# RFC 216: Push Streams

Draft

55 Pages


## Abstract

10 point Arial Centered.

The OSGi specifications have described how to distribute OSGi services across VM boundaries now for a number of years. However many OSGi services are synchronous in their nature. Many of today's business applications require asynchronous distributed communication mechanisms. While the OSGi Event Admin specification describes an asynchronous eventing model inside the Java VM this does not address event distribution to other Vms. In addition, while the OSGi Asynchronous Services specification defines mechanisms for asynchronously invoking services, it does not address some concerns specific to eventing. This RFP aims to address the issue of Distributed Events in an OSGi context.

# 0 Document Information

## 0.1 License

### DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

## 0.2 Trademarks



## 0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

## 0.4 Table of Contents

## 0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 1.

`Source code is shown in this typeface.`

## 0.6 Revision History

The last named individual in this history is currently responsible for this document.

| Revision | Date | Comments |
|---|---|---|
| Initial | Nov 12 2015 | Initial version of the document Split from RFC 214 Tim Ward (Paremus) |

# 1 Introduction

This RFC began as an RFP nearly two years ago, in an effort to provide a better asynchronous messaging and eventing solution between OSGi framework. The RFP experienced some delays because parts of the problem space related to other OSGi RFCs. The primary blocks were the lack of an “updatatable” remote service, and the lack of native support for asynchronous primitives. The Enterprise R6 release will include both RSA 1.1, the Async Service, and OSGi Promises, meaning that further progress is now possible for Distributed Eventing.

As part of distributed eventing it became necessary to think further about event streams and event processing. Modern event processing frameworks such as Akka and Reactive Streams offer a number of useful features for asynchronous systems. Similarly Java 8's introduction of functional programming techniques via lambda expressions opens up a wide variety of solution spaces not previously available.

# 2 Application Domain

Distributed systems may be built using a number of different *interaction patterns.* Despite vocal proponents for each approach - it is increasingly clear that no one architectural solution is optimal in every context. Rather there is a continuous spectrum of interaction behaviors. If at all possible – these should ideally be supported in a consistent / coherent manner.

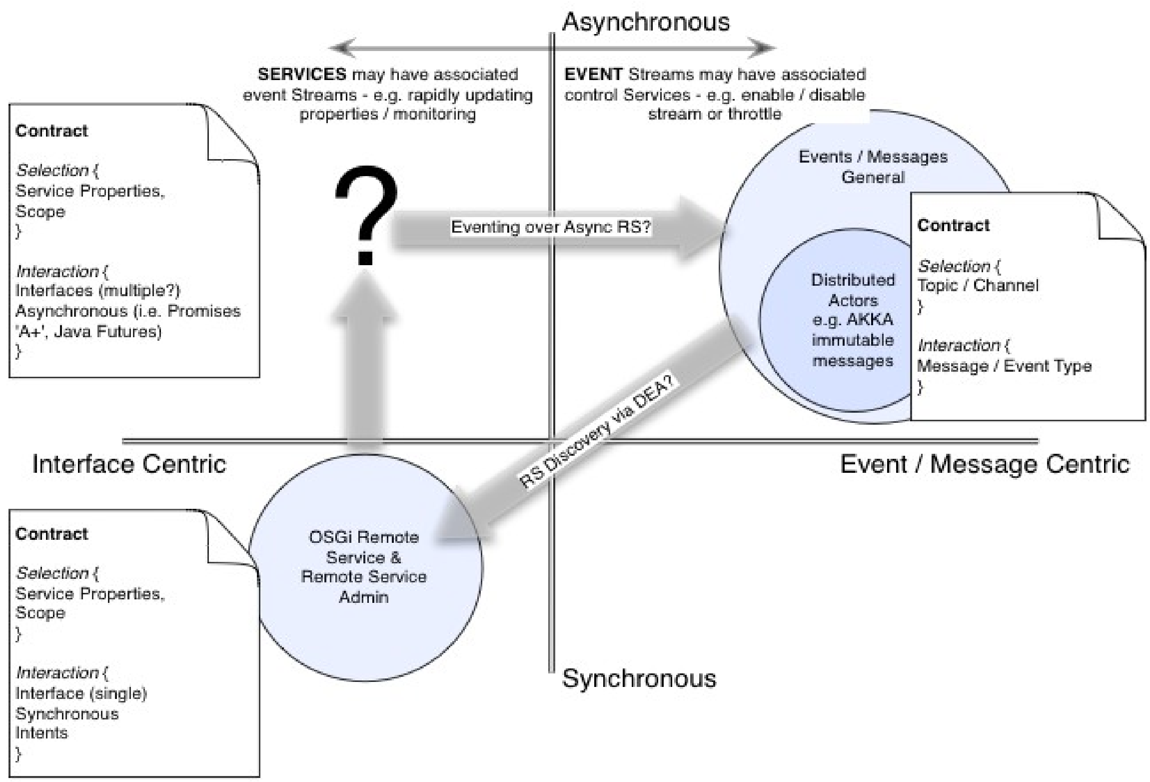



**Figure 1: Types of distributed interaction**

**Synchronous RPC Services:** The OSGi Alliance addressed the requirement for Remote Services via the *remote service* and *remote service admin specifications:* these specifications for synchronous RPC Services. In a dynamic environment (1..n) Services may be dynamically discovered, a sub-set of which (1..m where m<n) may be selected for use based on advertised Service properties. Service properties might be *Immutable* e.g. *Version, Location* or *Ownership* information – or *Mutable*: *reliability metrics*, *rating* or *cost metrics* – which may changing frequently in time.

It should be noted that:

- The RSA architecture is modular – allowing different Data Distribution and Discovery Providers to be used. This approach is extremely flexible. Some RSA implementations may choose to use a distributed P2P event substrate to provide Service discovery while other implementations use some form of static look-up service. Which ever is used - a coherent OSGi architecture is maintained. The use of an distributed Event Substrate for Service Discovery is one example of how RS/RSA and Distributed Eventing might interact.

- The current RSA specification does not address Service property updates: properties may change – and one does not necessarily wish to remove and re-add a Service because of this change. In more extreme cases, for volatile Service properties, one may wish to monitor these. Here a reference to the appropriate event streams might be advertised as Service properties. This scenario highlights a second potential relationship between RS/RSA and Distributed Eventing. (Note that it is planned to update the Remote Service Admin specification for Enterprise R6 to support Service property updates, see RFC 203.)

Note the suggested RS/RSA enhancements are out of scope of this RFP – they are mentioned to illustrate the potential relationships between RS/RSA and a Distributed Eventing specification.

**Asynchronous Services:** Synchronous Services will block until a response is received from the local or remote service. However, on occasions it is preferable for the client to return from the call immediately and continue – some form of notification received at a future point in time – i.e. a 'promise' or a 'future'.

While a number of remote service data distribution providers (RSA) can – in principle support - asynchronous operation, there are currently no OSGi specifications to address support for asynchronous Services (local or remote). Such a specification is desirable as asynchronous Services are increasing popular in Cloud and Distributed environments – and increasingly the JavaScript development community (e.g. node.js). Work in the planned JavaScript RFC will look at implementations of asynchronous Service Registries.

As indicated in Figure 1 – in static environments - Asynchronous Services might be used as a mechanism to implement distributed events. This makes less sense in dynamic environments as some form of discovery mechanism is required – which is usually event based for scaling. So Distributed Eventing would more likely underpin RS/RSA.

This is an important area that requires OSGi Alliance specification work, work that clear relates to both Distributed Eventing & RS / RSA, but Asynchronous Services are out of scope of this current RFP; they are the topic of RFP 132.

**Distributed Events / Messages:**

Asynchronous Message / Event based approaches are increasingly the underpinnings of large scale distributed environments including 'Cloud'. In these distributed systems *Publishers* endpoints are decoupled from *Subscribers* endpoints; association is achieved via the *Topic* the *Publishers* and *Subscribers* choose to subscribe to a named Topic - and /or – a specific Message type.

Implementations vary considerably – and range from 'classic' enterprise message solutions - (e.g. JMS/AMQP) with centralized message brokers – to peer-to-peer de-centralized P2P solutions – e.g. 0MQ and the Object Management Group's (OMG) Data Distribution Service - see .http://www.omg.org/technology/documents/dds_spec_catalog.html

In principle asynchronous message based system provide the potential for greater scalability. However one cannot naively claim that asynchronous messaging will always scale more effectively than synchronous services: the performance characteristics are implementation dependent. An asynchronous messaging Service implemented via a central broker may introduce significant latency, throughput bottlenecks and points of architectural fragility. Whereas a dynamic Services approach with effective Service endpoint load balancing capabilities – would avoid these issues. However, correctly implemented P2P asynchronous message based systems will out perform both - with lower latency, higher throughput and increased resilience.

Due to the increased level of end-point de-coupling and potentially the use of parallel asynchronous pipelines, interaction contracts within message / event based systems are more challenging. Unlike a Service centric approach - failure of Subscribers is not obvious to Publishers (or visa-versa).

Dependent upon the *Capabilities* of the Distributed Eventing provider – events may / may not be durable – and in-order delivery of message / events may / may not be possible.

- Broker based messaging solutions (i.e. JMS Brokers) typically rely on ACID transactions between publishers and the message broker, then the message broker and subscribers. Such solutions are typically used for chaining interactions between coarse grained services running on large compute servers – i.e. Mainframes / large Unix Systems in traditional Enterprise environments. However centralized brokers / ACID transactions represent bottlenecks and points of fragility: failing to efficiently scale in large distributed highly parallel asynchronous environments.

- Increasingly highly distributed / parallelized environments typical of 'Cloud' are using P2P messaging solutions with compensational / recovery / eventual consistency / based approaches to recovery. In such environments the components with a distributed system need to be idempotent as messages / events / may be re-injected in response to some form of timeout or failure notifications. In such environments aggregation points are still required to coordination at input (fan-out) and output (fan-in) boundaries of the parallel flows.

