MicroProfile Reactive Streams Operators Specification

James Roper, Clement Escoffier, Gordon Hutchison

1.0, January 17, 2019

Table of Contents

MicroProfile Reactive Streams Operators ... 2
Introduction ... 3
- Rationale ... 3
- Scope ... 4
- Reactive Streams API dependency ... 4

Design ... 5
- Stages ... 5
- Graphs ... 5
- Usage ... 7

Implementation ... 8
- Memory visibility ... 8
- User exceptions ... 8
- Error propagation ... 8
- Cleanup ... 9
- null elements ... 9

CDI Integration ... 10
- Injection of engines ... 10
- Contexts ... 10

Reactive Streams Usage Examples ... 12
- Trivial closed graph ... 12
- Building a publisher ... 12
- Building a subscriber ... 13
- Building a processor ... 13
- Consuming a publisher ... 13
- Feeding a subscriber ... 14

Similarities and differences with the Java Stream API ... 15
- Asynchronous processing ... 15
- No parallel processing ... 15
- Other differences ... 16

SPI ... 17
TCK ... 18
- Structure ... 18
  - API TCK ... 18
    - Running the API TCK ... 18
  - SPI TCK ... 19
    - Running the SPI TCK ... 19
  - Running the TCK in a container ... 19

```
Specification: MicroProfile Reactive Streams Operators Specification

Version: 1.0

Status: Final

Release: January 17, 2019
```


```
Copyright (c) 2018 Contributors to the Eclipse Foundation

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

   http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

MicroProfile Reactive Streams Operators

Introduction

This specification defines an API for manipulating Reactive Streams, providing operators such as `map`, `filter`, `flatMap`, in a similar fashion to the `java.util.stream` API introduced in Java 8. It also provides an SPI for implementing and providing custom Reactive Streams engines, allowing application developers to use whichever engine they see fit.

Rationale

The `java.util.stream` API provides functionality necessary for manipulating streams in memory introducing functional programming into the Java language. However, manipulating potentially infinite asynchronous streams has some very different requirements which the `java.util.stream` API is not suitable for.

Reactive Streams is a specification for asynchronous streaming between different libraries and/or technologies, which is included in JDK9 as the `java.util.concurrent.Flow` spec. Reactive Streams itself however is an SPI for library and technology vendors to implement, it is not intended that application developers implement it as the semantics are very complex.

Commonly, Reactive Streams requires more than just plumbing `publishers` to `subscribers`, the stream typically needs to be manipulated in some way, such as applying operations such as `map`, `filter` and `flatMap`. Neither Reactive Streams, nor the JDK, provides an API for doing these manipulations. Since users are not meant to implement Reactive Streams themselves, this means the only way to do these manipulations currently is to depend on a third party library providing operators, such as Akka Streams, RxJava or Reactor.

This API seeks to fill that gap, so that MicroProfile application developers can manipulate Reactive Streams without bringing in a third party dependency. By itself, this API is not interesting to MicroProfile, but with the addition of other Reactive features, such as the MicroProfile Reactive Messaging proposal, it is essential.

