Reactive Programming

With Spring Framework 5

Reactive Streams API

Reactive Streams API

- Goal is to create a standard for asynchronous stream processing with non-blocking back pressure.
- Reactive Streams started in 2013 by engineers from Netflix, Pivotal, Lightbend (formerly Typesafe), Red Hat, Twitter, and Oracle.
- Reactive Streams is a set of 4 interfaces which define the API

Reactive Streams API

- On April 30th, 2015, version 1.0.0 of Reactive Streams was released.
- Under JEP-266, Reactive Streams is now part of the Java 9 JDK.
- Adoptions: Akka Streams, MongoDB, Ratpack, Reactive Rabbit, Project Reactor (Spring 5), RxJava, Slick 3.0, Vert.x 3.0, Cassandra, ElasticSearch, Kafka, Play
- On August 9th, 2017 version 1.0.1 of Reactive Streams was released.

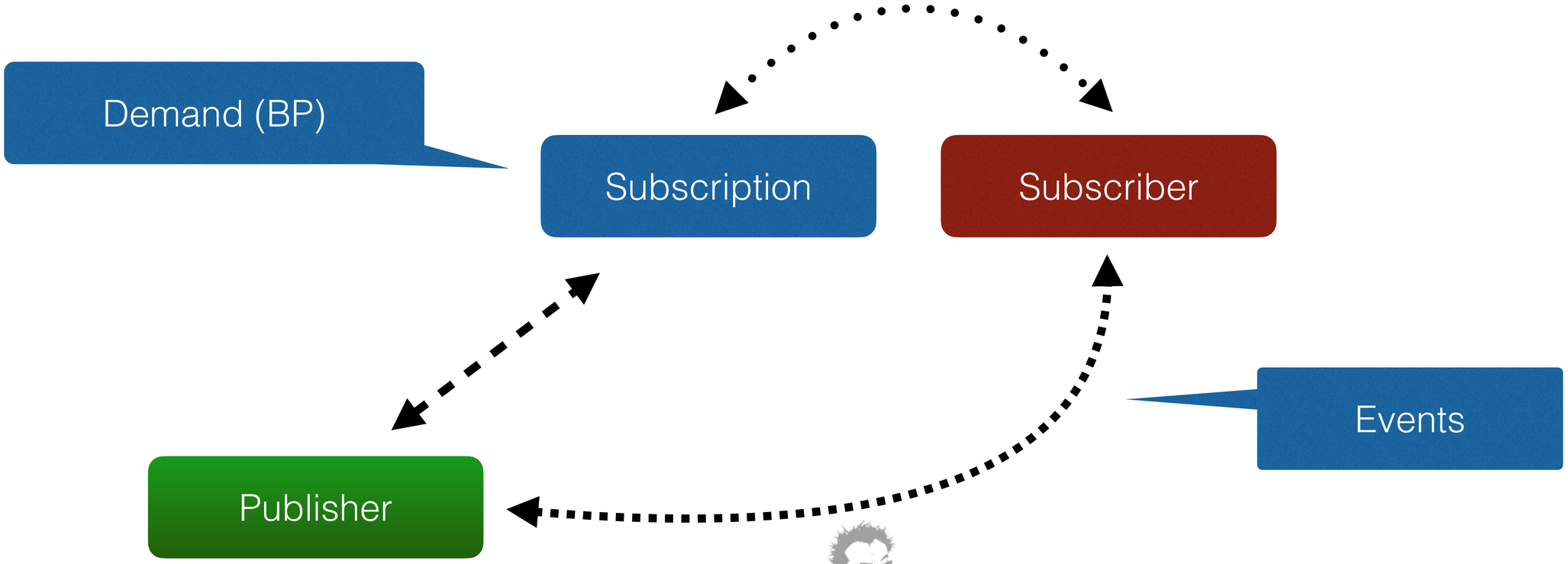
```java
/**
 * A {@link Publisher} is a provider of a potentially unbounded number of sequenced elements, publishing them according to
 * the demand received from its {@link Subscriber}(s).
 * <p>
 * A {@link Publisher} can serve multiple {@link Subscriber}s subscribed {@link #subscribe(Subscriber)} dynamically
 * at various points in time.
 *
 * @param <T> the type of element signaled.
 */
public interface Publisher<T> {

    /**
     * Request {@link Publisher} to start streaming data.
     * <p>
     * This is a "factory method" and can be called multiple times, each time starting a new {@link Subscription}.
     * <p>
     * Each {@link Subscription} will work for only a single {@link Subscriber}.
     * <p>
     * A {@link Subscriber} should only subscribe once to a single {@link Publisher}.
     * <p>
     * If the {@link Publisher} rejects the subscription attempt or otherwise fails it will
     * signal the error via {@link Subscriber#onError}.
     *
     * @param s the {@link Subscriber} that will consume signals from this {@link Publisher}
     */
    public void subscribe(Subscriber<? super T> s);
}
```

```java
public interface Subscriber<T> {
    /**
     * Invoked after calling {@link Publisher#subscribe(Subscriber)}.
     * <p>
     * No data will start flowing until {@link Subscription#request(long)} is invoked.
     * <p>
     * It is the responsibility of this {@link Subscriber} instance to call {@link Subscription#request(long)} whenever more data is wanted.
     * <p>
     * The {@link Publisher} will send notifications only in response to {@link Subscription#request(long)}.
     *
     * @param s
     *            {@link Subscription} that allows requesting data via {@link Subscription#request(long)}
     */
    public void onSubscribe(Subscription s);

    /**
     * Data notification sent by the {@link Publisher} in response to requests to {@link Subscription#request(long)}.
     *
     * @param t the element signaled
     */