From a developer perspective '*Actors*' are an increasingly popular asynchronous programming style. While popularised by the the Scala / Akka community – Java Actor frameworks also exist – i.e. the kilim actor framework (http://www.malhar.net/sriram/kilim/) and the NetFlix RXJava project https://github.com/Netflix/RxJava/wiki. In these environments local asynchronous events (locally using a message / mailbox pattern) may be distributed between remote 'Actors' via a plug-able messaging layer; e.g. for Akka 0MQ or via Camel / Camel plug-ins. An OSGi Distributed Eventing specification would provide a natural remoting substrate for '*Actor*' / OSGi based applications.

## 2.1 Point-to-point/Queue semantics with current Event Admin Service

Some projects use the OSGi Service Registry Hooks to add point-to-point and/or queue messaging semantics to existing Event Admin Service implementations. This approach is working well for these projects and does not actually require a change to the Event Admin Service specification as it uses the hooks to only show the listeners that should receive the message to the Event Admin Service. While not distributed across remote frameworks such a design could also be relevant in a distributed context.

## 2.2 Existing approaches to distribute the Event Admin Service

A number of projects have successfully implemented a distribution-enabled Event Admin Service employing the existing OSGi API of the Event Admin Service to send events to remote clients. A master thesis was also written on the topic in 2009 by Marc Schaaf [4].

While this approach is very useful in certain situations, it has limitations which make the current Event Admin Service not generally applicable as a service for distributing events.

## 2.3 Terminology + Abbreviations

**Event**: a notification that a circumstance or situation has occurred or an incident happened. Events are represented as data that can be stored and forwarded using any mechanism and/or technology and often include information about the time of occurrence, what caused the event and what entity created the event.

**Message**: a piece of data conveyed via an asynchronous remote communication mechanism such as a message queue product or other middleware. A message can contain an event, but can also have other information or instructions as its payload.

Common definitions for messaging systems include:

**Queue**: A messaging channel which delivers each message only to one receiver even if multiple receivers are registered, the message will only be delivered to one of them. If a receiver disconnects than all following messages are distributed between the remaining recipients. (It should be configurable that if no recipient is registered when a message is about to be delivered if the message is kept until a receiver is registered or if the message will be lost)

**Topic**: A publish and subscribe channel that delivers a message to all currently subscribed receivers. Therefore one message is broadcasted to multiple recipients. If no subscription of a receiver is available when a message is about to be delivered, the message is discarded. If a messaging client is disconnected for a period of time, it will miss all messages transferred during this period.

# 3 Problem Description

The OSGi Alliance has an elegant approach to event management via the Event Admin service. This model, however has a number of drawbacks

## 3.1 Issues with current Event Admin

It should be noted that the following issues exist with the current Event Admin specification. There is no concept of 'contract' and the messages are untyped, so each participant has to continually work out what kind of message it has received, validate it, handle errors and missing info, work out what it should send in response.

- Current Event Admin only specifies how to  send and receive events
- What to do after receiving an Event is unspecified...
- Current Event Admin events are maps, where the values can be anything - Java's instanceof operator to find out the type. Does this / should this / be modernized to be DTO centric?

This is fine if we don't want to go to the trouble of defining a contract for a particular interaction, but the risk is that modules become *more* tightly coupled because of hidden assumptions about the form of events they exchange. Also Event Admin is missing features such as the ability to send a point-to-point reply to a specific message, perhaps to a specific endpoint or subset of endpoints (perhaps via correlation IDs ).

*For these reasons it may not be possible to repurpose the existing Event Admin since it is already designed for a certain set of local use-cases, and there may be backwards compatibility concerns. Hence a completely new distributed eventing design may be required that might optionally replace or complement the local Event Admin service.*

## 3.2 Event Pipelines

The ReactiveX effort provides an API for event stream processing, where “Observers” have events pushed to them, and may publish the event on, or publish another related event as a result. In general this programming model leads to the creation of event “pipelines” through which events flow and are transformed.

This model is effectively a “push based” version of the Java 8 Streams API, which provides a functional pipleline for operating on Collections. The Streams API is inherently “pull based” as it relies on iterators/spliterators to “pull” the next entry from the stream. This is the primary difference between synchronous and asynchronous models. In an asynchronous world entries are pushed into the pipeline.

The other key difference between a pull-based and push based architecture is that pull-based models inherently throttle themselves. A slow part of the pipeline consumes the thread of execution, preventing more events from being created and overloading the system. In a push-based model the non-blocking nature forces “extra” events to be queued. Fast producers can easily overwhelm slow consumers. To combat this asynchronous systems introduce “back-pressure”. Back-pressure is used to indicate that an event source should slow down its event production to avoid overwhelming the consumer.

### 3.2.1 Buffering and Circuit breakers

An important part of stream processing is the use of buffering. Importantly, buffers provide an opportunity for thread switching  in the asynchronous pipeline. This allows event producing threads to be returned to the event source without forcing them to execute the entire pipeline.

Buffers also provide an opportunity to create “circuit breakers”. Event storms occur when a large number of events occur in a short time, and can overwhelm the system. Buffering policies can move the system into a “blocking” state, or can simply disconnect the listener by “breaking” the pipeline. This is known as circuit breaker behaviour.

# 4 Requirements

DE010 – The solution MUST allow the sending of asynchronous messages to remote recipients.

DE012 – The solution MUST support a one-to-many, pub-sub/topic messaging semantic.

DE015 – The solution MUST support a one-to-one, queue messaging semantic.

DE020 – The solution MUST be independent of messaging technology used. This may be message broker based, peer-to-peer using a centralized approach or otherwise.

DE030 – The solution MUST allow implementations to advertise their supported Qualities of Service.

DE040 – The solution MUST provide a mechanism to select an Event Service provider based on its provided QoS.

DE042 – The solution SHOULD define a list of well-known QoS. Implementations MUST NOT be required to support all of these well known QoS.

DE045 – An implementation MUST be allowed to provide additional proprietary Qualities of Service.

DE047 – The solution MUST enable the message sender to specify the actual QoS used for sending a certain message.

DE048 – The solution MUST provide a facility for failure detection and/or reporting in cases where the requested Quality of Service cannot be satisfied.

DE050 – Events / Messages MUST be language agnostic – enabling a remote non-Java party to participate; e.g. C/C++ OSGi based agents.

DE055 – The solution MAY define a standard message encoding, for example using XML, JSON and/or other technology if appropriate.

DE060 – The solution MUST provide the means for point-to-point based communications for example to allow replies to specific messages – an event targeted to a specific node.

DE080 – The solution MUST provide the means to obtain information on the sender of an event e.g. bundleID, Framework UUID, SubSystem name. This information MAY be incomplete if the message didn't originate in an OSGi framework.

DE085 – The solution SHOULD provide the means to discover available Topics and Queues..

DE087 – The solution MUST allow certain Topics and Queues to not be advertised in the discovery mechanism.

DE088 – The solution SHOULD allow certain messages to be hidden from potential receivers.

DE090 – The solution SHOULD NOT prevent an implementation from providing a basic distribution solution for the existing Event Admin. While this will not provide all features of a Distributed Eventing solution, it is shown to be useful in certain contexts.

# 5 Technical Solution

A key part of the event processing model is defining how the events are processed. As OSGi R7 is moving to Java 8 the API design should enable idiomatic use of lambda expressions and functional programming techniques.

## 5.1 Asynchronous Event Processing

The three primitives of event processing are the event producer, the event consumer, and the event type itself.

### 5.1.1 The Event

Asynchronous Event processing is simplest when the Java Type model can be leveraged. The org.osgi.service.event.Event class provides a flexible model of an event, however it also requires the use of "magic string" keys and that values be cast to implementation types. The event type used in distributed eventing should therefore be parameterizable with the type of the payload that it contains.

Whilst some events are "one off" occurrences, the majority of events form part of an event stream. The time between events may be large, or unpredictable, however event topics exist because it is very unusual for streams to consist of a single event. In general event streams consist of a series of data events, followed by zero or one terminal event. The terminal event may be a clean "close" or it may indicate a failure of some kind. In the case of an infinite event stream there may never be a terminal event (for example a series of temperature readings from a sensor has no logical "end").

To encapsulate the above this RFC proposes a PushEvent API type, with an EventType enum to indicate the type of the event. The event has a `getType()` method to access the type information, `getData()` and `getFailure()` methods to access the data or failure that triggered the event, and an `isTerminal()` method to indicate that the method represents the end of the stream. There are also static methods for creating data, error or close events.

In order to facilitate the easy remoting of events the `PushEvent` type is `final` and contains only easy to serialize data. Assuming that the event payload is also serializable the complete state of the event can be reconstituted from its accessor methods.

Whilst it is not enforced, it is strongly recommended that event data be immutable. Mutable state in an event may not be visible between threads, and may cause corruption when processing.

### 5.1.2 Consuming Asynchronous Events

Consuming an event should be a simple process for simple use cases, but needs to be sufficiently flexible to allow more complex systems to be built. A functional interface allows for simple inline implementations to be created. Therefore the event consumer should be a SAM type.

Providing back-pressure to the event producer also requires that the event consumer return a value from the event consuming method. As event processing systems may have significant performance and latency challenges it would be ideal if the type were a primitive or an enum instance as these types avoid performance and garbage collection overheads. This RFC proposes that the return value from the consumer method should therefore be a `long` which indicates either that:

- The stream should be closed (a value < 0)
- Event delivery should continue (a value == 0)
- That delivery of the next event should be delayed by x nanoseconds (a value of x where x > 0)

The proposed API is therefore:

```
@ConsumerType
@FunctionalInterface
public interface PushEventConsumer<T> {

        long accept(PushEvent<? extends T> event) throws Exception;

}
```

This API is simple to provide using a lambda function. Typical PushEventConsumer implementations will receive a number of Data events, followed by a terminal event to represent the end of the stream. In a multithreaded environment it is is possible that some data events may arrive concurrently or out of order. After a terminal event, however, no more events will be delivered.

Note that the back-pressure returned to the source has only a best-effort guarantee. The source is permitted to call the consumer again without waiting for all, or even any, of the requested time to pass.