There are a number of different approaches to handling the cross cutting concerns relating to actually running streams. These include how and whether context is propagated, concurrency models, buffering, hooks for monitoring and remoting. Different implementations of Reactive Streams offer different approaches based on varying opinions on how these cross cutting concerns should be treated.

For this reason, this API provides an underlying SPI to allow different engines to be plugged in to actually run the streams. Implementors of this spec will provide a default implementation, however users can select to use a custom implementation if they need. Additional flexibility is provided by using a service loaded implementation of the factory class that creates the stream builders that implement the fluent API, enabling the provided implementation to be replaced.

This specification started as a proposal for the JDK, but after discussions there, it was decided that the JDK was not the right place to incubate such an API. It is the intention that once incubated in MicroProfile, this specification will be put to the JDK again as a proposal.

Scope

This specification aims to define a set of operators to manipulate Reactive Streams. The proposed API is voluntarily close to the `java.util.stream` API.

Reactive Streams API dependency

Reactive Streams has been included in the JDK9 as the `java.util.concurrent.Flow` API, which contains the `Publisher`, `Subscriber`, `Subscription` and `Processor` interfaces as nested interfaces of `Flow`. MicroProfile however is not ready to move to a baseline requirement for JDK9 or above.

For this reason, this proposal uses the JDK6 compatible `org.reactivestreams` API, which provides identical `Publisher`, `Subscriber`, `Subscription` and `Processor` interfaces as members of the `org.reactivestreams` package. This dependency contains nothing but those interfaces.

It has been discussed that MicroProfile could copy those interfaces itself, so as to not add this dependency, however this would most likely frustrate use of Reactive Streams in MicroProfile. There is a large ecosystem built around the `org.reactivestreams` interfaces. If application developers wanted to leverage that ecosystem in their application, they would have to write adapters to bridge the two APIs. Given that the `org.reactivestreams` dependency is a jar that contains only these interfaces, identical to the interfaces in JDK9, there doesn't seem to be any value in copying them again into MicroProfile.

To facilitate an eventual migration to the JDK9 interfaces, once MicroProfile adopts JDK9 or later as a baseline JDK version, all methods that pass `org.reactivestreams` interfaces to the user (either as a return value, or by virtue of a user providing a callback to the method to receive it) will have `Rs` added to their name. For example, `getSubscriber` will be called `getRsSubscriber`. This will allow new methods to be added in future that return `java.util.concurrent.Flow` interfaces, without the `Rs` in the name, allowing the existing `Rs` methods to be kept for a limited time for backwards compatibility. Methods that accept a `org.reactivestreams` interface do not need to be given the same treatment, as support for the JDK9 interfaces can be added by overloading them, with backwards compatibility being maintained (see reactive approach for MicroProfile).

