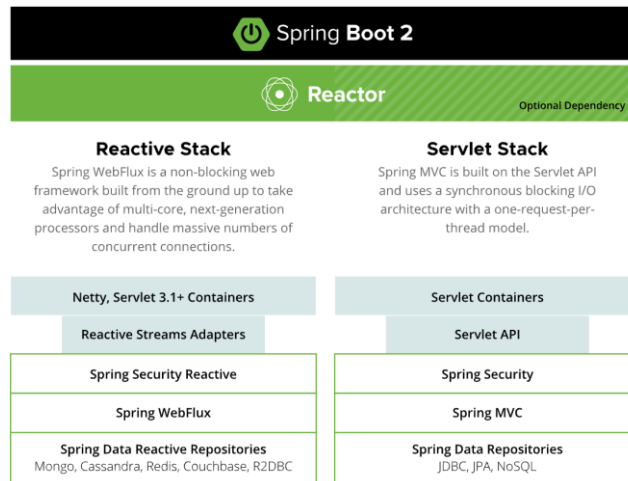
© JMA 2021. All rights reserved

Introducción

- El marco web original incluido en Spring Framework, Spring Web MVC, fue diseñado específicamente para la API de Servlet y los contenedores de Servlet. El marco web de pila reactiva, Spring WebFlux, se agregó más adelante en la versión 5.0 con soporte reactivo no bloqueante con contrapresión.
- Una de las principales razones por las que los desarrolladores pasan del código bloqueante al código no bloqueante es la eficiencia. El código reactivo hace más trabajo con menos recursos. Project Reactor y Spring WebFlux permiten a los desarrolladores aprovechar los procesadores multinúcleo de próxima generación, que manejan cantidades potencialmente masivas de conexiones simultáneas. Con el procesamiento reactivo, se puede satisfacer a más usuarios simultáneos con menos instancias de microservicio.
- Es importante acceder a los datos y procesarlos de forma reactiva. MongoDB, Redis y Cassandra tienen soporte reactivo nativo en Spring Data. Muchas bases de datos relacionales (Postgres, Microsoft SQL Server, MySQL, H2 y Google Spanner) tienen soporte reactivo a través del proyecto Reactive Relational Database Connectivity (R2DBC). En el mundo de la mensajería, Spring Cloud Stream también admite el acceso reactivo a plataformas como RabbitMQ y Kafka.
- Así mismo, deben ejecutarse en servidores sin bloqueo, como los contenedores Netty, Tomcat, Jetty o Undertow, o que tengan soporte para Servlet 3.1+. Esto y tener que disponer del acceso reactivo a base de datos limita los escenarios de uso.

© JMA 2021. All rights reserved

Stack



© JMA 2021. All rights reserved

Reactor

- Project Reactor es una base totalmente no bloqueante con soporte de contrapresión incluido. Es la base de la pila reactiva en el ecosistema Spring y se presenta en proyectos como Spring WebFlux, Spring Data y Spring Cloud Gateway.
- Reactor proporciona los tipos Mono y Flux para trabajar en secuencias de datos de 0..1 (Mono) o 0..N (Flux) elementos a través de un rico conjunto de operadores alineados con el vocabulario de operadores de ReactiveX.
- Reactor es una biblioteca de Flujos reactivos y, por lo tanto, todos sus operadores admiten contrapresión sin bloqueo. Reactor tiene un fuerte enfoque en Java del lado del servidor. Se desarrolla en estrecha colaboración con Spring.

