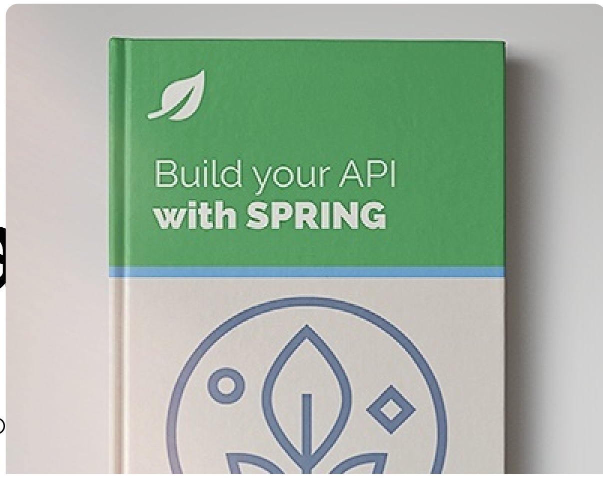
(http://baeldung.com)



Ja tive Streams

Last mo

by b /author/baeldung/)

Java (ory/java/) +

Java 9 -9/)

I just announced the new *Spring 5* modules in REST With Spring:

>> CHECK OUT THE COURSE (/rest-with-spring-course#new-modules)

1. Overview

In this article, we'll be looking at the Java 9 Reactive Streams. Simply put, we'll be able to use the *Flow* class, which encloses the primary building blocks for building reactive stream processing logic.

Reactive Streams is a standard for asynchronous stream processing with non-blocking back pressure. This specification is defined in the *Reactive Manifesto (http://www.reactive-streams.org/)*, and there are various implementations of it, for example, *RxJava* or *Akka-Streams.*

2. Reactive API Overview

To build a *Flow*, we can use three main abstractions and compose them into asynchronous processing logic.

Every *Flow* needs to process events that are published to it by a Publisher instance; the *Publisher* has one method – *subscribe().*

If any of the subscribers want to receive events published by it, they need to subscribe to the given *Publisher.*

The receiver of messages needs to implement the *Subscriber* interface. Typically this is the end for every *Flow* processing because the instance of it does not send messages further.

We can think about *Subscriber* as a *Sink.* This has four methods that need to be overridden – *onSubscribe(), onNext(), onError(),* and *onComplete().* We'll be looking at those in the next section.

If we want to transform incoming message and pass it further to the next *Subscriber,* we need to implement the *Processor* interface. This acts both as a *Subscriber* because it receives messages, and as the *Publisher* because it processes those messages and sends them for further processing.

3. Publishing and Consuming Messages

Let's say we want to create a simple *Flow,* in which we have a *Publisher* publishing messages, and a simple *Subscriber* consuming messages as they arrive – one at the time.

Let's create an *EndSubscriber* class. We need to implement the *Subscriber* interface. Next, we'll override the required methods.

The *onSubscribe()* method is called before processing starts. The instance of the *Subscription* is passed as the argument. It is a class that is used to control the flow of messages between *Subscriber* and the *Publisher:*

```
public class EndSubscriber<T> implements Subscriber<T> {
    private Subscription subscription;
    public List<T> consumedElements = new LinkedList<>();

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }
}
```

We also initialized an empty *List* of *consumedElements* that'll be utilized in the tests.

Now, we need to implement the remaining methods from the *Subscriber* interface. The main method here is onNext() – this is called whenever the *Publisher* publishes a new message:

```
@Override
public void onNext(T item) {
    System.out.println("Got : " + item);
    subscription.request(1);
}
```

Note that when we started the subscription in the *onSubscribe()* method and when we processed a message we need to call the *request()* method on the *Subscription* to signal that the current *Subscriber* is ready to consume more messages.

Lastly, we need to implement *onError()* – which is called whenever some exception will be throw in the processing, as well as *onComplete()* – called when the *Publisher* is closed:

```
@Override
public void onError(Throwable t) {
    t.printStackTrace();
}

@Override
public void onComplete() {
    System.out.println("Done");
}
```

Let's write a test for the Processing *Flow.* We'll be using the *SubmissionPublisher* class – a construct from the *java.util.concurrent* – which implements the *Publisher* interface.

We're going to be submitting *N* elements to the *Publisher* – which our *EndSubscriber* will be receiving:

```
@Test
public void whenSubscribeToIt_thenShouldConsumeAll()
  throws InterruptedException {

    // given
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
    EndSubscriber<String> subscriber = new EndSubscriber<>();
    publisher.subscribe(subscriber);
    List<String> items = List.of("1", "x", "2", "x", "3", "x");

    // when
    assertThat(publisher.getNumberOfSubscribers()).isEqualTo(1);
    items.forEach(publisher::submit);
    publisher.close();

    // then
     await().atMost(1000, TimeUnit.MILLISECONDS)
       .until(
         () -> assertThat(subscriber.consumedElements)
         .containsExactlyElementsOf(items)
     );
}
```

Note, that we're calling the *close()* method on the instance of the *EndSubscriber.* It will invoke *onComplete()* callback underneath on every *Subscriber* of the given *Publisher.*

Running that program will produce the following output:

```
Got : 1
Got : x
Got : 2
Got : x
Got : 3
Got : x
Done
```

4. Transformation of Messages

Let's say that we want to build similar logic between a *Publisher* and a *Subscriber*, but also apply some transformation.