Where reactive streams objects are imported, core requirements are specified by the combination of the Reactive Streams specification and this specification but there may be other requirements in the context of a specific implementation. Similarly, reactive streams objects that are generated by APIs in this specification will behave according to the Reactive Streams specification but aspects of their behaviour beyond that, such as handling contexts, buffering policy, fan-out and so on are implementation specific. Users should consult the relevant documentation for their environment.

Design

The design of MicroProfile Reactive Streams Operators is centered around **builders** for the various shapes of streams.

Stages

Each builder contains zero or more stages. There are three different shapes of stages:

- Publisher. A publisher stage has an outlet, but no inlet.

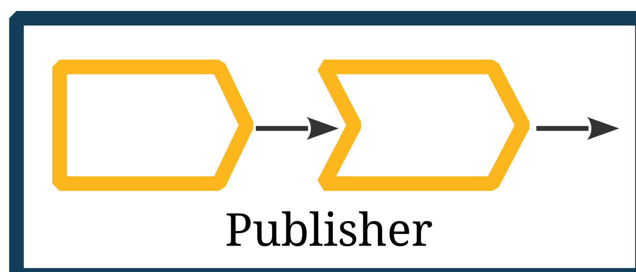
- Processor. A processor stage has an inlet and an outlet.

- Subscriber. A subscriber stage has an inlet, but no outlet.

Graphs

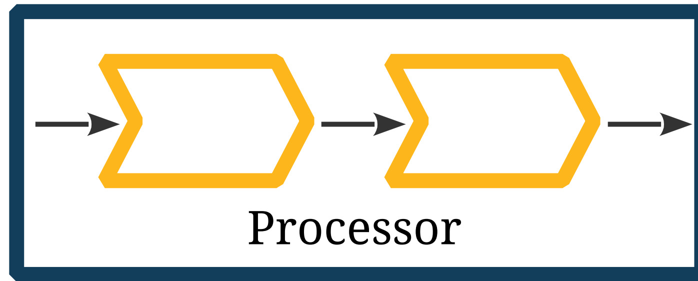
Stream stages can be built into graphs, using the builders. There are four different shapes of graphs that can be built:

- Publishers. A publisher has one outlet but no inlet, and is represented as a Reactive Streams `Publisher` when built. It contains one publisher stage, followed by zero or more processor stages. This is called a `PublisherBuilder`

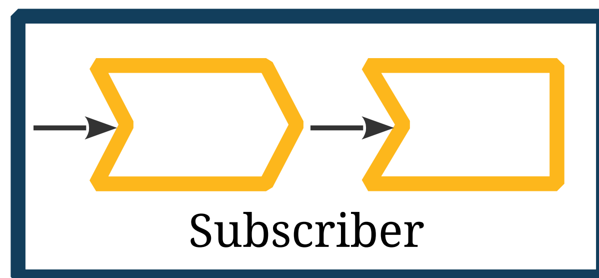


- Processors. A processor has one inlet and one outlet, and is represented as a Reactive Streams

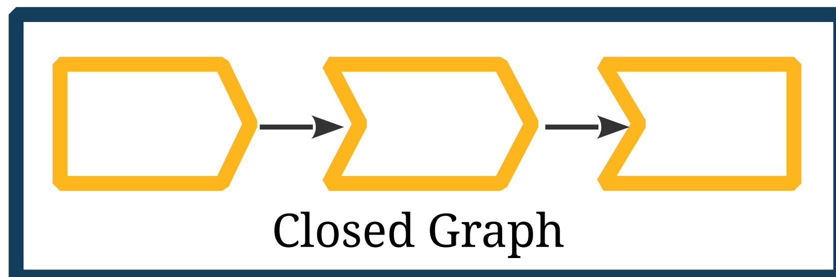
`Processor` when built. It contains zero or more processor stages. This is called a `ProcessorBuilder`.



- Subscribers. A subscriber has one inlet but no outlet, and it also has a result. It is represented as a product of a Reactive Streams `Subscriber` and a `CompletionStage` that is redeemed with the result, or error if the stream fails, when built. It contains zero or more processor stages, followed by a single subscriber stage. This is called a `SubscriberBuilder`.



- Closed graphs. A closed graph has no inlet or outlet, both having being provided in during the construction of the graph. It is represented as a `CompletionStage` of the result of the stream. It contains a publisher stage, followed by zero or more processor stages, followed by a subscriber stage. This is called a `CompletionRunner`. The result is retrieved using the `run` method.



While building a stream, the stream may change shape during its construction. For example, a publisher may be collected into a `List` of elements. When this happens, the stream becomes a closed graph, since there is no longer an outlet, but just a result, the result being the `List` of elements:

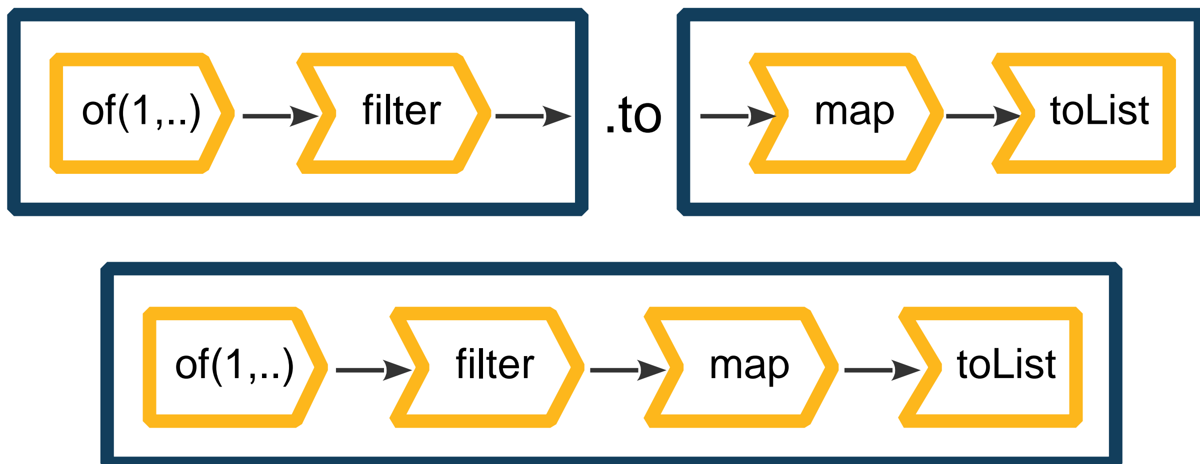
Here's an example of a more complex situation where a `PublisherBuilder` is plumbed to a `SubscriberBuilder`, producing a `CompletionRunner`:

```
PublisherBuilder<Integer> evenIntsPublisher =
  ReactiveStreams.of(1, 2, 3, 4)
    .filter(i -> i % 2 == 0); ①

SubscriberBuilder<Integer, List<Integer>> doublingSubscriber =
  ReactiveStreams.<Integer>builder()
    .map(i -> i = i * 2)
    .toList(); ②

CompletionRunner<List<Integer>> result =
  eventIntsPublisher.to(doublingSubscriber); ③
```

① A publisher of integers 2 and 4.

② A subscriber that first doubles integers, then collects into a list.

③ A closed graph that when run, will produce the result in a `CompletionStage`.



Usage

When MicroProfile specifications provide an API that uses Reactive Streams, it is intended that application developers can return and pass the builder interfaces directly to the MicroProfile APIs. In many cases, application developers will not need to run the streams themselves. However, should they need to run the streams directly themselves, they can do so by using the streams `build` or `run` methods. `PublisherBuilder`, `SubscriberBuilder` and `ProcessorBuilder` all provide a `build` method that returns a `Publisher`, `CompletionSubscriber` and `Processor` respectively, while `CompletionRunner`, since it actually runs the stream, provides a `run` method that returns a `CompletionStage`.

The `CompletionSubscriber` class is so named because, where a `CompletionStage` is a stage of asynchronous computation that completes with a value or an error, a `CompletionSubscriber` is subscriber to an asynchronous stream that completes with a value or an error.

The `build` and `run` methods both provide a zero arg variant, which uses the default Reactive Streams engine provided by the platform, as well as an overload that takes a `ReactiveStreamsEngine`, allowing application developers to use a custom engine when they please.

Implementation

The behavior of each stage is specified in the javadocs for stages in the SPI, and tested by the TCK.

Generally, across all stages, the following requirements exist:

Memory visibility

Within a single stage in a single stream, implementations must guarantee a *happens-before* relationship between any invocation of all callbacks supplied to that stage.

Between different stages of a single stream, it is recommended that a *happens-before* relationship exists between callbacks of different stages, however this is not required, and end users of the API must not depend on this. Implementations are expected to implement this for performance reasons, this is also known as operator fusing. Implementations may decide, for whatever reason, not to fuse stages in certain circumstances, when this is done, it is said that there is an asynchronous boundary between the two stages.

When Reactive Streams interfaces, that is, `Publisher`, `Subscriber` or `Processor`, are wrapped in a stage, implementations may place an asynchronous boundary between that stage and its neighbors if they choose. Whether implementations do or don't place an asynchronous boundary there, they must conform to the Reactive Streams specification with regards to memory visibility.

User exceptions

Exceptions thrown by user supplied callbacks must be caught and propagated downstream, unless otherwise handled by an error handling stage.

User exceptions must not be thrown by the `build` or `run` methods of the builders. For example, if a user supplies an `Iterable` as a source for a `PublisherBuilder`, and the `iterator()` method is invoked synchronously from the `build` method on the publisher, and it throws an exception, this exception must be caught, and propagated through the stream, not thrown from the `build` method.

An exception to this is the callbacks provided in user supplied Reactive Streams, ie `Publisher`, `Subscriber` and `Processor`. Since these interfaces are specified not to throw any exceptions when the consumer is adhering to the spec, implementations may assume that these interfaces won't throw exceptions, and the behavior of an implementation should these interfaces throw exceptions is unspecified.

In some cases, exceptions may be wrapped, for example, `CompletableFuture` wraps exceptions in a `CompletionException`. It is recommended that implementations do not wrap exceptions if they don't need to, but rather that they propagate them downstream as is.

Error propagation

Errors may be eagerly propagated by stages if an implementation chooses to do this. Such propagation can aid with fast failure of stages, to ensure things like connection failures do not wait excessively to discover that they have failed.

A consequence of this is that errors can in certain circumstances overtake elements. For example, the `flatMapCompletionStage` stage may receive an error while an element is being asynchronously processed. This error may immediately be propagated downstream, which will mean the eventual redemption of the `CompletionStage` returned by the mapper function for the stage will be ignored, and the element it is redeemed with will be dropped.

When a user does not desire this behavior, and wants to guarantee that stages don't drop elements when upstream errors arrive, they may insert an error recovery stage, such as `onErrorResume`, before the stage that they don't want to drop elements from. They will then need to implement an out of band mechanism to propagate the error, such as wrapping it in an element, should they wish to handle it downstream.

Cleanup

Implementations must assume that any `PublisherBuilder`, `SubscriberBuilder` or `ProcessorBuilder` supplied to them, or any stages within, potentially hold resources, and must be cleaned up when the stream shuts down. For example, a `concat` stage accepts two `PublisherBuilder's. If the stream is cancelled before the first is finished, the second must still be built, and then cancelled too.

null elements

Reactive Streams does not allow `null` elements. Hence, any user callbacks that return `null` as an element to be emitted by a stage must cause the stage to fail with a `NullPointerException`.

CDI Integration

MicroProfile Reactive Streams Operators implementations may be used independently of a CDI container. Consequently, implementations are not required to provide any integration with CDI.

This section of the specification is intended to provide advice for how other specifications may integrate CDI with MicroProfile Reactive Streams Operators.

Injection of engines

If a MicroProfile container provides an implementation of MicroProfile Reactive Streams Operators, then it must make an application scoped `ReactiveStreamsEngine` available for injection.

Contexts

This specification places no requirements on the propagation of CDI context, or what context(s) should be active when user supplied callbacks are executed during the running of a stream.

Other specifications that use this specification may require that implementations make certain context's to be active when user callbacks are executed. In this case, it is expected that such specifications will have the responsibility of running the streams.

For example, a hypothetical WebSocket specification may allow user code to return a `ProcessorBuilder` to handle messages:

```
@WebSocket("/echo")
public ProcessorBuilder<Message, Message> echoWebsocket() {
  return ReactiveStreams.<Message>builder().map(message ->
    new Message("Echoing " + message.getText())
  );
}
```

In this case, the implementation of that hypothetical WebSocket specification is responsible for running the `ProcessorBuilder` returned by the user, and that specification may require that the engine it uses to run it makes the request context active in callbacks, such as the `map` callback above. Since the implementation of that specification is in control of which engine is used to run the processor, this requirement can be made by that specification. It is the responsibility of that implementation (and the MicroProfile container that pulls these implementations together) to ensure that the MicroProfile Reactive Streams Operators implementation used is able to support CDI context propagation.

In contrast, if a user is responsible for executing a stream, like in the following hypothetical example:

```java
@WebSocket("/echo")
public void echoWebsocket(PublisherBuilder<Message> incoming,
  SubscriberBuilder<Message, Void> outgoing) {

  incoming.map(message ->
    new Message("Echoing " + message.getText()
  ).to(outgoing).run();
}
```

Then there is no clear way for the container to control how the engine used there, which in this case would be loaded through the Java `ServiceLoader` API, would propagate context. For this reason, any cases where users are running their own streams are not expected to require any CDI context propagation, and it is recommended that specifications favour APIs where the container runs the stream, not the user, to facilitate context propagation.

Reactive Streams Usage Examples

Trivial closed graph

This just shows the fluency of the API. It wouldn't make sense to actually do the below in practice, since the JDK8 streams API itself is better for working with in memory streams.

```
CompletionStage<Optional<Integer>> result = ReactiveStreams
    .fromIterable(() -> IntStream.range(1, 1000).boxed().iterator())
    .filter(i -> (i & 1) == 1)
    .map(i -> i * 2)
    .collect(Collectors.reducing((i, j) -> i + j))
    .run();
```

Building a publisher

This shows how common collection types can be converted to a `Publisher` of the elements in the collection.

```
List<MyDomainObject> domainObjects = ...

Publisher<ByteBuffer> publisher = ReactiveStreams
    .fromIterable(domainObjects)
    .buildRs(); ①
```

① The `Rs` suffix indicates the method produces a Reactive Streams `Publisher`.

The above example shows a very simple conversion of a `List` to a `Publisher`, of course other operations can be done on the elements before building the `Publisher`, in this case we go on to transform each object to a line in a CSV file, and then represent it as a stream of bytes.

