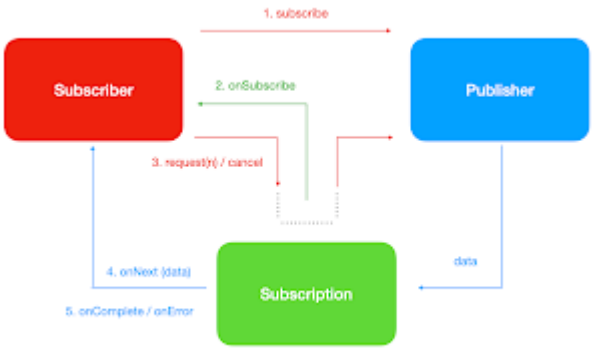
Xorcery implements Reactive Streams - Parte 1

on **noviembre 09, 2023** in **xorcery** with **No hay comentarios.**

¿What are Reactive Streams?



In order to understand how Xorcery allows us to work with reactive streams, we have to know what Reactive Streams are.

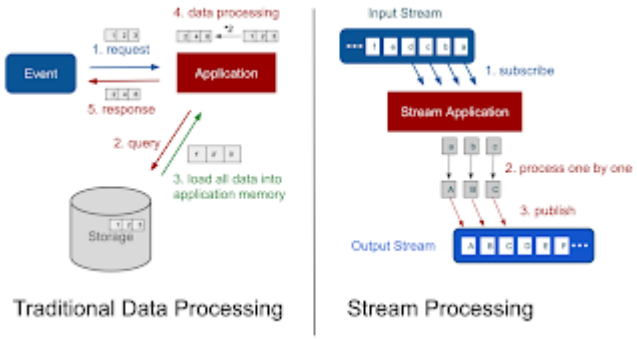
The official Reactive Streams page defines the following:

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

Then you have to understand what is "**processing in a streaming fashion**", "**asynchronous**", "**back pressure**", and "**standard**".

Stream Processing

The figure below compares traditional data processing vs. stream processing.

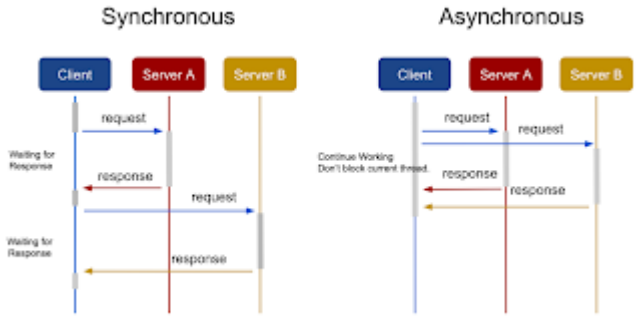


The traditional method on the left in each request/response saves the data that is queried in the database in the application memory. If the size of the requested data is larger than the size of available memory, an "**out of memory**" error will occur as a result. There is another scenario where the service or application receives many simultaneous requests and a large amount of GC (garbage collection) is triggered in a short period of time, causing the server to not respond normally.

On the other hand, there is **the stream processing method**, where large amounts of data can be processed with little system memory. With this type of processing, you can create a pipeline that subscribes to any incoming data, processes the data, and then publishes that data. Thanks to this, the server is capable of processing large amounts of data elastically.

Asynchronous Method

Let's compare the asynchronous method with the synchronous method. The following figure shows the process of both methods:



In the **synchronous method**, a request sent by the client is blocked until the server sends a response. Being "blocked" means that the current thread cannot execute another task and has to go into a waiting state. If two requests are sent to servers A and B, the request must receive a response from server A before moving to server B. However, with the **asynchronous method**, the current thread is not blocked and can execute other tasks without waiting for a response. The thread can be used for other tasks after sending a request to server A or sending a separate request to server B. The advantages of asynchronous methods compared to synchronous methods are:

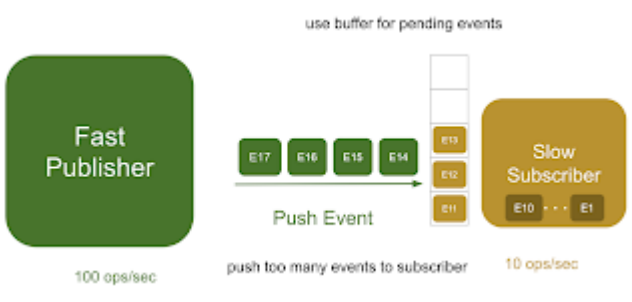
- **Speed** - you can get quick responses by sending two simultaneous requests
- **Less resource usage** - more requests can be processed with fewer threads since threads can execute tasks in parallel without being blocked.

Back pressure

Before going into more details about "back pressure", let's look at the observer pattern, push method, and pull method made famous by RxJava.