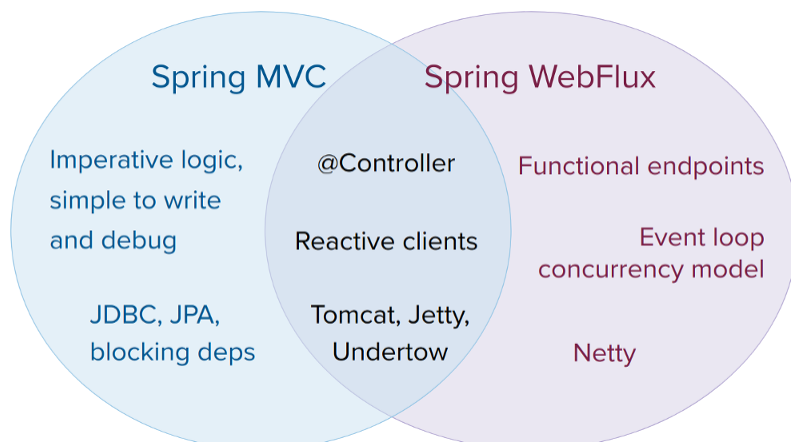
© JMA 2021. All rights reserved

Modelos de programación

- Spring WebFlux ofrece dos modelos de programación:
 - Controladores anotados: coherentes con Spring MVC y basados en las mismas anotaciones del módulo Spring MVC. Tanto los controladores Spring MVC como WebFlux admiten tipos de retorno reactivos (Reactor y RxJava). WebFlux también admite `@RequestBody` y argumentos reactivos.
 - Puntos finales funcionales: modelo de programación funcional, ligero y basado en Lambda. Se puede pensar en esto como una pequeña biblioteca o un conjunto de utilidades que una aplicación puede usar para enrutar y manejar solicitudes. La gran diferencia con los controladores anotados es que la aplicación está a cargo del manejo de solicitudes de principio a fin en lugar de declarar la intención a través de anotaciones y ser devuelto a la llamada.

© JMA 2021. All rights reserved

Spring MVC vs Spring WebFlux



© JMA 2021. All rights reserved

Controladores anotados

- Spring WebFlux proporciona un modelo de programación basado en anotación, donde los componentes utilizan anotaciones `@Controller` y `@RestController` para expresar asignaciones de petición, de entrada solicitud, excepciones y demás. Los controladores anotados tienen firmas de métodos flexibles y no tienen que extender clases base ni implementar interfaces específicas.
- Permite definir el mapeo con las anotaciones `@RequestMapping`, `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping` y `@PatchMapping` siguiendo las convenciones de patrones de Spring MVC.
- Así mismo, soporta la inyección de parámetros con las mismas anotaciones de Spring MVC.
- Es en el `@ResponseBody` donde aparecen la diferencias al admitir los tipos reactivos:
 - `Flux<T>`, `Mono<T>`, `ResponseEntity<Mono<T>>` o `Mono<ResponseEntity<T>>`, `Observable<ServerSentEvent>`, `Flux<ServerSentEvent>` y otros tipos reactivos.

© JMA 2021. All rights reserved

Recursos

- Son clases Java con la anotación `@RestController` (`@Controller` + `@ResponseBody`) y representan a los servicios REST, son controller que reciben y responden a las peticiones de los clientes.
`@RestController`

```
public class PaisController {  
    @Autowired  
    private PaisRepository paisRepository;
```
- Los métodos de la clase que interactúan con el cliente deben llevar la anotación `@RequestMapping`, con la subruta y el `RequestMethod`.

```
@RequestMapping(value = "/países/{id}", method = RequestMethod.GET)  
public Mono<ResponseEntity<Pais>> getToDoById(@PathVariable("id") String id) {  
    return Mono.just(new ResponseEntity<Pais>(paisRepository.findById(id).get(), HttpStatus.OK));  
}
```

© JMA 2021. All rights reserved

RequestMapping

- La anotación `@RequestMapping` permite asignar solicitudes a los métodos de los controladores.
- Tiene varios atributos para definir URL, método HTTP, parámetros de solicitud, encabezados y tipos de medios.
- Se puede usar a el nivel de clase para expresar asignaciones compartidas o a el nivel de método para limitar a una asignación de endpoint específica.
- También hay atajos con el método HTTP predefinido:
 - `@GetMapping`
 - `@PostMapping`
 - `@PutMapping`
 - `@DeleteMapping`
 - `@PatchMapping`

© JMA 2021. All rights reserved

Patrones de URI

- Establece que URLs serán derivadas al controlador.
- Puede asignar solicitudes utilizando los siguientes patrones globales y comodines:
 - ? Coincide con un carácter
 - * Coincide con cero o más caracteres dentro de un segmento de ruta
 - ** Coincide con cero o más segmentos de ruta hasta el final de la ruta. Solo puede ir al final del patrón.
 - {param} Coincide con un segmento de ruta (*) y lo captura como un parámetro
 - Con {param:\d+} debe coincidir con la expresión regular
 - Con {*param} captura hasta el final de la ruta. Solo puede ir al final del patrón.
- También puede declarar variables en la URI y acceder a sus valores con anotando con `@PathVariable` los parámetros, debe respetarse la correspondencia de nombres:

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public Mono<Pet> findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
    // ...
}
```

© JMA 2021. All rights reserved

Restricciones

- **consumes**: formatos MIME permitidos del encabezado Content-type
`@PostMapping(path = "/pets", consumes = "application/json")`
`public void addPet(@RequestBody Pet pet) {`
- **produces** : formatos MIME permitidos del encabezado Accept
`@GetMapping(path = "/pets/{petId}", produces = "application/json;charset=UTF-8")`
`@ResponseBody`
`public Mono<Pet> getPet(@PathVariable String petId) {`
- **params**: valores permitidos de los QueryParams
- **headers**: valores permitidos de los encabezados
`@GetMapping(path = "/pets/{petId}", params = "myParam=myValue", headers = "myHeader=myValue")`
`public void findPet(@PathVariable String petId) {`

© JMA 2021. All rights reserved

Inyección de Parámetros

- El API decodifica la petición e inyecta los datos como parámetros en el método.
- Es necesario anotar los parámetros para indicar la fuente del dato a inyectar.
- En las anotaciones será necesario indicar el nombre del origen en caso de no existir correspondencia de nombres con el de los parámetros.
- El tipo de origen, en la mayoría de los casos, es String que puede discrepar con los tipos de los parámetros, en tales casos, la conversión de tipo se aplica automáticamente en función de los convertidores configurados.
- Spring WebFlux, a diferencia de Spring MVC, admite tipos reactivos en el modelo, por ejemplo, `Mono<Integer> id`, sin embargo hay que tener en cuenta que hay que tratarlo como contenedor reactivo.
- Por defecto los parámetros son obligatorios, se puede indicar que sean opcionales, se inicializan a null si no reciben en la petición salvo que se indique el valor por defecto:

`@RequestParam(required=false, defaultValue="1")`

© JMA 2021. All rights reserved

Inyección de Parámetros

Anotación	Descripción
@PathVariable	Para acceder a las variables de la plantilla URI.
@MatrixVariable	Para acceder a pares nombre-valor en segmentos de ruta URI.
@RequestParam	Para acceder a los parámetros de solicitud del Servlet (QueryString o Form), incluidos los archivos de varias partes. Los valores de los parámetros se convierten al tipo de argumento del método declarado.
@RequestHeader	Para acceder a las cabeceras de solicitud. Los valores de encabezado se convierten al tipo de argumento del método declarado.
@CookieValue	Para el acceso a las cookies. Los valores de las cookies se convierten al tipo de argumento del método declarado.

© JMA 2021. All rights reserved

Inyección de Parámetros

Anotación	Descripción
@RequestBody	Para acceder al cuerpo de la solicitud HTTP. El contenido del cuerpo se convierte al tipo de argumento del método declarado utilizando implementaciones <code>HttpMessageConverter</code> .
@RequestPart	Para acceder a una parte en una solicitud multipart/form-data, convertir el cuerpo de la parte con un <code>HttpMessageConverter</code> .
@ModelAttribute	Para acceder a un atributo existente en el modelo (instanciado si no está presente) con enlace de datos y validación aplicada.
@SessionAttribute	Para acceder a cualquier atributo de sesión, a diferencia de los atributos de modelo almacenados en la sesión como resultado de una declaración <code>@SessionAttributes</code> de nivel de clase.
@RequestAttribute	Para acceder a los atributos de solicitud.

© JMA 2021. All rights reserved

Inyección de Parámetros

```
// http://localhost:8080/params/1?nom=kk
```

```
@GetMapping("/params/{id}")
public Mono<String> cotilla(
    @PathVariable String id,
    @RequestParam String nom,
    @RequestHeader("Accept-Language") String language,
    @CookieValue("JSESSIONID") String cookie) {
    StringBuilder sb = new StringBuilder();
    sb.append("id: " + id + "\n");
    sb.append("nom: " + nom + "\n");
    sb.append("language: " + language + "\n");
    sb.append("cookie: " + cookie + "\n");
    return Mono.just(sb.toString());
}
```

© JMA 2021. All rights reserved

Respuesta

- La anotación `@ResponseBody` (incluida en el `@RestController`) en un método indica que el retorno será serializado en el cuerpo de la respuesta a través de un `HttpMessageConverter`.

```
@PostMapping("/invierte")
@ResponseBody
public Mono<Punto> body(@RequestBody Punto p) {
    int x = p.getX();
    p.setX(p.getY());
    p.setY(x);
    return Mono.just(p);
}
```

- El código de estado de la respuesta se puede establecer con la anotación `@ResponseStatus`:
`@PostMapping`
`@ResponseStatus(HttpStatus.CREATED)`
`public void add(@RequestBody Punto p) { ... }`

© JMA 2021. All rights reserved

Respuesta personalizada

- La clase `ResponseEntity` permite agregar estado y encabezados a la respuesta (no requiere la anotación `@ResponseBody`).