### 5.1.3 Producing Asynchronous Events

Producing a stream of asynchronous events is reasonably easy. The producer starts a listener, or a background thread, and generates an event as a reaction to some external change (which may simply be a progression in time). The event producer then delivers the event to one or more consumers. As with the `PushEventConsumer` it is preferable to make this a SAM type.

In addition to registering consumers it must also be possible to close the stream of events before it finshes on its own (for example a temperature monitor may be disabled while the stream of temperature events never terminates). Therefore the registration method should return a `Closeable`.

The proposed API is therefore:

```
@ConsumerType
@FunctionalInterface
public interface PushEventSource<T> {

        Closeable open(PushEventConsumer<? super T> event) throws
Exception;

}
```

This API is simple to provide using a lambda function. Typical PushEventSource implementations will produce a number of Data events, followed by a terminal event to represent the end of the stream. The event source should also keep track of any back-pressure requested by the sources, and delay event delivery as requested. This may involve buffering the responses, or simply skipping one or more events. In the case where it is not possible to locally buffer or skip events then the source may continue to deliver the events by ignoring the back-pressure. The source is also required to send a terminal event when a consumer requests it with a negative back-pressure. No further events may be sent after a terminal event.

## 5.2 Asychronous Event Pipelines

Directly connecting a PushEventConsumer and a PushEventSource is a simple way to consume events, however it leaves all of the processing up to the event consumer. Much of this processing is complex, and could be dramatically simplified by including some basic functional concepts from the Java 8 Streams API.

A PushStream is effectively a push-based version of the Java 8 Stream, except where the Stream returns a value, the AsyncStream returns a promise to the value.

### 5.2.1 Simple Stream operations

Operations on a PushStream, much like operations on a Java 8 Stream, are either “intermediate” or “terminal”. Intermediate operations are ones that act upon events in the stream, but return a stream for continued processing. Intermediate operations are also lazy, in particular they do not cause the PushStream to become connected to the PushEventSource. Terminal operations, however, are ones that return a result (wrapped in a Promise) or void.

Intermediate operations may be either Stateless or Stateful. Stateful operations may require a degree of buffering within the stream, and can cause either significant latency or require large amounts of memory. Stateful operations may also be impossible on streams which do not terminate.

Terminal operations trigger the connection of the PushStream to the PushEventSource, causing event delivery to begin. Some terminal operations are short circuiting. Short circuiting operations are ones that do not need to process the entire stream to reach a result, for example findAny(). Other operations, such as min and max, must process every entry to reach a result, and so will not complete until they receive a close event from the event source (this may occur as a result of the stream being closed).

| Stateless Intermediate operations | Stateful Intermediate Operations | Terminal Operations | Short circuit terminal operations |
|---|---|---|---|
| filter() | distinct() | forEach() | anyMatch() |
| map() | sorted() | forEachEvent() | allMatch() |
| flatMap() | limit(long) | toArray() | noneMatch() |
| split() | skip(long) | reduce() | findFirst() |
| sequential() | buffer() | collect() | findAny() |
| fork(); | coalesce() | min() | |
| merge() | window() | max() | |
| | | count() | |

Simple Stream examples 1 – How many odd numbers are there in the next 20 events?:

```
PushStream<Integer> ps = getStream();
Promise<Long> = ps.limit(20)
          .filter(x -> (x & 1) == 0)
          .count();
```

Simple Stream examples 2 – What is the biggest integer less than 100 in the next 1000 events from two different streams?:

```
PushStream<Integer> ps = getStream();
PushStream<Long> ps2 = getOtherStream();

Promise<Optional<Integer>> = ps.merge(
                    ps2.filter(l -> l < Integer.MAX_VALUE)
                              .map(Long::intValue))
          .limit(1000)
          .filter(x -> x < 1000)
          .max();
```

### 5.2.2 The PushStream lifecycle

PushStream instances are created in a disconnected state, and only connect to the underlying event source when a terminal operation is invoked.

PushStream objects implement Closeable, and can therefore be closed at any point. When a stream is closed a close event will be sent to the downstream consumer (if it exists), and any events subsequently received by the stream will return negative back-pressure, so as to close the stream back to its source.

If at any point the stream encounters an error then an error event will be sent to the downstream consumer, and any events subsequently received by the stream will return negative back-pressure, so as to close the stream back to its source.

Finally, if a PushEventConsumer consuming from an PushStream returns negative back-pressure then the stream pipeline will be closed and the negative value returned to the parent. Any events subsequently received by the stream will not be forwarded to the consumer and the stream will return negative back-pressure, so as to close the stream back to its source.

### 5.2.3 PushStream lifecycle callbacks

Clients may register a Runnable callback with the PushStream using the `onClose(Runnable)` method. This callback will run when the stream is closed, regardless of how it is closed (e.g. a call to the close method, or a negative return from a downstream handler). If the PushStream is already closed when the handler is registered then the callback will run immediately. The thread used to run the callback is undefined. Only one close callback may be registered with a particular stage of the pipeline.

Clients may also register a Callback to be notified if the stream completes with an error using the `onError(Consumer<? super Throwable>)`. The PushStream does not hold on to events once it is closed, so this callback will only be called if it is registered before an error occurs. Only one error callback may be registered with a particular stage of the pipeline.

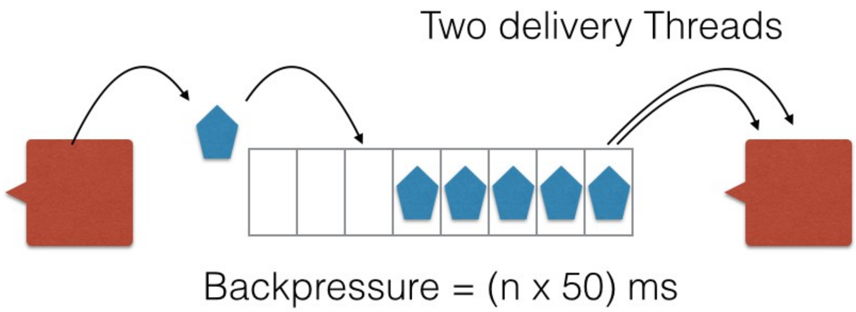
### 5.2.4 Buffering

Buffering is an important part of asynchronous stream processing. Introducing buffers allows processing to be moved onto a different thread, and for the number of processing threads to be changed. Buffering can therefore protect an PushEventSource from having its event generation thread “stolen” by a consumer which executes a long running operation. As a result the PushEventSource can be written more simply.

Buffering also provides a “fire break” for back-pressure. Back-pressure return values propagate back along a PushStream until they reach a part of the stream that is able to respond. For some PushEventSource implementations it is not possible to slow or delay event generation, however a buffer can always respond to back pressure by not releasing events from the buffer. Buffers can therefore be used to “smooth out” sources that produce bursts of events more quickly than they can be immediately processed. This simplifies the creation of PushEventConsumer instances, which can rely on their back-pressure requests being honoured.

Simple buffering is provided by the PushStream using default configuration values (provided by its creator) however more fine-grained control of the buffering can be achieved by supplying the details of the buffer, including:

- The level of parallelism that the downstream side of the buffer should use for event delivery
- The Executor that the buffer should use to deliver downstream events
- The BlockingQueue implementation that should be used to queue the events
- A QueuePolicy, which is responsible for adding events into the queue
- A PushbackPolicy, which determines how much back-pressure should be applied by the buffer




### 5.2.5 Coalescing and windowing

Coalescing and windowing are types of operation that do not exist on pull-based streams. In effect both coalescing and windowing are partial reduction operations. They both consume a number of events and reduce them into a single event which is then forwarded on. In this way they also behave like a buffer, storing up events and only passing them on when the necessary criteria are met.

The primary difference between coalescing and windowing is the way in which the next stage of processing is triggered. A coalescing stage collects events until it has the correct number and then passes them to the handler function, regardless of how long this takes. A windowing stage collects events for a given amount of time, and then passes the collected events to the handler function, regardless of how many events are collected.

To avoid the need for a potentially infinite buffer a windowing stage may also place a limit on the number of events to be buffered. If this limit is reached then the window finishes early and the buffer is passed to the client, just like a coalescing stage. In this mode of operation the handler function is also passed the length of time for which the window lasted.

When coalescing events there is no opportunity for feedback from the event handler while the events are being buffered. As a result backpressure from the handler is zero except when the event triggers a call to the next stage. When the next stage is triggered the back-pressure from that stage is returned.

As windowing requires the collected events to be delivered asynchronously there is no opportunity for back-pressure from the previous stage to be applied upstream. Windowing therefore returns zero back-pressure in all cases except when a buffer size limit has been declared and is reached. If a window size limit is reached then the windowing stage returns the remaining window time as back pressure. Applying back pressure in this way means that the event source will tend not to repeatedly oversaturate the window.

When a coalescing or windowing stream is closed, or receives a close event, then any events in the buffer are immediately passed to the downstream handler, followed by a close event.

**Coalescing Example:**

In the following example the event stream is a sequence of integers running from zero to 49 inclusive. The stream collapses each set of three values into their sum, and then forwards on the result

```
PushStream<Integer> ps = getStream();
List<Integer> list = ps
          .coalesce(3, (e) -> e.stream().reduce(0, (a,b) -> a + b))
          .collect(toList())
```

```
        .getValue();

        // The resulting list is:
        // [3,12,21,30,39,48,57,66,75,84,93,102,111,120,129,138,97]
```

**Windowing Example:**

In the following example the event stream is a sequence of 50 events emitted on average once every 20 milliseconds. The stream windows for 200 milliseconds, counts the events received and then forwards this on as the result

```
PushStream<Integer> ps = getStream();
List<Integer> list = ps.window(200, MILLISECONDS, Collection::size)
                               .collect(toList())
                               .getValue();

        // The resulting list is:
        // [9,12,9,10,9,1]
```

### 5.2.6 Forking

Sometimes the processing that needs to be performed on an event is long-running. An important part of the asynchronous eventing model is that callbacks are short and non-blocking, which means that these callbacks should not run using the primary event thread. One solution to this is to buffer the stream, allowing a thread handoff at the buffer and limiting the impact of the long-running task. Buffering, however, has other consequences, and so it may be the case that a simple thread hand-off is preferable.

