Reactive streams programming over WebSockets with Helidon SE

The WebSocket server endpoint as a subscriber

The WebSocket client endpoint as a publisher

Handling error signals with Java

Full-stack reactive coding with JavaScript

Handling error signals with JavaScript

Conclusion

Dig deeper

REACTIVE PROGRAMMING

Reactive streams programming over WebSockets with Helidon SE

With Helidon SE, client applications can regulate asynchronous traffic by signaling remote publishers how much data to send at a time.

by Daniel Kec

September 11, 2020

Download a PDF of this article

Reactive streams programming is gaining popularity as a way to handle asynchronous stream processing, and more and more APIs are adopting a reactive approach. With a reactive approach, everything has to be asynchronous and nonblocking—and the implementation needs a mechanism for feedback to regulate data flow, so subscribers won't be drowned by faster publishers.

This need for bidirectional asynchronous communication can constrain the developer's options for streaming data remotely over the network. There are heavy-duty messaging systems that can address this problem, of course, and they can tame large amounts of asynchronous traffic flowing over the network. However, those are usually big guns requiring considerable heavy infrastructure. But what about simple use cases of server-to-client reactive connections, which aren't sending huge amounts of data and where the client signals the server how much data to send?

However, the best solution, I believe, is to employ the Reactive Streams API. With this API and bidirectional communication in the streams, an application can signal the publishers how much stuff it can work with—in other words, it can apply the necessary

backpressure to start and stop the stream as needed or desired. See **Figure 1**.

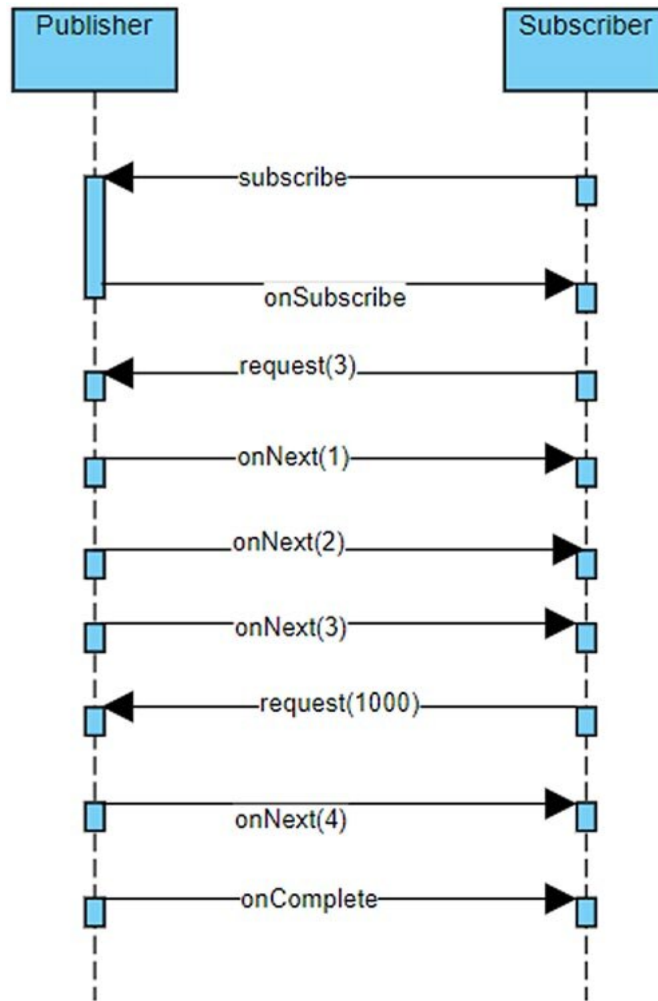

Figure 1. The publish/subscribe model for the Reactive Streams API

There are existing solutions for connecting publishers to subscribers via the network, such as RSocket, Reactive gRPC, and ServiceTalk. Those are specification-compliant and ready to use.

But if you are already working with a bidirectional network protocol such as WebSocket with the Helidon SE microservices library for Java, how hard would it be to simply use WebSocket directly for connecting client subscribers to remote publishers? That's what I'll explore in this article.

To be clear, implementing Reactive Streams for JVM API by yourself can be tricky and is usually discouraged because this seemingly simple API has a complicated specification. But as a reward, you can have total control over the serialization of the stream items and more versatility to recover from network issues.

The WebSocket protocol is actually well suited for such a task, and I'll show you the benefits and drawbacks of using WebSocket with Helidon SE. I'll create a WebSocket server with

Helidon SE for publishing to remote reactive subscribers, define custom reactive signals, serialize them to JSON, and connect to the server (via the stream) from Java and from JavaScript-based subscribers.

And all of that is end-to-end reactive.

You can find all the examples for this article on GitHub.

Before continuing, be aware that Helidon supports two programming models for writing microservices:

- Helidon SE, which I am using here, is a microframework that supports the reactive programming model.
- Helidon MP is an Eclipse MicroProfile runtime that allows the Jakarta EE community to run microservices in a portable way.

The WebSocket server endpoint as a subscriber

To connect reactive streams over a network, an application needs an intermediate reactive subscriber on the server side and a publisher on the client side. This may seem a little confusing since the server side is actually the one doing the publishing—but you should consider the WebSocket network bridge as a processor in the middle of the stream. Thus, the server side has to act as a subscriber for its upstream, while the client side acts a publisher for its downstream. See **Figure 2**.

