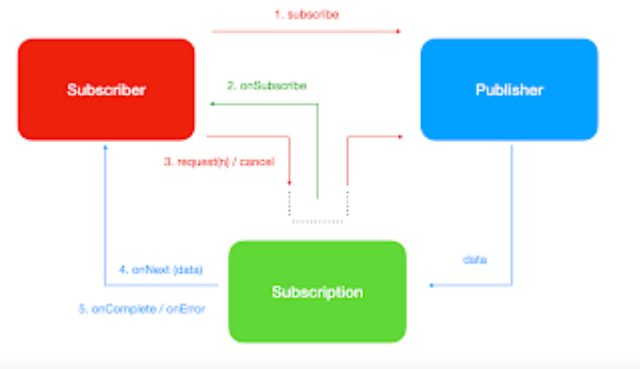
XORCERY IMPLEMENTA REACTIVE STREAMS - PARTE 1

on **noviembre 06, 2023** in **xorcery** with **No hay comentarios.**

¿QUÉ ES REACTIVE STREAMS?



Para poder entender como nos permite **Xorcery** trabajar con **reactive streams**, tenemos que saber que es Reactive Streams.

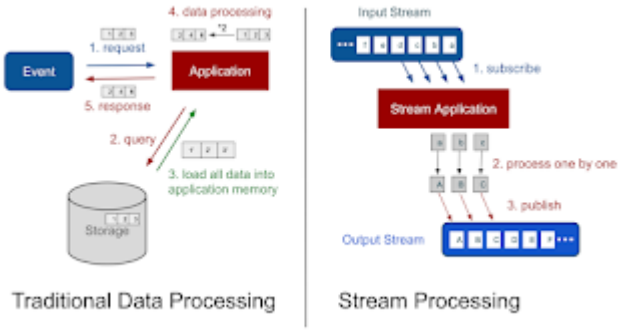
La página oficial de Reactive Streams define lo siguiente:

> *Reactive Streams is a standard for asynchronous data processing in a streaming fashion with non-blocking back pressure.*

Entonces se tiene que entender que es "**processing in a streaming fashion**", "**asynchronous**", "**back pressure**", y "**standard**".

Procesamiento de Stream

La figura abajo compara el procesamiento de datos tradicional vs. el procesamiento de streams.



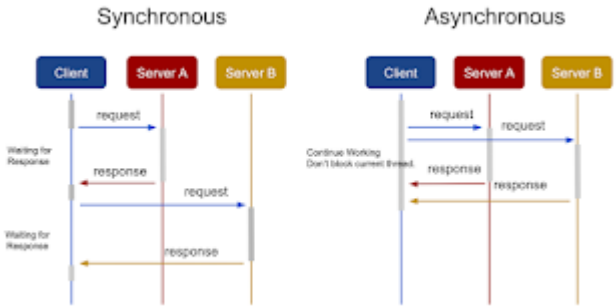
El método tradicional de datos a la izquierda en cada request/response va guardando la data que es consultada en la base de datos en la memoria de la aplicación. Si el tamaño de la data solicitada es más grande que el tamaño de memoria disponible, un error "**out of memory**" se va a producir como resultado. Hay otro escenario donde el servicio o aplicación recibe muchos requests simultáneos y se activa una gran cantidad de GC (recolección de basura) en un corto periodo de tiempo, lo que provoca que el servidor no responda de forma normal.

Por otro lado, se tiene el **método de procesamiento de streams**, donde se puede procesar grandes cantidades de datos con poca memoria del sistema. Con este tipo de procesamiento, se puede crear un pipeline que se suscribe a cualquier data entrante, se procesa la data, y luego se publica dicha data.

Gracias a esto, el servidor es capaz de procesar grandes cantidades de datos de forma elástica.

Método Asíncrono

Vamos a comparar el método asíncrono con el método síncrono. La siguiente figura muestra el proceso de ambos métodos:



En el **método síncrono**, un request enviado por el cliente es bloqueado hasta que el servidor envía una respuesta. Ser "bloqueado" significa que el hilo actual no puede ejecutar otra tarea y tiene que pasar a un estado de espera. Si dos requests son enviados al servidor A y B, el request debe recibir una respuesta desde el servidor A antes de moverse al servidor B. Sin embargo, con el **método asíncrono**, el hilo actual no es bloqueado y puede ejecutar otras tareas sin esperar una respuesta. El hilo puede ser usado por otras tareas después de enviar un request al servidor A, o enviar un separado request al servidor B. La ventaja de los métodos asíncronos comparados con los métodos síncronos son:

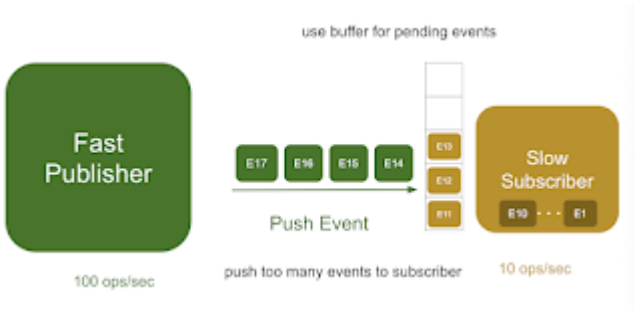
- **Velocidad** - tú puedes obtener respuestas rápidas enviando dos requests simultáneos
- **Menos uso de recursos** - se puede procesar más requests con menos hilos, ya que los hilos pueden ejecutar tareas en paralelo sin ser bloqueados.

Contrapresión

Antes de entrar en más detalles sobre la "contrapresión", veamos el patrón observador, método push y método pull que hizo famoso RxJava.

Método Push

En el **patrón observador**, un **publicador** transfiere data pusheando está a un **suscriptor**. Si el publicador envía 100 mensajes por segundo, y el suscriptor solo puede procesar 10 mensajes en 1 segundo, ¿qué pasaría? El suscriptor tendrá que guardar los eventos pendientes en una cola.



La cantidad de memoria del servidor es limitada. Si 100 mensajes por segundos es enviada al servidor, el buffer se llenará instantáneamente.

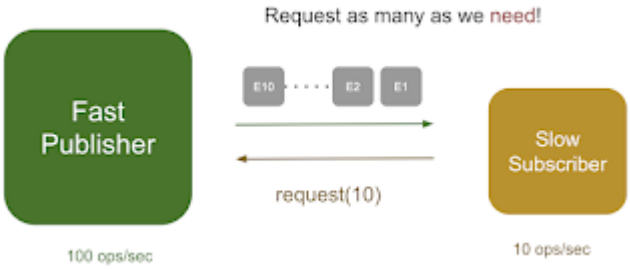
Sí, se tiene un **buffer estático**, los nuevos mensajes serán rechazados, estos tendrán que enviarse nuevamente y eso causará procesamiento adicional y carga de red y CPU.

Si se tiene un **buffer de longitud variable**, el servidor en algún momento tendrá un error "out of memory" debido al intento de guardar los eventos.

¿Cómo solucionamos este problema? ¿Podemos tener un publicador que solo envía una cantidad de mensajes que el suscriptor puede manejar? Esto es lo que se llama **contrapresión**.

Método Pull

Con el método pull, un suscriptor puede procesar 10 operaciones a la vez con solo 10 operaciones requests al publicador. El publicador puede enviar la cantidad solicitada, y el suscriptor está seguro de no tener ningún error "out of memory".



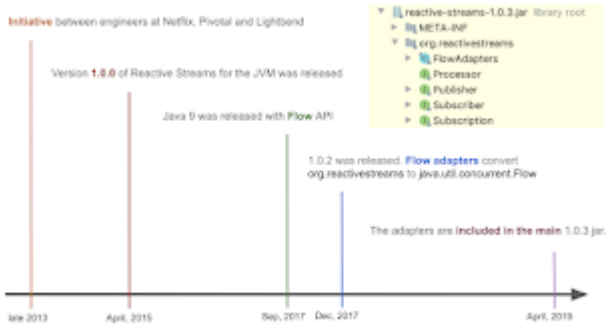
Además, si un mismo suscriptor está procesando actualmente 8 operaciones, puede solicitar 2 operaciones más para que el número de mensajes no supere los límites de lo que puede procesar. Con el método pull, los suscriptores tienen la libertad de elegir el tamaño de los datos que reciben. El método que permite a los suscriptores extraer dinámicamente solicitudes de datos dentro de su capacidad es la contrapresión.

Estandarización

Reactive Streams es un API estandarizado.