Forking allows users to specify a maximum number of concurrent downstream operations. Incoming events will block if this limit has been reached. If there are blocked threads then the returned back pressure for an event will be equal to the number of queued threads multiplied by the supplied timeout value. If there are no blocked threads then the backpressure will be zero.

### 5.2.7 Splitting

Sometimes it is desirable to split a stream into multiple parallel pipelines. These pipelines are independent from the point at which they are split, but share the same source and upstream pipeline.

Splitting a stream is possible using the `split(Predicate<? super T>... predicates)` method. For each predicate an AsyncStream will be returned that receives the events accepted by the predicate.

Note that the lifecycle of a split stream differs from that of a normal stream in two key ways,

1. The stream will begin event delivery when **any** of the downstream handlers encounters a terminal operation

2. The stream will only close when **all** of the downstream handlers are closed.

## 5.3 The PushStreamProvider

The `PushStreamProvider` is a service that can be used to assist with a variety of asynchronous event handling use cases. An `PushStreamProvider` can create `PushStream` instances from an event source, it can buffer an `PushEventConsumer`, or it can turn a `PushStream` into a reusable PushEvent`Source`.

### 5.3.1 Buffered Streams, Unbuffered Streams and Buffered Consumers

The primary use for the `PushStreamProvider` is to create Buffered `PushStream` instances. The two `createStream()` methods mirror the PushStream's `buffer()` method. The default buffering configuration is determined by the implementation of the AsyncStreamProvider, but this can be overridden by explicitly providing the buffering parameters. If the default configuration is used then the PushStream will run using the PushStreamProvider's internal threadpool.

If no buffering is required, then a raw unbuffered PushStream can be created. This uses the incoming event delivery thread to process the events, and therefore users must be careful not to block the thread, or perform long-running tasks.

Both buffered and unbuffered streams are created in a disconnected state, and that the PushEventSource will not be opened until a terminal operation is encountered.

In addtion to buffering streams the PushStreamProvider is also able to buffer PushEventConsumers directly. This wraps the the consumer with a buffer so that it can be implemented simply, but still gain the advantage of additional parallelism and back-pressure.

### 5.3.2 QueuePolicy

A queue policy is used to determine what should happen when an event is added to the queue. The queue implementation must be a BlockingQueue, however subtypes may be used (such as a BlockingDeque) to enable more advanced custom Queueing policies.

There are three basic QueuePolicyOption values that can be used by clients.

- DISCARD_OLDEST – Attempts to add the event to the queue, and discards the event at the head of the queue if the event cannot immediately be added. This process is repeated until the event is queued
- BLOCK – Attempts to add the event to the queue, blocking until the event can be added
- FAIL – Attempts to add the event to the queue, throwing an Exception if the event cannot be immediately added.

### 5.3.3 PushbackPolicy

A PushbackPolicy is used to determine the amount of back-pressure that should be provided by the buffer. As with a QueuePolicy custom PushbackPolicy implmentations that depend on specific queue implementations may be used. A number of simple PushbackPolicyOption types exist, and can be used to create a PushbackPolicy based on a base time in milliseconds:

- FIXED – Returns a fixed value for every event
- ON_FULL_FIXED – Returns zero until the buffer is full, at which point a fixed value is returned

- ON_FULL_EXPONENTIAL – Returns zero until the buffer is full, at which point an exponentially increasing value is returned. Once the buffer is no longer full the back-pressure value is reset
- ON_FULL_CLOSE – Returns zero until the buffer is full, at which point a negative value is used to close the stream
- LINEAR – Returns a value that linearly increases from zero to a fixed value depending on the remaining capacity of the buffer

### 5.3.4 Streams as Event Sources

The final feature of the `PushStreamProvider` is that it enables `PushStream` implementations to be used as `PushEventSources`. It is simple to connect a single `PushEventConsumer` to an `AsyncStream`, however connecting multiple consumers over time is more difficult.

Converting a stream to an EventSource buffers the events before distributing them to any connected consumers.

## 5.4 Simplifying AsyncEventSource creation

The `PushEventSource` and `PushEventConsumer` are both functional interfaces, however it is noticeably harder to implement a `PushEventSource` than a `PushEventConsumer`. A Push`EventSource` must be able to support multiple independently closeable consumer registrations, all of which are providing potentially different amounts of backpressure.

To simplify the case where a user wishes to write a basic event source the `AsyncStreamProvider` is able to create a `SimplePushEventSource`. The `SimplePushEventSource` handles the details of implementing PushEventSource, providing a simplified API for the event producing code to use.

Events can be sent via the `SimplePushEventSource publish(T t)` method at any time until it is closed. These events may be silently ignored if no consumer is connected, but if one or more consumers are connected then the event will be asynchronously delivered to them, When the event has been delivered to all of the connected consumers then the returned promise will resolve.

Close or error events can be sent equally easily using the `endOfStream()` and `error(Exception e)` methods. These will send disconnection events to all of the currently connected consumers and remove them from the SimplePushEventSource. Note that sending these events does not close the SimpleAsyncEventSource, subsequent connection attempts will succeed, and events can still be published.

In addition to the publication methods the `SimplePushEventSource` provides an `isConnected()` method. This method gives a point-in-time snapshot of whether there are any connections to the `SimpleAsyncEventSource`. If this method returns false then the event producer may wish to avoid creating the event, particularly if it is computationally expensive to do so

# 6 Data Transfer Objects

*RFC 185 defines Data Transfer Objects as a generic means for management solutions to interact with runtime entities in an OSGi Framework. DTOs provides a common, easily serializable representation of the technology.*

*For all new functionality added to the OSGi Framework the question should be asked: would this feature benefit from a DTO? The expectation is that in most cases it would.*

*The DTOs for the design in this RFC should be described here and if there are no DTOs being defined an explanation should be given explaining why this is not applicable in this case.*

*This section is optional and could also be provided in a separate RFC.*

# 7 Javadoc

## 7.1 The Async Stream API

# OSGi Javadoc

11/12/15 10:27 AM

# Package org.osgi.util.pushstream

`@org.osgi.annotation.versioning.Version(value="1.0.0")`

| Interface Summary | | Page |
|---|---|---|
| ***PushbackPolicy*** | | 23 |
| ***PushEventConsumer*** | An Async Event Consumer asynchronously receives Data events until it receives either a Close or Error event. | 31 |
| ***PushEventSource*** | An event source. | 33 |
| ***PushStream*** | An Async Stream fullfils the same role as the Java 8 stream but it reverses the control direction. | 34 |
| ***PushStreamProvider*** | | 46 |
| ***QueuePolicy*** | | 49 |
| ***SimplePushEventSource*** | | 52 |

| Class Summary | | Page |
|---|---|---|
| **PushEvent** | An AsyncEvent is an immutable object that is transferred through a communication channel to push information to a downstream consumer. | 26 |

| Enum Summary | | Page |
|---|---|---|
| **PushbackPolicyOption** | | 24 |
| **PushEvent.EventType** | | 29 |
| **QueuePolicyOption** | | 50 |

# Interface PushbackPolicy

**org.osgi.util.pushstream**

---

```
@org.osgi.annotation.versioning.ConsumerType
@FunctionalInterface
public interface PushbackPolicy
```

---

| Method Summary | | Page |
|---|---|---|
| long | **pushback**(U queue)<br>Given the current state of the queue, determine the level of back pressure that should be applied | 23 |

## Method Detail

### pushback

```
long pushback(U queue)
       throws Exception
```

Given the current state of the queue, determine the level of back pressure that should be applied

**Throws:**

`Exception`

# Enum PushbackPolicyOption

org.osgi.util.pushstream

```
java.lang.Object
  └java.lang.Enum<PushbackPolicyOption>
      └org.osgi.util.pushstream.PushbackPolicyOption
```

All Implemented Interfaces:
Comparable<PushbackPolicyOption>, Serializable

---

```
public enum PushbackPolicyOption
extends Enum<PushbackPolicyOption>
```

---

| Enum Constant Summary | Page |
|---|---|
| **FIXED** | 24 |
| **LINEAR** | 25 |
| **ON_FULL_CLOSE** | 24 |
| **ON_FULL_EXPONENTIAL** | 24 |
| **ON_FULL_FIXED** | 24 |

| Method Summary | | Page |
|---|---|---|
| abstract PushbackPolicy<T,U> | **getPolicy**(long value) | 25 |
| static PushbackPolicyOption | **valueOf**(String name) | 25 |
| static PushbackPolicyOption[] | **values**() | 25 |

## Enum Constant Detail

### FIXED

```
public static final PushbackPolicyOption FIXED
```

---

### ON_FULL_FIXED

```
public static final PushbackPolicyOption ON_FULL_FIXED
```

---

### ON_FULL_EXPONENTIAL

```
public static final PushbackPolicyOption ON_FULL_EXPONENTIAL
```

---

### ON_FULL_CLOSE

```
public static final PushbackPolicyOption ON_FULL_CLOSE
```

---

### LINEAR

```
public static final PushbackPolicyOption LINEAR
```

## Method Detail

### values

```
public static PushbackPolicyOption[] values()
```

---

### valueOf

```
public static PushbackPolicyOption valueOf(String name)
```

---

### getPolicy

```
public abstract PushbackPolicy<T,U> getPolicy(long value)
```

# Class PushEvent

org.osgi.util.pushstream

```
java.lang.Object
  └ org.osgi.util.pushstream.PushEvent
```

**Type Parameters:**
  T - The associated Data type

All Implemented Interfaces:
  Serializable

---

```
final public class PushEvent
extends Object
implements Serializable
```

An AsyncEvent is an immutable object that is transferred through a communication channel to push information to a downstream consumer. The event has three different subtypes:

- Data – Provides access to a typed data element in the stream
- Close – The stream is closed. After receiving this event, no more events will follow and and the consumer must assume the stream is dead.
- Error – The upstream ran into an irrecoverable problem and is sending the reason downstream. No more events will follow after this event

---

| Nested Class Summary | | Page |
|---|---|---|
| static enum | **PushEvent.EventType** | 29 |

| Method Summary | | Page |
|---|---|---|
| static PushEvent<T> | **close**()<br>Create a new close event. | 27 |
| static PushEvent<T> | **data**(T payload)<br>Create a new data event | 27 |
| static PushEvent<T> | **error**(Exception e)<br>Create a new error event | 27 |
| T | **getData**()<br>Return the data for this event or throw an exception | 27 |
| Exception | **getFailure**()<br>Return the error or throw an exception if this is not an error type | 27 |
| PushEvent.EventType | **getType**()<br>Get the type of this event | 26 |
| boolean | **isTerminal**()<br>Answer if no more events will follow after this event. | 27 |
| PushEvent<X> | **nodata**()<br>Convenience to cast a close/error event to another payload type. | 28 |

## Method Detail

### getType

```
public PushEvent.EventType getType()
```

Get the type of this event

## getData

```
public T getData()
          throws IllegalStateException
```

Return the data for this event or throw an exception

**Returns:**
the data payload
**Throws:**
`IllegalStateException`

## getFailure

```
public Exception getFailure()
```

Return the error or throw an exception if this is not an error type

**Returns:**
the exception

## isTerminal

```
public boolean isTerminal()
```

Answer if no more events will follow after this event.