```
@GetMapping(value="/pais")
public Mono<ResponseEntity<List<Pais>>> getAll(){
    return Mono.just(new ResponseEntity<List<Pais>>>(
        paisRepository.findAll(),
        HttpStatus.OK));
}
```

- La clase `ResponseEntity` dispone de builder para generar la respuesta:

```
return Mono.just(ResponseEntity.ok().eTag(etag).build(body));
```

© JMA 2021. All rights reserved

Respuestas asíncronas

- `Mono<T>`: Respuesta de tipo reactivo de valor único.
- `Flux<T>`: Respuesta de tipo reactivo de valor múltiple.
- `ResponseEntity<Mono<T>>` o `ResponseEntity<Flux<T>>`: hace que el estado de la respuesta y los encabezados se conozcan de inmediato, mientras que el cuerpo se proporciona de forma asíncrona en un momento posterior.
- `Mono<ResponseEntity<T>>`: proporciona las respuesta: estado de respuesta, encabezados y cuerpo, de forma asíncrona en un momento posterior. Esto permite que el estado de respuesta y los encabezados varíen según el resultado del manejo de solicitudes asíncronas.
- `Mono<ResponseEntity<Mono<T>>>` o `Mono<ResponseEntity<Flux<T>>>`: son otra alternativa posible, aunque menos común: proporcionan el estado de respuesta y los encabezados de forma asíncrona primero y luego el cuerpo de la respuesta, también de forma asíncrona, en segundo lugar.

© JMA 2021. All rights reserved

Mapecto de respuestas genéricas a excepciones.

- Un requisito común para los servicios REST es incluir detalles de error en el cuerpo de la respuesta.
- Spring Framework no lo hace automáticamente porque la representación de los detalles de error en el cuerpo de la respuesta es específica de la aplicación.
- Una clase `@RestController` puede contar con métodos anotados con `@ExceptionHandler` que intercepten determinadas excepciones producidas en el resto de los métodos de la clase y devuelven un `ResponseEntity` que permite establecer el estado y el cuerpo de la respuesta.
- Esto mismo se puede hacer globalmente en clases anotadas con `@ControllerAdvice` que solo tienen los correspondientes métodos `@ExceptionHandler`.
- `@RestControllerAdvice` es una anotación compuesta que se anota con `@ControllerAdvice` y `@ResponseBody`, lo que esencialmente significa que los métodos `@ExceptionHandler` se representan en el cuerpo de la respuesta a través de la conversión del mensaje (en comparación con la resolución de la vista o la representación de la plantilla).

© JMA 2021. All rights reserved

Excepciones personalizadas

```
public class NotFoundException extends Exception {
    private static final long serialVersionUID = 1L;
    public NotFoundException() {
        super("NOT FOUND");
    }
    public NotFoundException(String message) {
        super(message);
    }
    public NotFoundException(Throwable cause) {
        super("NOT FOUND", cause);
    }
    public NotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    public NotFoundException(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```

© JMA 2021. All rights reserved

Error Personalizado

```
public class ErrorMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    private String error, message;
    public ErrorMessage(String error, String message) {
        this.error = error;
        this.message = message;
    }
    public String getError() { return error; }
    public void setError(String error) { this.error = error; }
    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }
}
```

© JMA 2021. All rights reserved

@RestControllerAdvice

```
@RestControllerAdvice
public class ApiExceptionHandler {
    @ExceptionHandler({NotFoundException.class})
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public Mono<ErrorMessage> notFoundRequest(ServerHttpRequest request, Exception exception) {
        return Mono.just(new ErrorMessage(exception.getMessage(), request.getURI().toString()));
    }

    @ExceptionHandler({ BadRequestException.class, MissingRequestHeaderException.class,
        FileNotFoundException.class, InvalidDataException.class, MethodArgumentNotValidException.class
    })
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public Mono<ErrorMessage> badRequest(Exception exception) {
        return Mono.just(new ErrorMessage(exception.getMessage(), ""));
    }
}
```

© JMA 2021. All rights reserved

Servicio Web RESTful

```
@RestController
@RequestMapping("/api/actores")
public class ActorResource {
    @Autowired
    private ActorReactiveRepository dao;

    @Autowired
    private Validator validator;

    @GetMapping
    public Flux<Actor> getAll() {
        // ...
    }

    @GetMapping(path =("/{id}")
    public Mono<Actor> getOne(@PathVariable int id) throws NotFoundException {
        // ...
    }
}
```