El desarrollo de Reactive Streams comenzó el 2013 por ingenieros de Netflix, Pivotal y Lightbend. Netflix es responsable del desarrollo de RxJava, Pivotal para WebFlux y Lightbend de Akka, una implementación de modelos de actores de procesamiento distribuido. Lo que estas empresas tenían en común era que todas requerían API de streaming. Sin embargo, los streams solo tienen sentido si se combinan y fluyen orgánicamente. Para que los datos fluyeran ininterrumpidamente, estas diferentes empresas necesitaban utilizar especificaciones compartidas o, en otras palabras, necesitaban estandarización.

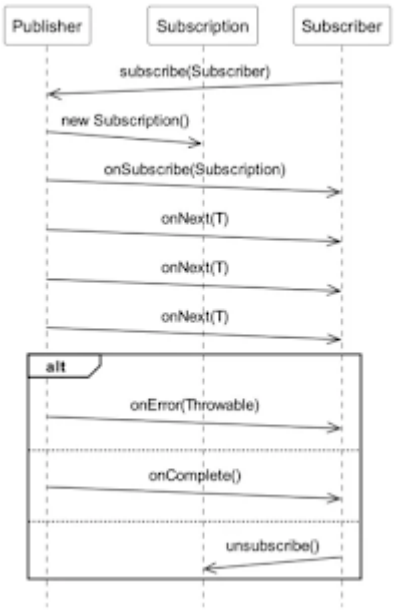
Durante abril de 2015, Reactive Streams lanzó sus especificaciones 1.0.0 que se pueden utilizar en JVM. En septiembre de 2017, Java 9 agregó el Flow API, que incluye la API, las especificaciones y el método de extracción de Reactive Streams y lo empaquetó en **java.util.concurrent**. Reactive Streams, que fue un esfuerzo compartido entre miembros de la comunidad y algunas empresas, ha sido reconocido oficialmente y agregado como parte oficial de Java. Tres meses después, Reactive Streams lanzó un adaptador que es compatible con Flow, lo que permite utilizar bibliotecas existentes.



Reactive Extensions

Reactive Extensions (Reactive X) es una familia de frameworks multi plataforma para manejar streams de eventos sincronos o asincronos, originalmente creados por Erik Meijer en Microsoft. La implementación de Reactive Extensions para java es el Netflix RxJava framework.

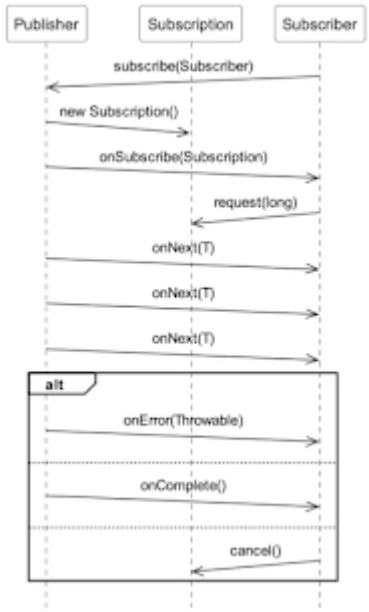
En terminos simples, las Reactive Extensions son una combinación del patrón Observer y el patrón iterator y programación funcional. Gracias al patrón Observer, se toma la habilidad de que los consumidores se suscriban a los eventos del productor. Gracias al patrón Iterator, se toma la habilidad de manejar los tres tipos de eventos de stream (data, error, completion). Gracias a la programación funcional se toma la habilidad de manejar los eventos de streams con métodos encadenados (filter, transform, combine, etc).



Reactive Streams

Es un desarrollo adicional de Reactive Extensiones, en el cual se usa backpressure para lograr un balance de performance entre el productor y consumidor. **En terminos simples es una combinación de Reactive Extensiones y batching**.

La principal diferencia entre ellos es quién inicia el intercambio. En Reactive Extensions, un productor envía eventos a un suscriptor tan pronto como estén disponibles y en cualquier número. **En Reactive Streams, un productor debe enviar eventos a un suscriptor solo después de que hayan sido solicitados y no más del número solicitado**.



Ventajas:

- El consumidor puede iniciar el intercambio en cualquier momento.
- El consumidor puede parar el intercambio en cualquier momento.
- El consumidor puede determinar cuando el productor terminará de generar eventos
- La latencia es menor que la comunicación sincrona pull porque el producto envía los eventos al consumidor, tan pronto, cómo están disponibles.
- El consumidor puede uniformemente manejar eventos de streams de tres tipos (data, error, completion).
- Manejar eventos de streams con métodos encadenados que son mas sencillos que implementar controladores de eventos anidados.
- Implementar productores y consumidores concurrentes no es una tarea sencilla.