**Returns:**
true if a data event, otherwise false.

## data

```
public static PushEvent<T> data(T payload)
```

Create a new data event

**Parameters:**
`payload` - The payload

## error

```
public static PushEvent<T> error(Exception e)
```

Create a new error event

**Parameters:**
`e` - The error
**Returns:**
a new error event with the given error

## close

```
public static PushEvent<T> close()
```

Create a new close event.

**Returns:**
A close event

---

## nodata

```
public PushEvent<X> nodata()
```

Convenience to cast a close/error event to another payload type. Since the payload type is not needed for these events this is harmless. This therefore allows you to forward the close/error event downstream without creating anew event.

# Enum PushEvent.EventType

org.osgi.util.pushstream

```
java.lang.Object
  └java.lang.Enum<PushEvent.EventType>
      └org.osgi.util.pushstream.PushEvent.EventType
```

All Implemented Interfaces:
Comparable<PushEvent.EventType>, Serializable

Enclosing class:
PushEvent

---

```
public static enum PushEvent.EventType
extends Enum<PushEvent.EventType>
```

---

| Enum Constant Summary | Page |
|---|---|
| **CLOSE** | 29 |
| **DATA** | 29 |
| **ERROR** | 29 |

| Method Summary | | Page |
|---|---|---|
| static PushEvent.EventType | **valueOf**(String name) | 30 |
| static PushEvent.EventType[] | **values**() | 29 |

## Enum Constant Detail

### DATA

`public static final PushEvent.EventType DATA`

---

### ERROR

`public static final PushEvent.EventType ERROR`

---

### CLOSE

`public static final PushEvent.EventType CLOSE`

## Method Detail

### values

`public static PushEvent.EventType[] values()`

---

## valueOf

```
public static PushEvent.EventType valueOf(String name)
```

# Interface PushEventConsumer

org.osgi.util.pushstream

**Type Parameters:**

`T` - The type for the event payload

```
@org.osgi.annotation.versioning.ConsumerType
@FunctionalInterface
public interface PushEventConsumer
```

An Async Event Consumer asynchronously receives Data events until it receives either a Close or Error event.

| Field Summary | | Page |
|---|---|---|
| long | **ABORT**<br>If ABORT is used as return value, the sender should close the channel all the way to the upstream source. | 31 |
| long | **CONTINUE**<br>A 0 indicates that the consumer is willing to receive subsequent events at full speeds. | 31 |

| Method Summary | | Page |
|---|---|---|
| long | **accept**(PushEvent<? extends T> event)<br>Accept an event from a source. | 31 |

## Field Detail

### ABORT

```
public static final long ABORT = -1L
```

If ABORT is used as return value, the sender should close the channel all the way to the upstream source. The ABORT will not guarantee that no more events are delivered since this is impossible in a concurrent environment. The consumer should accept subsequent events and close/clean up when the Close or Error event is received. Though ABORT has the value -1, any value less than 0 will act as an abort.

### CONTINUE

```
public static final long CONTINUE = 0L
```

A 0 indicates that the consumer is willing to receive subsequent events at full speeds. Any value more than 0 will indicate that the consumer is becoming overloaded and wants a delay of the given milliseconds before the next event is sent. This allows the consumer to pushback the event delivery speed.

## Method Detail

### accept

```
long accept(PushEvent<? extends T> event)
      throws Exception
```

Accept an event from a source. Events can be delivered on multiple threads simultaneously. However, Close and Error events are the last events received, no more events must be sent after them.

**Parameters:**

`event` - The event

**Returns:**

less than 0 means abort, 0 means continue, more than 0 means delay ms

**Throws:**

`Exception`

# Interface PushEventSource

**org.osgi.util.pushstream**

**Type Parameters:**
`T` - The payload type

All Known Subinterfaces:
SimplePushEventSource

---

```
@org.osgi.annotation.versioning.ConsumerType
@FunctionalInterface
public interface PushEventSource
```

An event source. An event source can open a channel between a source and a consumer. Once the channel is opened (even before it returns) the source can send events to the consumer. A source should stop sending and automatically close the channel when sending an event returns a negative value, see `PushEventConsumer.ABORT`. Values that are larger than 0 should be treated as a request to delay the next events with those number of milliseconds.

---

| Method Summary | | Page |
|---|---|---|
| `Closeable` | `open(PushEventConsumer<? super T> aec)` Open the asynchronous channel between the source and the consumer. | 33 |

## Method Detail

### open

```
Closeable open(PushEventConsumer<? super T> aec)
        throws Exception
```

Open the asynchronous channel between the source and the consumer. The call returns a Closeable. This closeable can be closed, this should close the channel, including sending a Close event if the channel was not already closed. The closeable must be able to be closed multiple times without sending more than one Close events.

**Parameters:**
`aec` - the consumer (not null)

**Returns:**
a Closeable to

**Throws:**
`Exception`

# Interface PushStream

org.osgi.util.pushstream

**Type Parameters:**

`T` - The Payload type

All Superinterfaces:

AutoCloseable, Closeable

---

```
@org.osgi.annotation.versioning.ProviderType
public interface PushStream
extends Closeable
```

An Async Stream fulfils the same role as the Java 8 stream but it reverses the control direction. The Java 8 stream is pull based and this is push based. An Async Stream makes it possible to build a pipeline of transformations using a builder kind of model. Just like streams, it provides a number of terminating methods that will actually open the channel and perform the processing until the channel is closed (The source sends a Close event). The results of the processing will be send to a Promise, just like any error events. A stream can be used multiple times. The Async Stream represents a pipeline. Upstream is in the direction of the source, downstream is in the direction of the terminating method. Events are send downstream asynchronously with no guarantee for ordering or concurrency. Methods are available to provide serialization of the events and splitting in background threads.

---

| **Method Summary** | | **Page** |
|---|---|---|
| org.osgi.util.promise.Promise<Boolean> | **allMatch**(Predicate<? super T> predicate)<br>Closes the channel and resolve the promise with false when the predicate does not matches a pay load.If the channel is closed before, the promise is resolved with true. | 44 |
| org.osgi.util.promise.Promise<Boolean> | **anyMatch**(Predicate<? super T> predicate)<br>Close the channel and resolve the promise with true when the predicate matches a payload. | 44 |
| PushStream<T> | **buffer**()<br>Buffer the events in a queue using default values for the queue size and other behaviours. | 38 |
| PushStream<T> | **buffer**(int parallelism, Executor executor, U queue, QueuePolicy<T,U> queuePolicy, PushbackPolicy<T,U> pushbackPolicy)<br>Buffer the events in a queue using default values for the queue size and other behaviours. | 39 |
| PushStream<R> | **coalesce**(int count, Function<Collection<T>,R> f)<br>Coalesces a number of events into a new type of event. | 40 |
| PushStream<R> | **coalesce**(Function<? super T,Optional<R>> f)<br>Coalesces a number of events into a new type of event. | 40 |
| PushStream<R> | **coalesce**(IntSupplier count, Function<Collection<T>,R> f)<br>Coalesces a number of events into a new type of event. | 40 |
| org.osgi.util.promise.Promise<R> | **collect**(Collector<? super T,A,R> collector)<br>See Stream. | 43 |
| org.osgi.util.promise.Promise<Long> | **count**()<br>See Stream. | 44 |
| PushStream<T> | **distinct**()<br>Remove any duplicates. | 37 |
| PushStream<T> | **filter**(Predicate<? super T> predicate)<br>Only pass events downstream when the predicate tests true. | 36 |
| org.osgi.util.promise.Promise<Optional<T>> | **findAny**()<br>Close the channel and resolve the promise with the first element. | 45 |
| org.osgi.util.promise.Promise<Optional<T>> | **findFirst**()<br>Close the channel and resolve the promise with the first element. | 45 |