© JMA 2021. All rights reserved

Servicio Web RESTful

```
@PostMapping
public Mono<ResponseEntity<Object>> create(@Valid @RequestBody Actor item) throws BadRequestException {
    // ...
    URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
        .buildAndExpand(newItem.getActorId()).toUri();
    return Mono.just(ResponseEntity.created(location).build());
}

@PutMapping("/{id}")
@ResponseStatus(HttpStatus.ACCEPTED)
public void update(@PathVariable int id, @Valid @RequestBody Actor item) throws BadRequestException, NotFoundException {
    // ...
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void delete(@PathVariable int id) {
    // ..
}
}
```

© JMA 2021. All rights reserved

Puntos finales funcionales

- Spring WebFlux incluye WebFlux.fn, un modelo de programación funcional liviano en el que las funciones se utilizan para enrutar y manejar solicitudes, y los contratos están diseñados para la inmutabilidad.
- En WebFlux.fn, una solicitud HTTP se maneja con una función de controlador (HandlerFunction): una función que toma ServerRequest y devuelve un ServerResponse diferido (es decir Mono<ServerResponse>). Tanto la solicitud como el objeto de respuesta tienen contratos inmutables que ofrecen acceso compatible con JDK 8 a la solicitud y respuesta HTTP. HandlerFunction es el equivalente del cuerpo de un método @RequestMapping en el modelo de programación basado en anotaciones.
- Las solicitudes entrantes se enrutan a una función de controlador con una RouterFunction: una función que toma ServerRequest y devuelve un HandlerFunction diferido (es decir Mono<HandlerFunction>). Cuando la función del enrutador coincide, se devuelve una función de controlador; de lo contrario, un Mono vacío. RouterFunction es el equivalente a una anotación @RequestMapping, pero con la principal diferencia de que las funciones del enrutador proporcionan no solo datos, sino también comportamiento.

© JMA 2021. All rights reserved

HandlerFunction

- ServerRequest proporciona acceso al método HTTP, URI, encabezados y parámetros de consulta, mientras que el acceso al cuerpo se proporciona a través de los métodos body.

```
String id = request.pathVariable("id");  
Mono<Persona> item = request.bodyToMono(Persona.class);
```
- ServerResponse proporciona acceso a la respuesta HTTP y, dado que es inmutable, se puede utilizar un método build para crearla o se puede utilizar el constructor para establecer el estado de la respuesta, agregar encabezados de respuesta o proporcionar un cuerpo.

```
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(item,  
    Persona.class);  
ServerResponse.created(location).build();  
ServerResponse.noContent().build();  
ServerResponse.notFound().build()
```

© JMA 2021. All rights reserved

HandlerFunction

```
@Service
public class PersonasHandler {
    @Autowired
    private PersonasReactiveRepository dao;
    public Mono<ServerResponse> getAll(ServerRequest request) {
        return ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(dao.findAll(), Persona.class);
    }
    public Mono<ServerResponse> getOne(ServerRequest request) {
        return dao.findById(request.pathVariable("id")).flatMap(item -> ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).bodyValue(item))
            .switchIfEmpty(ServerResponse.notFound().build());
    }
    public Mono<ServerResponse> create(ServerRequest request) {
        return request.bodyToMono(Persona.class).flatMap(item -> ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(dao.save(item),
            Persona.class));
    }
    public Mono<ServerResponse> update(ServerRequest request) {
        return request.bodyToMono(Persona.class).flatMap(item -> ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(dao.save(item),
            Persona.class));
    }
    public Mono<ServerResponse> delete(ServerRequest request) {
        return dao.deleteById(request.pathVariable("id")).then(ServerResponse.noContent().build());
    }
}
```

© JMA 2021. All rights reserved

Rutas

- Las rutas se definen como Beans con las funciones del enrutador que se evalúan en orden: si la primera ruta no coincide, se evalúa la segunda y así sucesivamente. Por lo tanto, tiene sentido declarar rutas más específicas antes que las generales. Esto también es importante al registrar las funciones del enrutador como beans. Hay que tener en cuenta que este comportamiento es diferente del modelo de programación basado en anotaciones, donde el método de controlador "más específico" se selecciona automáticamente.