Desventajas:

- Un consumidor lento puede ser sobrecargado con eventos, debido a un productor rapido.
- Implementar productores y consumidores concurrentes no es una tarea simple.

Especificación Reactive Streams

Reactive Streams es una especificación que proporciona un estándar para el procesamiento de flujos asincrónicos con contrapresión sin bloqueo para diversos entornos de ejecución (JVM, .NET y JavaScript) y protocolos de red. La especificación Reactive Streams fue creada por ingenieros de Kaazing, Lightbend, Netflix, Pivotal, Red Hat, Twitter y otros.

La especificación describe el concepto de flujos reactivos que tienen las siguientes características:

- Los flujos reactivos pueden ser de unidifusión y multidifusión: un editor puede enviar eventos a uno o varios consumidores.
- Los flujos reactivos son potencialmente infinitos: pueden manejar cero, uno, muchos o un número infinito de eventos.
- Los flujos reactivos son secuenciales: un consumidor procesa eventos en el mismo orden en que los envía un productor.
- Los flujos reactivos pueden ser síncronos o asíncronos: pueden utilizar recursos informáticos para el procesamiento paralelo en etapas separadas.
- Los flujos reactivos no son bloqueantes: no desperdician recursos informáticos si el rendimiento de un productor y un consumidor son diferentes.
- Los flujos reactivos utilizan contrapresión obligatoria: un consumidor puede solicitar eventos a un productor de acuerdo con su velocidad de procesamiento.
- Los flujos reactivos utilizan buffers acotados: se pueden implementar sin buffers ilimitados, evitando errores de falta de memoria.

La especificación Reactive Streams para JVM (la última versión 1.0.4 se lanzó el 26 de mayo de 2022) contiene la especificación textual y la API de Java, que contiene cuatro interfaces que deben implementarse de acuerdo con esta especificación. También incluye el kit de compatibilidad tecnológica (TCK), un conjunto de pruebas estándar para pruebas de conformidad de implementaciones.

API Reactive Streams

La API de Reactive Streams consta de cuatro interfaces, que se encuentran en el paquete **org.reactivestreams**:

- La interfaz Publisher<T> representa un productor de datos y eventos de control.
- La interfaz Subscriber<T> representa un consumidor de eventos.
- La interfaz de Suscripción representa una conexión entre un Publicador y un Suscriptor.
- La interfaz Processor<T,R> representa un procesador de eventos que actúa como suscriptor y publicador.