| | | |
|---|---|---|
| PushStream<R> | **flatMap**(Function<? super T,? extends PushStream<? extends R>> mapper)<br>Flat map the payload value (turn one event into 0..n events of potentially another type). | 37 |
| org.osgi.util.promise.Promise<Void> | **forEach**(Consumer<? super T> action)<br>Execute the action for each event received until the channel is closed. | 42 |
| org.osgi.util.promise.Promise<Long> | **forEachEvent**(PushEventConsumer<? super T> action)<br>Pass on each event to another consumer until the stream is closed. | 45 |
| PushStream<T> | **fork**(int n, int delay, Executor e)<br>Execute the downstream events in up to n background threads. | 38 |
| PushStream<T> | **limit**(long maxSize)<br>Automatically close the channel after the maxSize number of elements is received. | 37 |
| PushStream<R> | **map**(Function<? super T,? extends R> mapper)<br>Map a payload value. | 36 |
| org.osgi.util.promise.Promise<Optional<T>> | **max**(Comparator<? super T> comparator)<br>See Stream. | 44 |
| PushStream<? extends T> | **merge**(PushEventSource<? extends T> source)<br>Merge in the events from another source. | 39 |
| org.osgi.util.promise.Promise<Optional<T>> | **min**(Comparator<? super T> comparator)<br>See Stream. | 44 |
| org.osgi.util.promise.Promise<Boolean> | **noneMatch**(Predicate<? super T> predicate)<br>Closes the channel and resolve the promise with false when the predicate matches any pay load.If the channel is closed before, the promise is resolved with true. | 44 |
| PushStream<T> | **onClose**(Runnable closeHandler)<br>Must be run after the channel is closed. | 36 |
| PushStream<T> | **onError**(Consumer<? super Throwable> closeHandler)<br>Must be run after the channel is closed. | 36 |
| org.osgi.util.promise.Promise<Optional<T>> | **reduce**(BinaryOperator<T> accumulator)<br>Standard reduce without identity, so the return is an Optional. | 43 |
| org.osgi.util.promise.Promise<T> | **reduce**(T identity, BinaryOperator<T> accumulator)<br>Standard reduce, see Stream. | 43 |
| org.osgi.util.promise.Promise<U> | **reduce**(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)<br>Standard reduce with identity, accumulator and combiner. | 43 |
| PushStream<T> | **sequential**()<br>Ensure that any events are delivered sequentially. | 40 |
| PushStream<T> | **skip**(long n)<br>Skip a number of events in the channel. | 38 |
| PushStream<T> | **sorted**()<br>Sorted the elements, assuming that T extends Comparable. | 37 |
| PushStream<T> | **sorted**(Comparator<? super T> comparator)<br>Sorted the elements with the given comparator. | 37 |
| PushStream<T>[] | **split**(Predicate<? super T>... predicates)<br>Split the events to different streams based on a predicate. | 39 |
| org.osgi.util.promise.Promise<Object> | **toArray**()<br>Collect the payloads in an Object array after the channel is closed. | 42 |
| org.osgi.util.promise.Promise<A> | **toArray**(IntFunction<A> generator)<br>Collect the payloads in an Object array after the channel is closed. | 42 |

| | | |
|---|---|---|
| PushStream <R> | **window**(LongSupplier time, IntSupplier maxEvents, Executor executor, BiFunction<Long,Collection<T>,R> f) Buffers a number of events over a variable time interval and then forwards the events to an accumulator function. | 42 |
| PushStream <R> | **window**(LongSupplier time, IntSupplier maxEvents, BiFunction<Long,Collection<T>,R> f) Buffers a number of events over a variable time interval and then forwards the events to an accumulator function. | 41 |
| PushStream <R> | **window**(long time, TimeUnit unit, Executor executor, Function<Collection<T>,R> f) Buffers a number of events over a fixed time interval and then forwards the events to an accumulator function. | 41 |
| PushStream <R> | **window**(long time, TimeUnit unit, Function<Collection<T>,R> f) Buffers a number of events over a fixed time interval and then forwards the events to an accumulator function. | 40 |

# Method Detail

## onClose

`PushStream<T> onClose(Runnable closeHandler)`

Must be run after the channel is closed. This handler will run after the downstream methods have processed the close event and before the upstream methods have closed.

**Parameters:**
`closeHandler` - Will be called on close
**Returns:**
This stream

---

## onError

`PushStream<T> onError(Consumer<? super Throwable> closeHandler)`

Must be run after the channel is closed. This handler will run after the downstream methods have processed the close event and before the upstream methods have closed.

**Parameters:**
`closeHandler` - Will be called on close
**Returns:**
This stream

---

## filter

`PushStream<T> filter(Predicate<? super T> predicate)`

Only pass events downstream when the predicate tests true.

**Parameters:**
`predicate` - The predicate that is tested (not null)
**Returns:**
Builder style (can be a new or the same object)

---

## map

`PushStream<R> map(Function<? super T,? extends R> mapper)`

Map a payload value.

**Parameters:**
`mapper` - The map function

**Returns:**
Builder style (can be a new or the same object)

---

## flatMap

`PushStream<R> flatMap(Function<? super T,? extends PushStream<? extends R>> mapper)`

Flat map the payload value (turn one event into 0..n events of potentially another type).

**Parameters:**
`mapper` - The flat map function

**Returns:**
Builder style (can be a new or the same object)

---

## distinct

`PushStream<T> distinct()`

Remove any duplicates. Notice that this can be expensive in a large stream since it must track previous payloads.

**Returns:**
Builder style (can be a new or the same object)

---

## sorted

`PushStream<T> sorted()`

Sorted the elements, assuming that T extends Comparable. This is of course expensive for large or infinite streams since it requires buffering the stream until close.

**Returns:**
Builder style (can be a new or the same object)

---

## sorted

`PushStream<T> sorted(Comparator<? super T> comparator)`

Sorted the elements with the given comparator. This is of course expensive for large or infinite streams since it requires buffering the stream until close.

**Returns:**
Builder style (can be a new or the same object)

---

## limit

`PushStream<T> limit(long maxSize)`

Automatically close the channel after the maxSize number of elements is received.

**Parameters:**
`maxSize` - Maximum number of elements has been received
**Returns:**
Builder style (can be a new or the same object)

---

## skip

```
PushStream<T> skip(long n)
```

Skip a number of events in the channel.

**Parameters:**
`n` - number of elements to skip
**Returns:**
Builder style (can be a new or the same object)

---

## fork

```
PushStream<T> fork(int n,
                   int delay,
                   Executor e)
            throws Exception
```

Execute the downstream events in up to n background threads. If more requests are outstanding apply delay * nr of delayed threads back pressure. A downstream channel that is closed or throws an exception will cause all execution to cease and the stream to close

**Parameters:**
`n` - number of simultaneous background threads to use
`delay` - Nr of ms/thread that is queued back pressure
`e` - an executor to use for the background threads.
**Throws:**
`Exception`

---

## buffer

```
PushStream<T> buffer()
              throws Exception
```

Buffer the events in a queue using default values for the queue size and other behaviours. Buffered work will be processed asynchronously in the rest of the chain. Buffering also blocks the transmission of back pressure to previous elements in the chain, although back pressure is honoured by the buffer.

Buffers are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm downstream event consumers. Buffering will not, however, protect downstream components from a source which produces events faster than they can be consumed. For fast sources `filter(Predicate)` and `coalesce(int, Function)` `fork(int, int, Executor)` are better choices.

**Throws:**
`Exception`

---

### buffer

```
PushStream<T> buffer(int parallelism,
                     Executor executor,
                     U queue,
                     QueuePolicy<T,U> queuePolicy,
                     PushbackPolicy<T,U> pushbackPolicy)
              throws Exception
```

Buffer the events in a queue using default values for the queue size and other behaviours. Buffered work will be processed asynchronously in the rest of the chain. Buffering also blocks the transmission of back pressure to previous elements in the chain, although back pressure is honoured by the buffer.

Buffers are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm downstream event consumers. Buffering will not, however, protect downstream components from a source which produces events faster than they can be consumed. For fast sources `filter(Predicate)` and `coalesce(int, Function)` `fork(int, int, Executor)` are better choices.

Buffers are also useful as "circuit breakers" in the pipeline. If a `PushbackPolicyOption.ON_FULL_CLOSE` or `QueuePolicyOption.FAIL` is used then a full buffer will trigger the stream to close, preventing an event storm from reaching the client.

**Throws:**
`Exception`

---

### merge

```
PushStream<? extends T> merge(PushEventSource<? extends T> source)
                       throws Exception
```

Merge in the events from another source. The resulting channel is not closed until this channel and the channel from the source are closed.

**Parameters:**
`source` - The source to merge in.
**Throws:**
`Exception`

---

### split

```
PushStream<T>[] split(Predicate<? super T>... predicates)
               throws Exception
```

Split the events to different streams based on a predicate. If the predicate is true, the event is dispatched to that channel on the same position. All predicates are tested for every event.

This method differs from other methods of AsyncStream in three significant ways:

- The return value contains multiple streams.
- This stream will only close when all of these child streams have closed.
- Event delivery is made to all open children that accept the event.

**Parameters:**
`predicates` - the predicates to test
**Returns:**
streams that map to the predicates
**Throws:**
`Exception`

---

## sequential

```
PushStream<T> sequential()
                  throws Exception
```

Ensure that any events are delivered sequentially. That is, no overlapping calls downstream. This can be used to turn a forked stream (where for example a heavy conversion is done in multiple threads) back into a sequential stream so a reduce is simple to do.

**Throws:**
`Exception`

---

## coalesce

```
PushStream<R> coalesce(Function<? super T,Optional<R>> f)
                throws Exception
```

Coalesces a number of events into a new type of event. The input events are forwarded to a accumulator function. This function returns an Optional. If the optional is present, it's value is send downstream, otherwise it is ignored.

**Throws:**
`Exception`

---

## coalesce

```
PushStream<R> coalesce(int count,
                       Function<Collection<T>,R> f)
                throws Exception
```

Coalesces a number of events into a new type of event. A fixed number of input events are forwarded to a accumulator function. This function returns new event data to be forwarded on.

**Throws:**
`Exception`

---

## coalesce

```
PushStream<R> coalesce(IntSupplier count,
                       Function<Collection<T>,R> f)
                throws Exception
```

Coalesces a number of events into a new type of event. A variable number of input events are forwarded to a accumulator function. The number of events to be forwarded is determined by calling the count function. The accumulator function then returns new event data to be forwarded on.