Push Method

In the **observer pattern**, a **publisher** transfers data by pushing it to a **subscriber**. If the publisher sends 100 messages per second, and the subscriber can only process 10 messages in 1 second, what would happen? The subscriber will have to save the pending events in a queue.



The amount of server memory is limited. If 100 messages per second are sent to the server, the buffer will fill instantly.

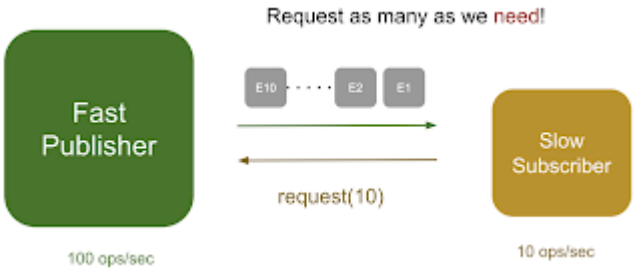
If you have a **static buffer**, new messages will be rejected, they will have to be sent again and that will cause additional processing and network and CPU load.

If you have a **variable length buffer**, the server will at some point have an "**out of memory**" error due to the attempt to save the events.

How do we solve this problem? Can we have a publisher that only sends a number of messages that the subscriber can handle? This is what is called **back pressure**.

Pull method

With the pull method, a **subscriber** can process 10 operations at a time with only 10 requests to the publisher. The publisher can send the requested amount, and the subscriber is sure not to have any "out of memory" errors.



Additionally, if the same subscriber is currently processing 8 operations, it can request 2 more operations so that the number of messages does not exceed the limits of what it can process. With the pull method, subscribers are free to

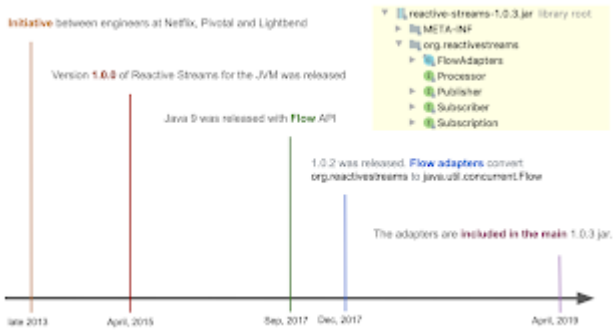
choose the size of the data they receive. The method that allows subscribers to dynamically pull data requests within their capacity is backpressure.

Standardization

Reactive Streams is a standardized API.

Development of Reactive Streams began in 2013 by engineers from Netflix, Pivotal, and Lightbend. Netflix is responsible for the development of RxJava, Pivotal for WebFlux, and Akka's Lightbend, an implementation of distributed processing actor models. What these companies had in common was that they all required streaming APIs. However, streams only make sense if they combine and flow organically. In order for data to flow uninterrupted, these different companies needed to use shared specifications, or in other words, they needed standardization.

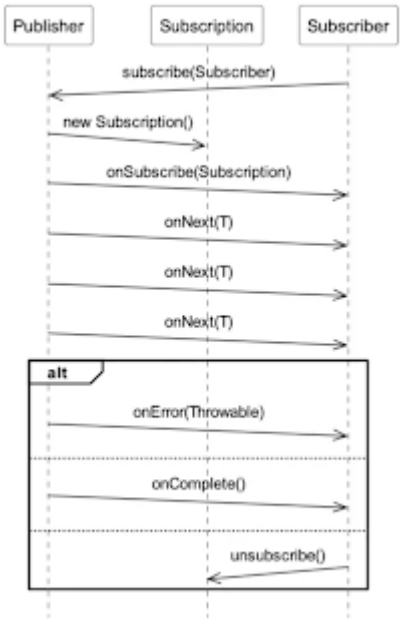
In April 2015, Reactive Streams released their 1.0.0 specifications that can be used in the JVM. In September 2017, Java 9 added the Flow API, which includes the Reactive Streams API, specifications, and pull method, and packaged it in **java.util.concurrent**. Reactive Streams, which was a shared effort between community members and some companies, has been officially recognized and added as an official part of Java. Three months later, Reactive Streams released an adapter that supports Flow, allowing existing libraries to be used.



Reactivate Extensions

Reactive Extensions (Reactive X) is a family of cross-platform frameworks for handling synchronous or asynchronous event streams, originally created by Erik Meijer at Microsoft. The Reactive Extensions implementation for Java is the Netflix RxJava framework.

In simple terms, Reactive Extensions are a combination of the Observer pattern, the iterator pattern, and functional programming. Thanks to the Observer pattern, the ability for consumers to subscribe to producer events is taken. Thanks to the Iterator pattern, the ability to handle the three types of stream events (data, error, completion) is taken. Thanks to functional programming, the ability to handle stream events with chained methods (filter, transform, combine, etc.) is acquired.