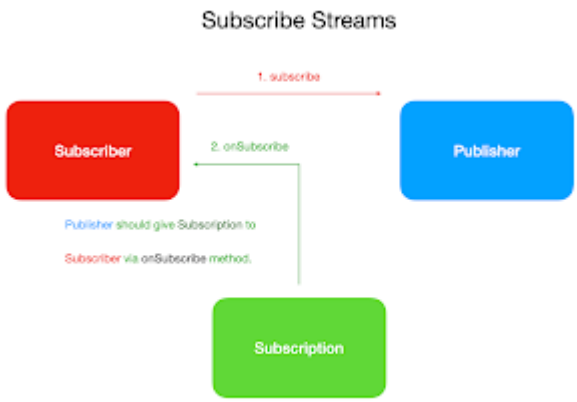
```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T item);
    public void onError(Throwable t);
    public void onComplete();
}

public interface Subscription {
    public void request(long n);
    public void cancel();
}

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

- Los **publishers** solo tienen una API **subscribe** que permite a los **subscribers** suscribirse.
- Los **Subscribers** tienen **onNext** para procesar la data recibida, **onError** para procesar errores, **onComplete** para completar tareas, y **onSubscribe** API para subscribirse con parámetros.
- Para **subscription**, hay una API **request** para requerir data y una API **cancel** para cancelar las **subscripciones**.

Ahora vamos a ver como la API es usada en Reactive Streams.

1. Un **subscriber** usa la función **subscribe** para solicitar una **subscription** a el **publisher**

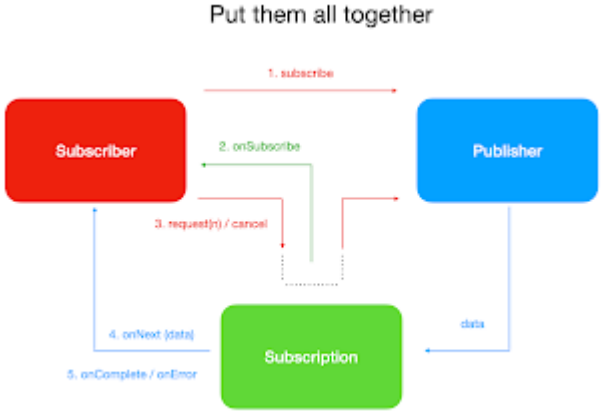
2. El **publisher** usa la función **onSubscribe** para enviar la **subscription** al **subscriber**.



3. La **subscription** ahora actúa como un medio entre un **subscriber** y **publisher**. Los **Subscribers** no solicitan la data directamente a los **publishers**. Los requests son enviados a los **publishers** usando la función **request** de la **subscription**.

4. El **publisher**, usando **subscription**, envía data con **onNext**, **onComplete** para tareas completadas, y **onError** para errores.

5. El **subscriber**, **publisher**, y **subscription** conforman una conexión organiza, comunicándose unos con otros; comenzando desde él **subscribe** hasta llegar al **onComplete**. Esto completa la estructura back pressure.



Entonces, la contrapresión parece útil, pero ¿cómo se usa realmente? La interfaz que ve arriba es toda la API de Reactive Streams disponible en su repositorio oficial de GitHub. No existe una implementación independiente que pueda utilizar.

¿Puedes implementarlo tú mismo? De hecho, puedes implementar una interfaz de **publisher** y generar una **subscription** utilizando las reglas anteriores. Sin embargo, esto no es todo. Reactive Streams viene con sus propias especificaciones y, a diferencia de una interfaz simple, estas especificaciones presentan reglas que deben seguirse durante la implementación.

Una vez que siga las especificaciones y tenga una implementación, puede validarla usando una herramienta llamada Reactive Streams TCK.

A menos que sea un experto en el campo, es difícil tener una implementación que satisfaga todas las reglas dadas. Especialmente el publisher es especialmente difícil de implementar.

El API JDK Flow

El JDK ha admitido la especificación Reactive Streams desde la versión 9 en forma de Flow API. La clase Flow contiene interfaces estáticas anidadas Publisher, Subscriber, Subscription y Processor, que son 100% semánticamente equivalentes a sus respectivas contrapartes de Reactive Streams. La especificación Reactive Streams contiene la clase FlowAdapters, que es un puente entre la API Reactive Streams (el paquete org.reactivestreams) y la API JDK Flow (la clase java.util.concurrent.Flow). La única implementación de la especificación Reactive Streams que JDK proporciona hasta ahora es la clase SubmissionPublisher que implementa la interfaz Publisher.

Conclusión

Antes de que Reactive Streams apareciera en el JDK, existían las API **CompletableFuture** y **Stream** relacionadas. La API CompletableFuture utiliza el modelo de comunicación push pero admite cálculos asincrónicos de un solo valor. La API Stream admite cálculos sincrónicos o asincrónicos de múltiples valores, pero utiliza el modelo de comunicación pull. Los flujos reactivos han ocupado un lugar vacante y admiten cálculos sincrónicos o asincrónicos de múltiples valores y también pueden cambiar dinámicamente entre los modelos de cálculo push y pull. Por lo tanto, los Reactive Streams son adecuados para procesar secuencias de eventos con velocidades impredecibles, como eventos de mouse y teclado, eventos de sensores y eventos de E/S vinculados a latencia de un archivo o red.