    public void onNext(T t);

    /**
     * Failed terminal state.
     * <p>
     * No further events will be sent even if {@link Subscription#request(long)} is invoked again.
     *
     * @param t the throwable signaled
     */
    public void onError(Throwable t);

    /**
     * Successful terminal state.
     * <p>
     * No further events will be sent even if {@link Subscription#request(long)} is invoked again.
     */
    public void onComplete();
}
```

```java
/**
 * A {@link Subscription} represents a one-to-one lifecycle of a {@link Subscriber} subscribing to a {@link Publisher}.
 * <p>
 * It can only be used once by a single {@link Subscriber}.
 * <p>
 * It is used to both signal desire for data and cancel demand (and allow resource cleanup).
 *
 */
public interface Subscription {
    /**
     * No events will be sent by a {@link Publisher} until demand is signaled via this method.
     * <p>
     * It can be called however often and whenever needed—but the outstanding cumulative demand must never exceed Long.MAX_VALUE.
     * An outstanding cumulative demand of Long.MAX_VALUE may be treated by the {@link Publisher} as "effectively unbounded".
     * <p>
     * Whatever has been requested can be sent by the {@link Publisher} so only signal demand for what can be safely handled.
     * <p>
     * A {@link Publisher} can send less than is requested if the stream ends but
     * then must emit either {@link Subscriber#onError(Throwable)} or {@link Subscriber#onComplete()}.
     *
     * @param n the strictly positive number of elements to requests to the upstream {@link Publisher}
     */
    public void request(long n);

    /**
     * Request the {@link Publisher} to stop sending data and clean up resources.
     * <p>
     * Data may still be sent to meet previously signalled demand after calling cancel as this request is asynchronous.
     */
    public void cancel();
}
```

```java
/**
 * A Processor represents a processing stage—which is both a {@link Subscriber}
 * and a {@link Publisher} and obeys the contracts of both.
 *
 * @param <T> the type of element signaled to the {@link Subscriber}
 * @param <R> the type of element signaled by the {@link Publisher}
 */
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

Reactive Streams with Backpressure



Spring MVC & Spring WebFlux

@Controller / @RequestMapping	Router Functions
spring-webmvc	spring-webflux
Servlet API	HTTP / Reactive Streams
Servlet Container	Tomcat, Jetty, Netty, Undertow

Spring Reactive Types

- Two new reactive types are introduced with Spring Framework 5
  - **Mono** - is a publisher with zero or one elements in data stream.
  - **Flux** - is a publisher with zero or MANY elements in the data stream.
    - Both types implement the Reactive Streams Publisher interface

SPRING FRAMEWORK

GURU