**Throws:**
`Exception`

---

## window

```
PushStream<R> window(long time,
                     TimeUnit unit,
                     Function<Collection<T>,R> f)
              throws Exception
```

Buffers a number of events over a fixed time interval and then forwards the events to an accumulator function. This function returns new event data to be forwarded on. Note that:

- The collection forwarded to the accumulator function will be empty if no events arrived during the time interval.
- The accumulator function will be run and the forwarded event delivered as a different task, (and therefore potentially on a different thread) from the one that delivered the event to this `PushStream`.
- Due to the buffering and asynchronous delivery required, this method prevents the propagation of back-pressure to earlier stages

**Throws:**

`Exception`

---

## window

```
PushStream<R> window(long time,
                     TimeUnit unit,
                     Executor executor,
                     Function<Collection<T>,R> f)
              throws Exception
```

Buffers a number of events over a fixed time interval and then forwards the events to an accumulator function. This function returns new event data to be forwarded on. Note that:

- The collection forwarded to the accumulator function will be empty if no events arrived during the time interval.
- The accumulator function will be run and the forwarded event delivered by a task given to the supplied executor.
- Due to the buffering and asynchronous delivery required, this method prevents the propagation of back-pressure to earlier stages

**Throws:**

`Exception`

---

## window

```
PushStream<R> window(LongSupplier time,
                     IntSupplier maxEvents,
                     BiFunction<Long,Collection<T>,R> f)
              throws Exception
```

Buffers a number of events over a variable time interval and then forwards the events to an accumulator function. The length of time over which events are buffered is determined by the time function. A maximum number of events can also be requested, if this number of events is reached then the accumulator will be called early. The accumulator function returns new event data to be forwarded on. It is also given the length of time for which the buffer accumulated data. This may be less than the requested interval if the buffer reached the maximum number of requested events early. Note that:

- The collection forwarded to the accumulator function will be empty if no events arrived during the time interval.
- The accumulator function will be run and the forwarded event delivered as a different task, (and therefore potentially on a different thread) from the one that delivered the event to this `PushStream`.
- Due to the buffering and asynchronous delivery required, this method prevents the propagation of back-pressure to earlier stages
- If the window finishes by hitting the maximum number of events then the remaining time in the window will be applied as back-pressure to the previous stage, attempting to slow the producer to the expected windowing threshold.

**Throws:**

`Exception`

---

## window

```
PushStream<R> window(LongSupplier time,
                     IntSupplier maxEvents,
                     Executor executor,
                     BiFunction<Long,Collection<T>,R> f)
              throws Exception
```

Buffers a number of events over a variable time interval and then forwards the events to an accumulator function. The length of time over which events are buffered is determined by the time function. A maximum number of events can also be requested, if this number of events is reached then the accumulator will be called early. The accumulator function returns new event data to be forwarded on. It is also given the length of time for which the buffer accumulated data. This may be less than the requested interval if the buffer reached the maximum number of requested events early. Note that:

- The collection forwarded to the accumulator function will be empty if no events arrived during the time interval.
- The accumulator function will be run and the forwarded event delivered as a different task, (and therefore potentially on a different thread) from the one that delivered the event to this `PushStream`.
- If the window finishes by hitting the maximum number of events then the remaining time in the window will be applied as back-pressure to the previous stage, attempting to slow the producer to the expected windowing threshold.

**Throws:**
`Exception`

---

## forEach

```
org.osgi.util.promise.Promise<Void> forEach(Consumer<? super T> action)
                                     throws Exception
```

Execute the action for each event received until the channel is closed. This is a terminating method, the returned promise is resolved when the channel closes.

This is a **terminal operation**

**Parameters:**
`action` - The action to perform
**Returns:**
A promise that is resolved when the channel closes.
**Throws:**
`Exception`

---

## toArray

```
org.osgi.util.promise.Promise<Object> toArray()
```

Collect the payloads in an Object array after the channel is closed. This is a terminating method, the returned promise is resolved when the channel is closed.

This is a **terminal operation**

**Returns:**
A promise that is resolved with all the payloads received over the channel

---

## toArray

```
org.osgi.util.promise.Promise<A> toArray(IntFunction<A> generator)
```

Collect the payloads in an Object array after the channel is closed. This is a terminating method, the returned promise is resolved when the channel is closed. The type of the array is handled by the caller using a generator function that gets the length of the desired array.

This is a **terminal operation**

**Returns:**
A promise that is resolved with all the payloads received over the channel

---

## reduce

```
org.osgi.util.promise.Promise<T> reduce(T identity,
                                        BinaryOperator<T> accumulator)
```

Standard reduce, see Stream. The returned promise will be resolved when the channel closes.

This is a **terminal operation**

**Parameters:**
`identity` - The identity/begin value
`accumulator` - The accumulator
**Returns:**
A

---

## reduce

```
org.osgi.util.promise.Promise<Optional<T>> reduce(BinaryOperator<T> accumulator)
```

Standard reduce without identity, so the return is an Optional. The returned promise will be resolved when the channel closes.

This is a **terminal operation**

**Parameters:**
`accumulator` - The accumulator
**Returns:**
an Optional

---

## reduce

```
org.osgi.util.promise.Promise<U> reduce(U identity,
                                        BiFunction<U,? super T,U> accumulator,
                                        BinaryOperator<U> combiner)
```

Standard reduce with identity, accumulator and combiner. The returned promise will be resolved when the channel closes.

This is a **terminal operation**

**Parameters:**
`combiner` - combines to U's into one U (e.g. how combine two lists)
**Returns:**
The promise

---

## collect

```
org.osgi.util.promise.Promise<R> collect(Collector<? super T,A,R> collector)
```

See Stream. Will resolve onces the channel closes.

This is a **terminal operation**

---

## min

`org.osgi.util.promise.Promise<Optional<T>> min(Comparator<? super T> comparator)`

See Stream. Will resolve onces the channel closes.

This is a **terminal operation**

---

## max

`org.osgi.util.promise.Promise<Optional<T>> max(Comparator<? super T> comparator)`

See Stream. Will resolve onces the channel closes.

This is a **terminal operation**

---

## count

`org.osgi.util.promise.Promise<Long> count()`

See Stream. Will resolve onces the channel closes.

This is a **terminal operation**

---

## anyMatch

`org.osgi.util.promise.Promise<Boolean> anyMatch(Predicate<? super T> predicate)`

Close the channel and resolve the promise with true when the predicate matches a payload. If the channel is closed before the predicate matches, the promise is resolved with false.

This is a **terminal operation**

---

## allMatch

`org.osgi.util.promise.Promise<Boolean> allMatch(Predicate<? super T> predicate)`

Closes the channel and resolve the promise with false when the predicate does not matches a pay load.If the channel is closed before, the promise is resolved with true.

This is a **terminal operation**

---

## noneMatch

`org.osgi.util.promise.Promise<Boolean> noneMatch(Predicate<? super T> predicate)`

Closes the channel and resolve the promise with false when the predicate matches any pay load.If the channel is closed before, the promise is resolved with true.

This is a **terminal operation**

---

### findFirst

```
org.osgi.util.promise.Promise<Optional<T>> findFirst()
```

Close the channel and resolve the promise with the first element. If the channel is closed before, the Optional will have no value.

**Returns:**
a promise

---

### findAny

```
org.osgi.util.promise.Promise<Optional<T>> findAny()
```

Close the channel and resolve the promise with the first element. If the channel is closed before, the Optional will have no value.

This is a **terminal operation**

**Returns:**
a promise

---

### forEachEvent

```
org.osgi.util.promise.Promise<Long> forEachEvent(PushEventConsumer<? super T> action)
                                          throws Exception
```

Pass on each event to another consumer until the stream is closed.

This is a **terminal operation**

**Returns:**
a promise

**Throws:**
`Exception`

# Interface PushStreamProvider

org.osgi.util.pushstream

```
@org.osgi.annotation.versioning.ProviderType
public interface PushStreamProvider
```

| Method Summary | | Page |
|---|---|---|
| PushEventSource<T> | **asEventSource**(PushStream<T> stream)<br>Convert an PushStream into an PushEventSource. | 47 |
| PushEventSource<T> | **asEventSource**(PushStream<T> stream, int parallelism, Executor executor, Supplier<U> queueFactory, QueuePolicy<T,U> queuePolicy, PushbackPolicy<T,U> pushbackPolicy)<br>Convert an PushStream into an PushEventSource. | 47 |
| PushEventConsumer<T> | **buffer**(PushEventConsumer<T> delegate)<br>Create a buffered PushEventConsumer with the default configured buffer, executor size, queue, queue policy and pushback policy. | 48 |
| PushEventConsumer<T> | **buffer**(PushEventConsumer<T> delegate, int parallelism, Executor executor, U queue, QueuePolicy<T,U> queuePolicy, PushbackPolicy<T,U> pushbackPolicy)<br>Create a buffered PushEventConsumer with custom configuration. | 48 |
| SimplePushEventSource<T> | **createSimpleEventSource**(Class<T> type)<br>Create a SimplePushEventSource with the supplied type and default buffering behaviours. | 48 |
| SimplePushEventSource<T> | **createSimpleEventSource**(Class<T> type, int parallelism, Executor executor, Supplier<U> queueFactory, QueuePolicy<T,U> queuePolicy, PushbackPolicy<T,U> pushbackPolicy)<br>Create a SimplePushEventSource with the supplied type and custom buffering behaviours. | 48 |
| PushStream<T> | **createStream**(PushEventSource<T> eventSource)<br>Create a stream with the default configured buffer, executor size, queue, queue policy and pushback policy. | 46 |
| PushStream<T> | **createStream**(PushEventSource<T> eventSource, int parallelism, Executor executor, U queue, QueuePolicy<T,U> queuePolicy, PushbackPolicy<T,U> pushbackPolicy)<br>Create a buffered stream with custom configuration. | 47 |
| PushStream<T> | **createUnbufferedStream**(PushEventSource<T> eventSource)<br>Create an unbuffered stream. | 47 |

## Method Detail

### createStream

```
PushStream<T> createStream(PushEventSource<T> eventSource)
```

Create a stream with the default configured buffer, executor size, queue, queue policy and pushback policy.

This stream will be buffered from the event producer, and will honour back pressure even if the source does not.

Buffered streams are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm downstream processors. Buffering will not, however, protect downstream components from a source which produces events faster than they can be consumed.

Event delivery will not begin until a terminal operation is reached on the chain of AsyncStreams. Once a terminal operation is reached the stream will be connected to the event source.

## createStream

```
PushStream<T> createStream(PushEventSource<T> eventSource,
                           int parallelism,
                           Executor executor,
                           U queue,
                           QueuePolicy<T,U> queuePolicy,
                           PushbackPolicy<T,U> pushbackPolicy)
```

Create a buffered stream with custom configuration.