```
Publisher<ByteBuffer> publisher = ReactiveStreams
    .map(obj -> String.format("%s,%s\n", obj.getField1(), obj.getField2()))
    .map(line -> ByteBuffer.wrap(line.getBytes()))
    .buildRs();
```

A `Publisher` built in this way has its behaviour specified by the Reactive Streams specification. Most reactive streams implementations enable the detailed control of further aspects of stream stage behaviour such as fan-out, back pressure policy, buffer sizes and so on. If you require control of such aspects, consult your implementation's documentation. One approach would be to create a single `Processor` using your implementation library's native API having the detailed behaviour required, subscribe that to the built `Publisher` and then subscribe downstream stream processing stages to that.

Building a subscriber

This shows building a subscriber for a byte stream, such as for the JDK9 HttpClient API. It assumes another library has provided a Reactive Streams Processor that parses byte streams into streams of objects.

```
Processor<ByteBuffer, MyDomainObject> parser = createParser();

CompletionSubscriber<ByteBuffer, List<MyDomainObject>> subscriber =
  ReactiveStreams.<ByteBuffer>builder()
    .via(parser)
    .toList()
    .build();

Subscriber<ByteBuffer> subscriber = subscriber; ①
CompletionStage<List<MyDomainObject>> result = subscriber.getCompletion(); ②
```

① The object can be deconstructed into the `Subscriber` part

② The `CompletionStage` can be retrieve using `getCompletion`

As a particular `SubscriberBuilder` may be constructed from a user supplied reactive streams `Subscriber` and subscribers can only be used once, care should be taken not to reuse builders where this may cause subscriber reuse, either for the emitted subscriber or a subscribing stage, such as a `Processor`, that is internal to a stream.

Building a processor

This shows building a processor, for example, a message library may require processing messages, and then emitting an ACK identifier so that each handled element can be acknowledged as handled.

```
Processor<Message<MyDomainObject>, MessageAck> processor =
    ReactiveStreams.<Message<MyDomainObject>>builder()
      .map(message -> {
        handleDomainObject(message.getMessage());
        return message.getMessageAck();
      })
      .buildRs();
  }
```

The comments above on avoiding inadvertent subscriber reuse are also relevant to `Processor` objects as these may contain internal subscription relationships as well as user supplied reactive streams `Processor` objects.

Consuming a publisher

A library may provide a Reactive Streams publisher that the application developer needs to

consume. This shows how that can be done.

```
Publisher<ByteBuffer> bytesPublisher = makeRequest();

Processor<ByteBuffer, MyDomainObject> parser = createParser();

CompletionStage<List<MyDomainObject>> result = ReactiveStreams
    .fromPublisher(bytesPublisher)
    .via(parser)
    .toList()
    .run();
```

Even though the MicroProfile Reactive Streams Operators API can be considered a 'lifted' API. The `parser` object above is directly embedded in the stream and such cannot be used to build more than one stream instance.

Feeding a subscriber

A library may provide a subscriber to feed a connection. This shows how that subscriber can be fed.

```
List<MyDomainObject> domainObjects = new ArrayList<>();

Subscriber<ByteBuffer> subscriber = createSubscriber();

CompletionStage<Void> completion = ReactiveStreams
    .fromIterable(domainObjects)
    .map(obj -> String.format("%s,%s\n", obj.getField1(), obj.getField2()))
    .map(line -> ByteBuffer.wrap(line.getBytes()))
    .to(subscriber)
    .run();
```

Similarities and differences with the Java Stream API

The API shares a lot of similarities with the Java Stream API. This similarity has been done on purpose to ease the adoption of the API. However, there are some differences and this section highlights them.

Asynchronous processing

The goal of the Reactive Stream Operators specification is to define building blocks to enable the implementation of asynchronous processing of stream of data. On the other hand, the Java Stream API provides a synchronous approach to compute a result by analyzing data conveyed in a stream. Because of this asynchronous vs. synchronous processing, the terminal stages (such as `collect`, `findFirst`…) define by this API return `CompletableStage<T>` and not `T`. Indeed, only when the result has been computed the returned `CompletableStage` is completed. As an example, here is the two versions of the same processing:

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Java Stream version
int sum = list.stream()
  .map(i -> i + 1)
  .mapToInt(i -> i)
  .sum();
// At the point the sum is computed
System.out.println("sum: " + sum);

// Reactive Streams Operators version
CompletionStage<Integer> future = ReactiveStreams.fromIterable(list)
  .map(i -> i + 1)
  .collect(Collectors.summingInt(i -> i))
  .run();
future.whenComplete((res, err) -> System.out.println("async sum: " + res));
```

The asynchronous vs. synchronous difference also means that the error propagation works differently. In the Java Streams API, the processing can be wrapped in a `try/catch` construct. In the asynchronous case, the error is propagated into the returned future. In the example above, the function passed to the `whenComplete` stage receives the result as well as the failure (if any). If the processing throws an exception, the function can react by looking at the `err` parameter.