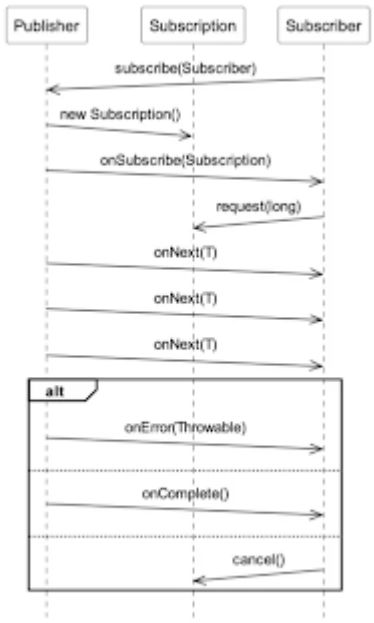


Reactivate Streams

It is an additional development of Reactive Extensions, in which backpressure is used to achieve a balance of performance between the producer and consumer. **In simple terms, it is a combination of Reactive Extensions and batching**.

The main difference between them is who initiates the exchange. In Reactive Extensions, a producer sends events to a subscriber as soon as they are available and in any number. **In Reactive Streams, a producer must send**

events to a subscriber-only after they have been requested and no more than the number requested.



Advantages:

- The consumer can initiate the exchange at any time.
- The consumer can stop the exchange at any time.
- The consumer can determine when the producer will finish generating events
- The latency is lower than synchronous pull communication because the product sends the events to the consumer as soon as they are available.
- The consumer can uniformly handle events from streams of three types (data, error, and completion).
- Handle stream events with chained methods that are simpler than implementing nested event handlers.
- Implementing concurrent producers and consumers is not a simple task.

Disadvantages:

- A slow consumer can be overloaded with events, due to a fast producer.
- Implementing concurrent producers and consumers is not a simple task.

Reactive Stream Specification

Reactive Stream is a specification that provides a standard for non-blocking backpressure asynchronous stream processing for various runtime environments (JVM, .NET, and JavaScript) and network protocols. The Reactive Streams specification was created by engineers from Kaazing, Lightbend, Netflix, Pivotal, Red Hat, Twitter, and others.

The specification describes the concept of reactive streams that have the following characteristics:

- Reactive streams can be unicast and multicast: a publisher can send events to one or more consumers.
- Reactive streams are potentially infinite: they can handle zero, one, many, or an infinite number of events.
- Reactive flows are sequential: a consumer processes events in the same order as a producer sends them.
- Reactive flows can be synchronous or asynchronous: they can use computing resources for parallel processing in separate stages.
- Reactive streams are non-blocking: they do not waste computing resources if the performance of a producer and a consumer are different.
- Reactive flows use mandatory backpressure: a consumer can request events from a producer according to its processing speed.
- Reactive flows use bounded buffers: they can be implemented without unbounded buffers, avoiding out-of-memory errors.

The Reactive Streams specification for JVM (latest version 1.0.4 was released on May 26, 2022) contains the textual specification and the Java API, which contains four interfaces that must be implemented according to this specification. It also includes the Technology Compatibility Kit (TCK), a standard test suite for conformance testing of implementations.

Reactive Streams API

The Reactive Streams API consists of four interfaces, found in the **org.reactivestreams** package:

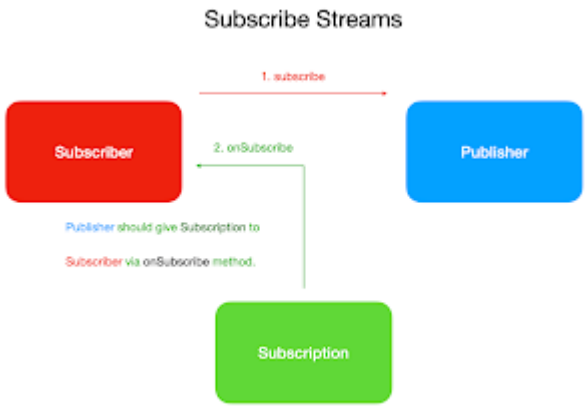
- The Publisher<T> interface represents a producer of data and control events.
- The Subscriber<T> interface represents an event consumer.
- The Subscription interface represents a connection between a Publisher and a Subscriber.
- The Processor<T,R> interface represents an event processor that acts as a subscriber and publisher.

```
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> {
}
```

- Publishers only have a subscribe API that allows subscribers to subscribe.
- Subscribers have onNext to process received data, onError to process errors, onComplete to complete tasks, and onSubscribe API to subscribe with parameters.
- For subscription, there is a request API to request data and a cancel API to cancel subscriptions.

Now let's see how the API is used in Reactive Streams.