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {
    @Bean
    public RouterFunction<?> routerPersonas(PersonasHandler handler) {
        return RouterFunctions.route()
            .GET("/personas/{id}", RequestPredicates.accept(MediaType.APPLICATION_JSON), handler::getOne)
            .GET("/personas", RequestPredicates.accept(MediaType.APPLICATION_JSON), handler::getAll)
            .POST("/personas", RequestPredicates.contentType(MediaType.APPLICATION_JSON), handler::create)
            .PUT("/personas/{id}", RequestPredicates.contentType(MediaType.APPLICATION_JSON), handler::update)
            .DELETE("/personas/{id}", handler::delete)
            .build();
    }
}
```

© JMA 2021. All rights reserved

WebClient

- Spring WebFlux incluye un cliente para realizar solicitudes HTTP. WebClient tiene una API fluida y funcional basada en Reactor, que permite la composición declarativa de lógica asíncronica sin la necesidad de lidiar con subprocesos o simultaneidad. Sin bloqueo, admite transmisión y se basa en los mismos códecs que también se utilizan para codificar y decodificar el contenido de solicitudes y respuestas en el lado del servidor.
- El método `retrieve()` se puede utilizar para declarar cómo extraer la respuesta.

```
WebClient client = WebClient.create("https://example.org");
Mono<ResponseEntity<Person>> result = client.get()
    .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .toEntity(Person.class);
```

© JMA 2021. All rights reserved

WebClient

- Los métodos `exchangeToMono()` y `exchangeToFlux()` son útiles para casos más avanzados que requieren más control, como para decodificar la respuesta de manera diferente según el estado de la respuesta:

```
Mono<Object> entityMono = client.get()
    .uri("/persons/1")
    .accept(MediaType.APPLICATION_JSON)
    .exchangeToMono(response -> {
        if (response.statusCode().equals(HttpStatus.OK)) {
            return response.bodyToMono(Person.class);
        } else if (response.statusCode().is4xxClientError()) {
            return response.bodyToMono(ErrorContainer.class); // Suppress error status code
        } else {
            return response.createException().flatMap(Mono::error); // Turn to error
        }
    });
```

© JMA 2021. All rights reserved

WebClient

- El cuerpo de la solicitud se puede codificar desde cualquier tipo asincrónico manejado por ReactiveAdapterRegistry:

```
Mono<Person> personMono = ... ;  
Mono<Void> result = client.put()  
    .uri("/persons/{id}", id)  
    .contentType(MediaType.APPLICATION_JSON)  
    .body(personMono, Person.class)  
    .retrieve()  
    .bodyToMono(Void.class);
```

© JMA 2021. All rights reserved

Spring Cloud LoadBalancer

- Un balanceador o equilibrador de carga fundamentalmente es un dispositivo de hardware o software que se interpone al frente de un conjunto de servidores que atienden una aplicación y, tal como su nombre lo indica, asigna o reparte las solicitudes que llegan de los clientes a los servidores usando algún algoritmo (desde un simple round-robin hasta algoritmos más sofisticados).
- Spring Cloud proporciona su propia abstracción e implementación del equilibrador de carga del lado del cliente. Para el mecanismo de equilibrio de carga, ReactiveLoadBalancer, se ha agregado una interfaz y se le han proporcionado implementaciones basadas en Round-Robin y Random.
- El balanceo de carga se basa en el descubrimiento de servicios que recupera mediante un cliente de descubrimiento disponible en la ruta de clases, como Spring Cloud Netflix Eureka, Spring Cloud Consul Discovery o Spring Cloud Zookeeper Discovery .
- Spring Cloud LoadBalancer puede integrarse con:
 - Spring RestTemplate como cliente de equilibrador de carga
 - Spring WebClient como cliente de equilibrador de carga
 - Spring OpenFeign como cliente de equilibrador de carga
 - Spring WebFlux WebClient con ReactorLoadBalancerExchangeFilterFunction
- Dependencia: Spring Cloud Routing > Cloud LoadBalancer

© JMA 2021. All rights reserved

Spring Cloud LoadBalancer

- Spring Cloud LoadBalance permite:
 - Cambiar entre los algoritmos de equilibrio de carga
 - Almacenamiento en caché
 - Equilibrio de carga basado en zonas
 - Comprobación del estado de las instancias (HealthCheck)
 - Establecer preferencia de Misma instancia, Sesión fija basada en solicitudes, basado en sugerencias.
 - Transformar la solicitud HTTP en el proceso de equilibrio de carga antes de enviarla.

© JMA 2021. All rights reserved

Cientes Load Balancer

- Se puede configurar WebClient para usar el ReactiveLoadBalancer. Si se agrega el iniciador Spring Cloud LoadBalancer al proyecto y si spring-webflux está en la ruta de clase, se configura automáticamente el filtro ReactorLoadBalancerExchangeFilterFunction.