Buffered streams are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm downstream processors. Buffering will not, however, protect downstream components from a source which produces events faster than they can be consumed.

Buffers are also useful as "circuit breakers" in the pipeline. If a `PushbackPolicyOption.ON_FULL_CLOSE` or `QueuePolicyOption.FAIL` is used then a full buffer will trigger the stream to close, preventing an event storm from reaching the client.

This stream will be buffered from the event producer, and will honour back pressure even if the source does not.

---

## createUnbufferedStream

```
PushStream<T> createUnbufferedStream(PushEventSource<T> eventSource)
```

Create an unbuffered stream. This stream will use the producer's thread(s) to process the events and will directly provide back pressure to the source.

**N.B.** If the `PushEventSource` does not respond to the backpressure responses then the stream may become overloaded. Consider using a buffered stream for anything other than trivial event processing.

Event delivery will not begin until a terminal operation is reached on the chain of AsyncStreams. Once a terminal operation is reached the stream will be connected to the event source.

---

## asEventSource

```
PushEventSource<T> asEventSource(PushStream<T> stream)
```

Convert an `PushStream` into an `PushEventSource`. The first call to `PushEventSource.open(PushEventConsumer)` will begin event processing. The `PushEventSource` will remain active until the backing stream is closed, and permits multiple consumers to `PushEventSource.open(PushEventConsumer)` it.

---

## asEventSource

```
PushEventSource<T> asEventSource(PushStream<T> stream,
                                 int parallelism,
                                 Executor executor,
                                 Supplier<U> queueFactory,
                                 QueuePolicy<T,U> queuePolicy,
                                 PushbackPolicy<T,U> pushbackPolicy)
```

Convert an `PushStream` into an `PushEventSource`. The first call to `PushEventSource.open(PushEventConsumer)` will begin event processing. The `PushEventSource` will remain active until the backing stream is closed, and permits multiple consumers to `PushEventSource.open(PushEventConsumer)` it.

---

### createSimpleEventSource

```
SimplePushEventSource<T> createSimpleEventSource(Class<T> type)
```

Create a `SimplePushEventSource` with the supplied type and default buffering behaviours. The SimpleAsyncEventSource will respond to back pressure requests from the consumers connected to it.

---

### createSimpleEventSource

```
SimplePushEventSource<T> createSimpleEventSource(Class<T> type,
                                                 int parallelism,
                                                 Executor executor,
                                                 Supplier<U> queueFactory,
                                                 QueuePolicy<T,U> queuePolicy,
                                                 PushbackPolicy<T,U> pushbackPolicy)
```

Create a `SimplePushEventSource` with the supplied type and custom buffering behaviours. The SimpleAsyncEventSource will respond to back pressure requests from the consumers connected to it.

**Parameters:**
`queueFactory` - A factory used to create a queue for each connected consumer

---

### buffer

```
PushEventConsumer<T> buffer(PushEventConsumer<T> delegate)
```

Create a buffered `PushEventConsumer` with the default configured buffer, executor size, queue, queue policy and pushback policy.

The returned consumer will be buffered from the event source, and will honour back pressure requests from its delegate even if the event source does not.

Buffered consumers are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm the consumer. Buffering will not, however, protect downstream components from a source which produces events faster than they can be consumed.

---

### buffer

```
PushEventConsumer<T> buffer(PushEventConsumer<T> delegate,
                            int parallelism,
                            Executor executor,
                            U queue,
                            QueuePolicy<T,U> queuePolicy,
                            PushbackPolicy<T,U> pushbackPolicy)
```

Create a buffered `PushEventConsumer` with custom configuration.

The returned consumer will be buffered from the event source, and will honour back pressure requests from its delegate even if the event source does not.

Buffered consumers are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm the consumer. Buffering will not, however, protect downstream components from a source which produces events faster than they can be consumed.

Buffers are also useful as "circuit breakers". If a `PushbackPolicyOption.ON_FULL_CLOSE` or `QueuePolicyOption.FAIL` is used then a full buffer will request that the stream close, preventing an event storm from reaching the client.

# Interface QueuePolicy

org.osgi.util.pushstream

```
@org.osgi.annotation.versioning.ConsumerType
@FunctionalInterface
public interface QueuePolicy
```

| Method Summary | | *Page* |
| --- | --- | --- |
| void | **doOffer**(U queue, PushEvent<? extends T> event)<br>Enqueue the event and return the remaining capacity available for events | *49* |

## Method Detail

### doOffer

```
void doOffer(U queue,
             PushEvent<? extends T> event)
      throws Exception
```

Enqueue the event and return the remaining capacity available for events

**Throws:**

Exception

# Enum QueuePolicyOption

org.osgi.util.pushstream

```
java.lang.Object
  └ java.lang.Enum<QueuePolicyOption>
      └ org.osgi.util.pushstream.QueuePolicyOption
```

All Implemented Interfaces:
Comparable<QueuePolicyOption>, Serializable

---

```
public enum QueuePolicyOption
extends Enum<QueuePolicyOption>
```

---

| Enum Constant Summary | Page |
|---|---|
| **BLOCK** | 50 |
| **DISCARD_OLDEST** | 50 |
| **FAIL** | 50 |

| Method Summary | | Page |
|---|---|---|
| abstract QueuePolicy<T,U> | **getPolicy**() | 51 |
| static QueuePolicyOption | **valueOf**(String name) | 51 |
| static QueuePolicyOption[] | **values**() | 50 |

## Enum Constant Detail

### DISCARD_OLDEST

`public static final QueuePolicyOption DISCARD_OLDEST`

---

### BLOCK

`public static final QueuePolicyOption BLOCK`

---

### FAIL

`public static final QueuePolicyOption FAIL`

## Method Detail

### values

`public static QueuePolicyOption[] values()`

---

### valueOf

`public static QueuePolicyOption valueOf(String name)`

---

### getPolicy

`public abstract QueuePolicy<T,U> getPolicy()`

# Interface SimplePushEventSource

**org.osgi.util.pushstream**

All Superinterfaces:

AutoCloseable, Closeable, PushEventSource<T>

---

```
@org.osgi.annotation.versioning.ProviderType
public interface SimplePushEventSource
extends PushEventSource<T>, Closeable
```

---

| Method Summary | | Page |
|---|---|---|
| void | **close**() <br> Close this source. | 52 |
| void | **endOfStream**() <br> Close this source for now, but potentially reopen it later. | 53 |
| void | **error**(Exception e) <br> Close this source for now, but potentially reopen it later. | 53 |
| boolean | **isConnected**() <br> Determine whether there are any `PushEventConsumer`s for this `PushEventSource`. | 53 |
| void | **publish**(T t) <br> Asynchronously publish an event to this stream and all connected `PushEventConsumer` instances. | 52 |

| Methods inherited from interface org.osgi.util.pushstream.PushEventSource |
|---|
| open |

## Method Detail

### close

`void close()`

Close this source. Calling this method indicates that there will never be any more events published by it. Calling this method sends a close event to all connected consumers. After calling this method any `PushEventConsumer` that tries to `PushEventSource.open(PushEventConsumer)` this source will immediately receive a close event.

**Specified by:**

`close` in interface `AutoCloseable`
`close` in interface `Closeable`

---

### publish

`void publish(T t)`

Asynchronously publish an event to this stream and all connected `PushEventConsumer` instances. When this method returns there is no guarantee that all consumers have been notified. Events published by a single thread will maintain their relative ordering, however they may be interleaved with events from other threads.

**Throws:**

`IllegalStateException` - if the source is closed

---

### endOfStream

`void endOfStream()`

Close this source for now, but potentially reopen it later. Calling this method asynchronously sends a close event to all connected consumers. After calling this method any `PushEventConsumer` that wishes may `PushEventSource.open(PushEventConsumer)` this source, and will receive subsequent events.

---

### error

`void error(Exception e)`

Close this source for now, but potentially reopen it later. Calling this method asynchronously sends an error event to all connected consumers. After calling this method any `PushEventConsumer` that wishes may `PushEventSource.open(PushEventConsumer)` this source, and will receive subsequent events.

---

### isConnected

`boolean isConnected()`

Determine whether there are any `PushEventConsumer`s for this `PushEventSource`. This can be used to skip expensive event creation logic when there are no listeners.



# 8 Considered Alternatives

# 9 Security Considerations

*Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.*

# 10 Document Support

## 10.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. The Power of Events'. D. C. Luckham. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[4]. Extending OSGi by Means of Asynchronous Messaging - Master Thesis, Marc Schaaf, September 2009, University of Applied Sciences and Arts Hannover. http://schaaf.es/docs/master_thesis_marc_schaaf_Extending_OSGi_by_Means_of_Asynchronous_Messaging.pdf

## 10.2 Author's Address

| | |
|---|---|
| Name | Tim Ward |
| Company | Paremus |
| Address | |
| Voice | |
| e-mail | tim.ward@paremus.com |

| | |
|---|---|
| Name | Peter Kriens |
| Company | Paremus |
| Address | |
| Voice | |
| e-mail | peter.kriens@aqute.biz |

| | |
|---|---|
| Name | Richard Nicholson |
| Company | Paremus Ltd |
| Address | 107-111 Fleet Street London |
| Voice | |
| e-mail | richard.nicholson@paremus.com |

| | |
|---|---|
| Name | Marc Schaaf |
| Company | |
| Address | |
| Voice | |
| e-mail | marc@marc-schaaf.de |

| | |
|---|---|
| Name | David Bosschaert |
| Company | Adobe |
| Address | |
| Voice | |
| e-mail | bosschae@adobe.com |

| | |
|---|---|
| Name | Carsten Ziegeler |
| Company | Adobe |
| Address | |
| Voice | |
| e-mail | cziegele@adobe.com |

| | |
|---|---|
| Name | Graham Charters |
| Company | IBM |
| Address | |
| Voice | |
| e-mail | charters@uk.ibm.com |

## 10.3 Acronyms and Abbreviations

## 10.4 End of Document