No parallel processing

The Reactive Streams specification is intrinsically sequential. So none of the parallel processing ability from the Java Stream API are supported. As a consequence, the API does not provide a `parallel()` method. Also, operations like `findAny` are not provided as the behavior would be equivalent to the provided `findFirst` method.

Other differences

- `allMatch`, `anyMatch` and `nonMatch` can be achieved by combining `filter` and `findFirst`
- `collect(Collector<? super T,A,R> collector)` - the combiner part of the collector is not used because of the sequential nature of Reactive Streams.
- `collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)` is provided as `collect (Supplier<R> supplier, BiConsumer<R,? super T> accumulator)`. Indeed, the combiner is not used because of the sequential nature of Reactive Streams.
- `count` is not provided but can be implemented using `.collect(Collectors.counting())` instead.
- `findAny` is not supported, use `findFirst` instead. Because of the sequential nature of Reactive Streams, the method has the same semantic.
- `flatMapTo` and `mapTo` are not provided. These can easily be replaced using regular `flatMap` and `map` methods, or methods from `Collectors`.
- `forEachOrdered` is not provided as Reactive Streams mandates ordering. So `forEach` should be used instead.
- `max` and `min` can be achieved using `.collect(Collectors.maxBy(…))` and `.collect(Collectors.minBy(…))`
- `sorted` is not supported
- `toArray` is not supported, `toList` can be used instead
- `onClose` is replaced by `onComplete`. Notice that the API also provides the `onError` and `onTerminate` methods.

SPI

The API is responsible for building graphs of stages based on the operators the user invoked. This is done using builder classes that implement the fluent API that is used to compose the graph. These builders are obtained using the `ReactiveStreamsFactory` interface, a default implementation of which exists in the core package. The static methods of the `ReactiveStreams` class delegate to a service loaded implementation of `ReactiveStreamsFactory`, enabling the core implementation to be replaced.

The stages form an SPI for `ReactiveStreamsEngine` implementations to build into a running stream. Examples of stages include:

- Map
- Filter
- Elements to publish
- Collect
- Instances of Reactive Streams `Publisher`, `Subscriber` and `Processor`

Each stage has either an inlet, an outlet, or both. A graph is a sequence of stages, consecutive stages will have an outlet and and inlet so that they can join - a graph that has a stage with no outlet followed by a stage that has an inlet is impossible, for example. Only the stages at the ends of the graph may have no inlet or outlet, whether these end stages have an inlet or outlet determines the shape of the overall graph. The API is responsible for ensuring that as graphs are constructed, only graphs that are logically possible are passed to the `ReactiveStreamsEngine` to construct.

The implementation discovery relies on the Java `ServiceLoader` mechanism. However, for environments not supporting this mechanism, the user can pass custom implementations using the following methods:

- `org.eclipse.microprofile.reactive.streams.operators.core.ReactiveStreamsEngineResolver#setInstance` - to configure the `ReactiveStreamEngine`
- `org.eclipse.microprofile.reactive.streams.operators.spi.ReactiveStreamsFactoryResolver#setInstance` - to configure the `ReactiveStreamsFactory`

These methods must be called before the access to the API and should only be used for integration purpose.

TCK

The MicroProfile Reactive Streams Operators TCK is provided to test compliance of implementations to the specification. It provides a mechanism both for testing implementations independent from a MicroProfile container, as well as implementations provided by a container.

Structure

There are two parts to the TCK, an API TCK, and an SPI TCK. In addition, the whole TCK can either be run directly against an SPI implementation, or against a container with an Arquillian adapter.

The TCK uses TestNG for testing, and contains many test classes. These test classes are brought together using TestNG `@Factory` annotated methods, so that running the TCK can be done by simply running a single test class that is a factory for the rest of the test classes.