```
public class MyClass {  
    @Autowired  
    private ReactorLoadBalancerExchangeFilterFunction lbFunction;  
  
    public Mono<String> doOtherStuff() {  
        return WebClient.builder().baseUrl("http://stores")  
            .filter(lbFunction)  
            .build()  
            .get().uri("/stores")  
            .retrieve().bodyToMono(String.class);  
    }  
}
```

- La URI debe utilizar un nombre de host virtual (es decir, un nombre de servicio, no un nombre de host). Se utiliza ReactorLoadBalancer para crear una dirección física completa.

© JMA 2021. All rights reserved

Spring Cloud Circuit Breaker con Resilience4j

- El disyuntor Spring Cloud proporciona una abstracción a través de diferentes implementaciones de disyuntores. Proporciona una API coherente para usar en las aplicaciones, lo que le permite al desarrollador elegir la implementación de disyuntor que mejor se adapte a sus necesidades para su aplicación.
- Las siguientes implementaciones son compatibles:
 - Netflix Hystrix (spring-cloud-starter-netflix-hystrix)
 - Resilience4J (spring-cloud-starter-circuitbreaker-resilience4j)
 - Sentinel (spring-cloud-starter-circuitbreaker-spring-retry)
 - Spring Retry (spring-cloud-starter-circuitbreaker-sentinal)

© JMA 2021. All rights reserved

Resilience4j

- Resilience4j es una biblioteca de tolerancia a fallos liviana y fácil de usar inspirada en Netflix Hystrix, pero diseñada para Java 8 y programación funcional. Liviana, porque la biblioteca solo usa Vavr, que no tiene ninguna otra dependencia de biblioteca externa
- Resilience4j proporciona funciones de orden superior (decoradores) para mejorar cualquier interfaz funcional, expresión lambda o referencia de método con un disyuntor, limitador de velocidad, reintento o mamparo, permitiendo concatenar más de un decorador. La ventaja es que se tiene la opción de seleccionar los decoradores que necesita y nada más.
- Los patrones soportados para aumentar la tolerancia a fallos debido a problemas de red o fallo de alguno de los múltiples servicios son:
 - Circuit breaker: para dejar de hacer peticiones cuando un servicio invocado está fallando.
 - Retry: realiza reintentos cuando un servicio ha fallado de forma temporal.
 - Bulkhead: limita el número de peticiones concurrentes salientes a un servicio para no sobrecargarlo.
 - Rate limit: limita el número de llamadas que recibe un servicio en un periodo de tiempo.
 - Cache: intenta obtener un valor de la cache y si no está presente de la función de la que lo recupera.
 - Time limiter: limita el tiempo de ejecución de una función para no esperar indefinidamente a una respuesta.

© JMA 2021. All rights reserved

Resilience4j

- Instalación: Spring Cloud Circuit Breaker > Resilience4j
- Para proporcionar una configuración predeterminada para todos los disyuntores:

```
@Bean
public Customizer<ReactiveResilience4JCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new Resilience4JConfigBuilder(id)
        .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())
        .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(4)).build()).build());
}
```

- De manera similar, se puede proporcionar una configuración personalizada:

```
@Bean
public Customizer<ReactiveResilience4JCircuitBreakerFactory> slowCustomizer() {
    return factory -> {
        factory.configure(builder -> builder
            .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(2)).build())
            .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults()), "slow", "slowflux");
        factory.addCircuitBreakerCustomizer(circuitBreaker -> circuitBreaker.getEventPublisher()
            .onError(normalFluxErrorConsumer).onSuccess(normalFluxSuccessConsumer), "normalflux");
    };
}
```

© JMA 2021. All rights reserved

Resilience4j

- Se puede configurar las instancias de CircuitBreaker e TimeLimiter en el archivo de propiedades de configuración de la aplicación.

```
resilience4j.circuitbreaker:
instances:
  backendA:
    registerHealthIndicator: true
    slidingWindowSize: 100
  backendB:
    registerHealthIndicator: true
    slidingWindowSize: 10
    permittedNumberOfCallsInHalfOpenState: 3
    slidingWindowType: TIME_BASED
resilience4j.timelimiter:
instances:
  backendA:
    timeoutDuration: 2s
    cancelRunningFuture: true
  backendB:
    timeoutDuration: 1s
    cancelRunningFuture: false
```

© JMA 2021. All rights reserved

Resilience4j

- Si resilience4j-bulkhead está en el classpath, Spring Cloud CircuitBreaker ajustará todos los métodos con un mamparo Resilience4j Bulkhead.
- Spring Cloud CircuitBreaker Resilience4j proporciona dos implementaciones de patrón de mamparo (bulkhead):
 - SemaphoreBulkhead: que usa semáforos (predeterminado)
 - FixedThreadPoolBulkhead: que usa una cola limitada y un grupo de subprocesos fijo.
- El Customizer<Resilience4jBulkheadProvider> permite proporcionar una configuración predeterminada para Bulkhead y ThreadPoolBulkhead.