We'll create the *TransformProcessor* class that implements *Processor* and extends *SubmissionPublisher* – as this will be both *Publisher* and S*ubscriber.*

We'll pass in a *Function* that will transform inputs into outputs:

```
public class TransformProcessor<T, R>
  extends SubmissionPublisher<R>
  implements Flow.Processor<T, R> {

    private Function<T, R> function;
    private Flow.Subscription subscription;

    public TransformProcessor(Function<T, R> function) {
        super();
        this.function = function;
    }

    @Override
    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }

    @Override
    public void onNext(T item) {
        submit(function.apply(item));
        subscription.request(1);
    }

    @Override
    public void onError(Throwable t) {
        t.printStackTrace();
    }

    @Override
    public void onComplete() {
        close();
    }
}
```

Let's now **write a quick test** with a processing flow in which the *Publisher* is publishing *String* elements.

Our *TransformProcessor* will be parsing the *String* as *Integer* – which means a conversion needs to happen here:

```
@Test
public void whenSubscribeAndTransformElements_thenShouldConsumeAll()
  throws InterruptedException {

    // given
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
    TransformProcessor<String, Integer> transformProcessor
      = new TransformProcessor<>(Integer::parseInt);
    EndSubscriber<Integer> subscriber = new EndSubscriber<>();
    List<String> items = List.of("1", "2", "3");
    List<Integer> expectedResult = List.of(1, 2, 3);

    // when
    publisher.subscribe(transformProcessor);
    transformProcessor.subscribe(subscriber);
    items.forEach(publisher::submit);
    publisher.close();

    // then
     await().atMost(1000, TimeUnit.MILLISECONDS)
       .until(() ->
         assertThat(subscriber.consumedElements)
         .containsExactlyElementsOf(expectedResult)
     );
}
```

Note, that calling the *close()* method on the base *Publisher* will cause the *onComplete()* method on the *TransformProcessor* to be invoked.

Keep in mind that all publishers in the processing chain need to be closed this way.

5. Controlling Demand for Messages Using the *Subscription*

Let's say that we want to consume only the first element from the Subscription, apply some logic and finish processing. We can use the *request()* method to achieve this.

Let's modify our *EndSubscriber* to consume only N number of messages. We'll be passing that number as the *howMuchMessagesConsume* constructor argument:

```
public class EndSubscriber<T> implements Subscriber<T> {

    private AtomicInteger howMuchMessagesConsume;
    private Subscription subscription;
    public List<T> consumedElements = new LinkedList<>();

    public EndSubscriber(Integer howMuchMessagesConsume) {
        this.howMuchMessagesConsume
          = new AtomicInteger(howMuchMessagesConsume);
    }

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }

    @Override
    public void onNext(T item) {
        howMuchMessagesConsume.decrementAndGet();
        System.out.println("Got : " + item);
        consumedElements.add(item);
        if (howMuchMessagesConsume.get() > 0) {
            subscription.request(1);
        }
    }
    //...

}
```

We can request elements as long we want to.

Let's write a test in which we only want to consume one element from the given *Subscription:*

```
@Test
public void whenRequestForOnlyOneElement_thenShouldConsumeOne()
  throws InterruptedException {

    // given
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
    EndSubscriber<String> subscriber = new EndSubscriber<>(1);
    publisher.subscribe(subscriber);
    List<String> items = List.of("1", "x", "2", "x", "3", "x");
    List<String> expected = List.of("1");

    // when
    assertThat(publisher.getNumberOfSubscribers()).isEqualTo(1);
    items.forEach(publisher::submit);
    publisher.close();

    // then
    await().atMost(1000, TimeUnit.MILLISECONDS)
      .until(() ->
        assertThat(subscriber.consumedElements)
       .containsExactlyElementsOf(expected)
    );
}
```

Although the *publisher* is publishing six elements, our *EndSubscriber* will consume only one element because it signals demand for processing only that single one.

By using the *request()* method on the *Subscription*, we can implement a more sophisticated back-pressure mechanism to control the speed of the message consumption.

6. Conclusion

In this article, we had a look at the Java 9 Reactive Streams.

We saw how to create a processing *Flow* consisting of a *Publisher* and a *Subscriber*. We created a more complex processing flow with the transformation of elements using *Processors*.

Finally, we used the *Subscription* to control the demand for elements by the *Subscriber*.

The implementation of all these examples and code snippets can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/core-java-9) – this is a Maven project, so it should be easy to import and run as it is.