Additionally, this TCK also uses the Reactive Streams TCK. The Reactive Streams TCK is used to test Reactive Streams `Publishers`, `Subscribers` and `Processors` to ensure that they conform to the Reactive Streams spec. Wherever possible, this TCK will use it to verify the `Publishers`, `Subscribers` and `Processors` built by the API. For example, the returned `Processor` built by building a simple `map` stage is run through the Reactive Streams `Processor` TCK verification.

It should be noted that the Reactive Streams TCK is structured in such a way that each numbered requirement in the Reactive Streams specification has a test, even if that requirement is untestable by the TCK, or if its optional. In the case where requirements are optional or untested, the Reactive Streams TCK skips the test. Consequently, when running this TCK, there are a significant number of skipped tests.

API TCK

This tests the API, to ensure that every API call results in the a graph with the right stages being built. It also tests that the necessary validation, for example for nulls, is done on each API call. The API TCK is particularly useful for clean room implementations that don't depend on the Eclipse MicroProfile Reactive Streams Operators API artifact. Passing the API TCK ensures that different implementations of the SPI will be compatible with different implementations of the API.

The API TCK doesn't need an implementation of the SPI to run, and so can be run against any implementation of the API. It is run as part of the Eclipse MicroProfile Reactive Streams Operators API's continuous integration.

Running the API TCK

The API TCK can be run by running the `org.eclipse.microprofile.reactive.streams.operators.tck.api.ReactiveStreamsApiVerification` class. For convenience, implementations might decide to subclass this in their `src/test/java` folder, so that it gets automatically picked up and run by build tools like Maven.

SPI TCK

This tests implementations of the SPI, to ensure that each different stage defined by the SPI behaves correctly, and to ensure that the implementation of that stage conforms to the Reactive Streams specification. In general, each stage has a verification test, which tests the behavior of that stage, error, completion and cancellation propagation, and anything else that is necessary for each stage. Additionally, each stage defines one or more Reactive Streams TCK verification classes, which tests that the `Publisher`, `Processor` or `Subscriber` built by a graph that contains just that stage conforms to the Reactive Streams specification.

Running the SPI TCK

The SPI TCK can be run by running the whole TCK (this includes running the API TCK, since the SPI TCK uses the API to create the stages, it doesn't make sense to verify an SPI without verifying that the API on top of it is doing the right thing too).

The whole TCK can be run by creating a subclass of `org.eclipse.microprofile.reactive.streams.operators.tck.ReactiveStreamsTck`. This requires passing an instance of `org.reactivestreams.tck.TestEnvironment` to the super constructor, which contains configuration like timeouts. Typically, the timeouts set by using the default constructor for `TestEnvironment` should be satisfactory. The `createEngine` method also needs to be implemented, this must create and return a `ReactiveStreamsEngine` for the TCK to test against. Optionally, a `shutdownEngine` method can be overridden to shutdown the engine if it holds any resources like thread pools.

Example 1. An example of using the TCK

```
public class MyReactiveStreamsTckTest extends ReactiveStreamsTck<MyEngine> {

  public MyReactiveStreamsTckTest() {
    super(new TestEnvironment());
  }

  @Override
  protected MyEngine createEngine() {
    return new MyEngine();
  }

  @Override
  protected void shutdownEngine(MyEngine engine) {
    engine.shutdown();
  }
}
```

Running the TCK in a container

A test class for running the TCK in a MicroProfile container is provided so that containers can verify compliance with the spec. This container verification comes also in the

`org.eclipse.microprofile.reactive.streams:microprofile-reactive-streams-operators-tck` artifact, but the TCK is located in the `org.eclipse.microprofile.reactive.streams.operators.tck.arquillian` package.

To run this TCK, the container must provide a `ReactiveStreamsEngine` to be tested as an injectable `ApplicationScoped` bean, and the MicroProfile Reactive Streams Operators API must be on the classpath. Having ensured this, the TCK can then be run by executing `org.eclipse.microprofile.reactive.streams.operators.tck.arquillian.ReactiveStreamsArquillianTck` . This class deploys the TCK to an Arquillian compatible container, and then runs all the tests in the container in its own configured TestNG suite on the container.

For convenience, implementations may want to subclass this class in their own `src/test/java` class, so that it can automatically be run by build tools like Maven.