Fundamentalmente, los desarrolladores de aplicaciones no deberían implementar por sí mismos las interfaces de la especificación Reactive Streams. En primer lugar, la especificación es bastante compleja, especialmente en contratos asincrónicos, y no se puede implementar correctamente fácilmente. En segundo lugar, la especificación no contiene API para operaciones de flujo intermedio. En cambio, los desarrolladores de aplicaciones deberían implementar las etapas de flujo reactivo (productores, procesadores, consumidores) utilizando los marcos existentes (Lightbend Akka Streams, Pivotal Project Reactor, Netflix RxJava) con sus API nativas mucho más ricas. Deben usar la API Reactive Streams solo para combinar etapas heterogéneas en una única secuencia reactiva.

Si deseas buscar datos en repositorios usando Reactive Streams, puede usar ReactiveMongo o Slick.
Si necesitas algo relacionado con la programación web, se puede utilizar Armeria, Vert.x, Play Framework o Spring WebFlux.
Todas estas diferentes implementaciones pueden comunicarse entre sí a través de Reactive Streams.

Xorcery

¿Qué viene a aportar Xorcery en este ecosistema? Te invito a esperar la parte 2.

Enjoy!

José

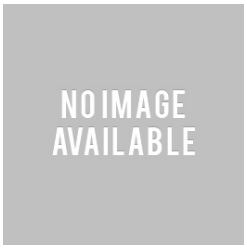
Share:

Related Posts:

Xorcery uses Jetty & Jersey and mTLS implementation and JWT support

Xorcery implementa Reactive Streams - Parte 1

Xorcery usa Jetty & Jersey e implementación de mTLS y soporte a JWT



Xorcery is born - Build high-performance microservices

Xorcery implements Reactive Streams - Parte 1

Entrada más reciente Página Principal Entrada antigua

0 comentarios:

Publicar un comentario

Para dejar un comentario, haz clic en el botón de abajo para acceder con Google.

ACCEDER CON GOOGLE

JoeDayz.pe

Cursos

About

Popular Posts

Xorcery implementa Reactive Streams - Parte 1

¿Qué es Reactive Streams? Para poder entender como nos permite Xorcery trabajar con reactive streams , tenemos que saber que es Reactive ...

Spring Boot + Thymeleaf + BootStrap

Categories

#github #java (1)

#guatejug (1)

#historiasdeprogramador (1)

aaii (1)

academia web (2)

acr (1)

Para este artículo vamos a ver como trabajar un administrador de clientes (abonados, inquilinos, miembros, etc) utilizando las siguientes t...

Blog Archive

grails (5)

groovy (3)

hangouts (1)

highchart-export-server (2)

huacho (1)

hudson (1)

hyperledger-composer (1)

hyperledger-fabric (1)

i-educa (1)

iBATIS (2)

icescrum (1)

informatica (1)

Intigas (1)

ITP_JAVA (1)

jakartaee (4)

jakartaee10 (1)

JasperReports (1)

java (3)

JavaCard (1)

JavaDayUNI (1)

JavaOne (1)

jhipster (2)

jmeter (1)

joedayz (46)

JOERP (4)

jpa (1)

jquery (1)

kafka (3)

kotlin (2)

Kubernetes (3)

lombok (1)

m2eclipse (1)

mac (2)

Matt Raible (1)

Maven (3)

microprofile (6)

microprofile-jwt jakartaee (2)

microprofile-jwt jdbc-realm jakartaee (1)

microprofile-jwt jdbc-realm payara (1)

microservicios (1)

Ministerio del Interior (1)

MJN (5)

móvil (1)

mysql (1)

namespaces (1)

navidad (1)

NET (4)

Nextel (1)

Novell (1)

ocjp (1)

Opentaps (2)

Oracle (1)

oraclecloud (1)

oraclefunctions (1)

oracleopenworld (1)

OSUM (1)

OSX (1)

p6spy (1)

Payara (5)

personal (1)

perujug jconfperu joedayz (2)

php (1)

play (1)

PMP (1)

podcasts (1)

PostgreSQL (8)

programacion (1)

pubsub (1)

PUCP (4)

quadim (2)

quarkus (1)

rackspace (1)

rails (2)

redis (1)

refactoring (2)

Reniec (1)

renovatebot (2)

Rider (1)

ruby (4)

rust (1)

scala (1)

SCD2010 (1)

SCJP (1)

Scrum (3)

Scrum evaluacion (1)

seminarios (1)

Setup (1)

SourceRepo (1)

spring (13)

spring 3.1 (1)

spring android (1)

spring mobile (1)

spring social (1)

spring-boot (9)

spring-boot-admin (2)

spring-cloud (1)

spring-cloud-config (2)

SpringCommunityDay (1)

SpringRoo (2)

springsource (1)

sqlserver (2)

start-up (1)

STS (1)

Subclipse (1)

Subversion (1)

SUN (1)

SUNAT (2)

synergyj (2)

Syscom (1)

Talleres (21)

Telefonica (1)

thedevconf (1)

thymeleaf (1)

Trac (1)

try-with-resources (1)

twitter (1)

Tye (1)

ubuntu (3)

UNI (3)

UNMSM (1)

UPC (1)

videos (1)

vimeo (1)

weblogic (2)

Workspace (1)

WPF (1)

xorcery (6)

xsd (1)

YaRetail (1)

YoMeQuedoEnCasa (1)

zuul (2)