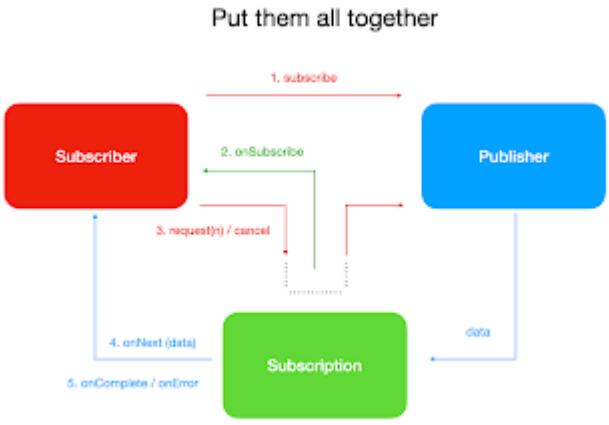
1. A **subscriber** uses the **subscribe** function to request a **subscription** from the **publisher**
2. The **publisher** uses the **onSubscribe** function to send the **subscription** to the **subscriber**.



3. The subscription now acts as a medium between a subscriber and publisher. Subscribers do not request data directly from publishers. Requests are sent to publishers using the subscription's request function.

4. The publisher, using subscription, sends data with onNext, onComplete for completed tasks, and onError for errors.

5. The subscriber, publisher, and subscription make up an organized connection, communicating with each other; starting from subscribe until reaching onComplete. This completes the back pressure structure.



So backpressure seems useful, but how is it actually used? The interface you see above is the entire Reactive Streams API available in its official GitHub repository. There is no standalone implementation that you can use.

Can you implement it yourself? In fact, you can implement a publisher interface and generate a subscription using the rules above. However, this is not all. Reactive Streams comes with its own specifications and unlike a simple interface, these specifications present rules that must be followed during implementation.

Once you follow the specifications and have an implementation, you can validate it using a tool called Reactive Streams TCK.

Unless you are an expert in the field, it is difficult to have an implementation that satisfies all the given rules. Especially the publisher is especially difficult to implement.

The JDK Flow API

The JDK has supported the Reactive Streams specification since version 9 in the form of the Flow API. The Flow class contains nested static interfaces Publisher, Subscriber, Subscription, and Processor, which are 100% semantically equivalent to their respective Reactive Streams counterparts.

The Reactive Streams specification contains the FlowAdapters class, which is a bridge between the Reactive Streams API (the **org.reactivestreams** package) and the JDK Flow API (the java.util.concurrent.Flow class). The only implementation of the Reactive Streams specification that JDK provides so far is the SubmissionPublisher class that implements the Publisher interface.

Conclusion

Before Reactive Streams appeared in the JDK, there were related **CompletableFuture** and **Stream** APIs. The CompletableFuture API uses the push communication model but supports asynchronous single-value calculations. The Stream API supports synchronous or asynchronous multi-value calculations, but uses the pull communication model. Reactive streams have taken a vacant spot and support synchronous or asynchronous multi-value calculations and can also dynamically switch between push and pull calculation models. Therefore, Reactive Streams are suitable for processing streams of events with unpredictable rates, such as mouse and keyboard events, sensor events, and latency-bound I/O events from a file or network.

Fundamentally, application developers should not implement the Reactive Streams specification interfaces themselves. First of all, the specification is quite complex, especially in asynchronous contracts, and cannot be easily implemented correctly. Second, the specification does not contain APIs for intermediate-stream operations. Instead, application developers should implement the reactive flow stages (producers, processors, consumers) using existing frameworks (Lightbend Akka Streams, Pivotal Project Reactor, Netflix RxJava) with their much richer native APIs. They should use the Reactive Streams API only to combine heterogeneous stages into a single reactive stream.

If you want to search data in repositories using Reactive Streams, you can use ReactiveMongo or Slick.
If you need something related to web programming, you can use Armeria, Vert.x, Play Framework, or Spring WebFlux.

All of these different implementations can communicate with each other through Reactive Streams.

Xorcery

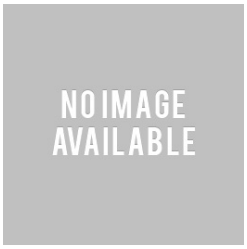
What does Xorcery contribute to this ecosystem? I invite you to wait for part 2.

Enjoy!

José

Share:

Related Posts:



Xorcery is born - Build high-performance microservices

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

Xorcery implements Reactive Streams - Parte 1

Xorcery uses Jetty & Jersey and mTLS implementation and JWT support

Xorcery implementa Reactive Streams - Parte 1

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

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

About

Categories

#github #java (1)

#guatejug (1)

#historiasdeprogramador (1)

aaii (1)

academia web (2)

acr (1)

Agile (2)

aks (2)

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

Blog Archive

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)

zool (2)