```
@Bean
public Customizer<Resilience4jBulkheadProvider> defaultBulkheadCustomizer() {
    return provider -> provider.configureDefault(id -> new Resilience4jBulkheadConfigurationBuilder()
        .bulkheadConfig(BulkheadConfig.custom().maxConcurrentCalls(4).build())
        .threadPoolBulkheadConfig(ThreadPoolBulkheadConfig.custom().coreThreadPoolSize(1)
            .maxThreadPoolSize(1).build())
        .build());
}
```

© JMA 2021. All rights reserved

Resilience4j

- Spring Retry proporciona compatibilidad con reintentos declarativos para aplicaciones Spring. Spring Retry proporciona una implementación de disyuntor mediante una combinación de él CircuitBreakerRetryPolicy y reintentos con estado . Todos los disyuntores creados con Spring Retry se crearán con CircuitBreakerRetryPolicy y un DefaultRetryState. Ambas clases se pueden configurar usando SpringRetryConfigBuilder.
- Para proporcionar una configuración predeterminada para todos los disyuntores:

```
@Bean
public Customizer<SpringRetryCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new SpringRetryConfigBuilder(id)
        .retryPolicy(new TimeoutRetryPolicy()).build());
}
```

© JMA 2021. All rights reserved

ANEXOS

© JMA 2021. All rights reserved

Enlaces

- Introducción
 - <https://www.reactivemanifesto.org/>
 - <https://refactoring.guru/es/design-patterns>
 - <https://docs.microsoft.com/es-es/azure/architecture/patterns/publisher-subscriber>
- Reactive Streams
 - <https://www.reactive-streams.org/>
 - <https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Flow.html>
 - <https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.3/README.md>
- Reactor Project
 - <https://github.com/reactor/reactor>
 - <https://projectreactor.io/>
- Spring:
 - <https://spring.io/projects>
- Spring WebFlux
 - <https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html>
- Ejemplos:

© JMA 2021. All rights reserved

GenerateData

- GenerateData es una herramienta para la generación automatizada de juegos de datos.
- Ofrece ya una serie de datos precargados en BBDD y un conjunto de tipos de datos bastante amplio, así como la posibilidad de generar tipos genéricos.
- Podemos elegir en que formato se desea la salida de entre los siguientes:
 - CSV
 - Excel
 - HTML
 - JSON
 - LDIF
 - Lenguajes de programación (JavaScript, Perl, PHP, Ruby, C#)
 - SQL (MySQL, Postgres, SQLite, Oracle, SQL Server)
 - XML
- Online: <http://www.generatedata.com/?lang=es>
- Instalación (PHP): <http://benkeen.github.io/generatedata/>

© JMA 2021. All rights reserved

Mockaroo

- <https://www.mockaroo.com/>
- Mockaroo permite descargar rápida y fácilmente grandes cantidades de datos realistas de prueba generados aleatoriamente en función de sus propias especificaciones que luego puede cargar directamente en su entorno de prueba utilizando formatos CSV, JSON, XML, Excel, SQL, ... No se requiere programación y es gratuita (para generar datos de 1.000 en 1.000).
- Los datos realistas son variados y contendrán caracteres que pueden no funcionar bien con nuestro código, como apóstrofes o caracteres unicode de otros idiomas. Las pruebas con datos realistas harán que la aplicación sea más robusta porque detectará errores en escenarios de producción.
- El proceso es sencillo ya que solo hay que ir añadiendo nombres de campos y escoger su tipo. Por defecto nos ofrece mas de 140 tipos de datos diferentes que van desde nombre y apellidos (pudiendo escoger estilo, género, etc...) hasta ISBNs, ubicaciones geográficas o datos encriptados simulados.
- Además es posible hacer que los datos sigan una distribución Normal o de Poisson, secuencias, que cumplan una expresión regular o incluso que fueren cadenas complicadas con caracteres extraños y cosas así. Tenemos la posibilidad de crear fórmulas propias para generarlos, teniendo en cuenta otros campos, condicionales, etc... Es altamente flexible.

© JMA 2021. All rights reserved