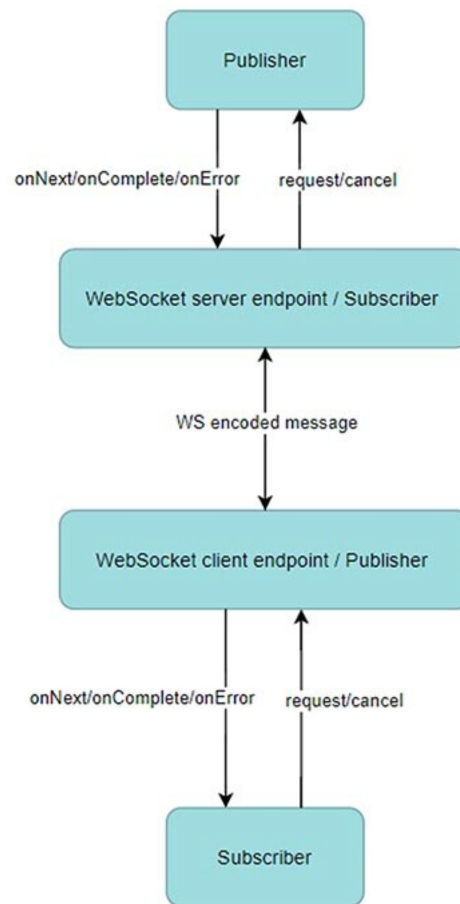

Figure 2. The upstream and downstream of the WebSocket network bridge

The WebSocket API is always connecting client endpoints to the server endpoint. The biggest difference between those two kinds of endpoints is that the server endpoint is instantiated, by default, by the WebSocket implementation for every client connection. Thus, to use the server endpoint as a subscriber, I must access the instance and subscribe it to the reactive stream after the connection is made.

First, I will use a `StreamFactory` class to publishers for this example application to supply a publisher every time the client is connected. The factory will return a simple new stream of the 10 string items emitted at half-second intervals.

```
public class StreamFactory {

   private final ScheduledExecutorService sch

   public Multi<String> createStream() {
       return Multi.interval(500, TimeUnit.MI
               .limit(10)
               .map(aLong -> "Message number:
   }
}
```

Now I'll create a simple custom wrapper to differentiate signals sent over WebSocket. For this example, let's assume it's a stream of strings. To make things simple, I'm not propagating subscribe or `onSubscribe` signals in this example. The impact of that simplification is that the application must wait for the WebSocket connection to be ready before sending a request signal, so the abstraction is not really perfect.

```
public class ReactiveSignal {

   public Type type;
   public long requested;
   public String item;
   public Throwable error;

   public enum Type {
       REQUEST,
       CANCEL,
       ON_NEXT,
       ON_ERROR,
       ON_COMPLETE
   }

public static ReactiveSignal request(long n)
   ReactiveSignal signal = new ReactiveSignal
   signal.type = Type.REQUEST;
   signal.requested = n;
   return signal;
}

public static ReactiveSignal cancel() {
   ReactiveSignal signal = new ReactiveSignal
   signal.type = Type.CANCEL;
   return signal;
}

public static ReactiveSignal next(String item
   ReactiveSignal signal = new ReactiveSignal
   signal.type = Type.ON_NEXT;
   signal.item = item;
   return signal;
}

public static ReactiveSignal error(Throwable
   ReactiveSignal signal = new ReactiveSignal
   signal.type = Type.ON_ERROR;
   signal.error = t;
   return signal;
}

public static ReactiveSignal complete() {
   ReactiveSignal signal = new ReactiveSignal
   signal.type = Type.ON_COMPLETE;
   return signal;
}

}
```

For this example, it's best to encode the WebSocket messages with JSON-B. This will pay off later, when I connect to the server from JavaScript.

```
public class ReactiveSignalEncoderDecoder
        implements Encoder.TextStream<Reactive

  private static final Jsonb jsonb = JsonbBui

    @Override
    public ReactiveSignal decode(final Reader 
        return jsonb.fromJson(reader, Reactive
    }

    @Override
    public void encode(final ReactiveSignal ob
        writer.write(jsonb.toJson(object));
    }

    @Override
    public void init(final EndpointConfig conf
    }

    @Override
    public void destroy() {
    }
}
```

Next I will prepare a WebSocket endpoint, which also will be a `Flow.Subscriber`, which means it can directly subscribe to the publisher created by `StreamFactory`. In this code, I'll assume the subscription needs to be realized before the endpoint can intercept a WebSocket message.

```
public class WebSocketServerEndpoint extends

    private static final Logger LOGGER = Logge

    private Session session;
    private Flow.Subscription subscription;

    @Override
    public void onOpen(Session session, Endpoi
        this.session = session;
        System.out.println("Session " + sessio

        session.addMessageHandler(new MessageH
            @Override
            public void onMessage(ReactiveSign
                System.out.println("Message "
                switch (signal.type) {
                    case REQUEST:
                        subscription.request(s
                        break;
                    case CANCEL:
                        subscription.cancel();
                        break;
                    default:
                        throw new IllegalState
                }
            }
        });
    }

    @Override
```

```
    public void onError(final Session session,
        LOGGER.log(Level.SEVERE, thr, () -> "W
        super.onError(session, thr);
    }

    @Override
    public void onClose(final Session session,
        super.onClose(session, closeReason);
        subscription.cancel();
    }

    @Override
    public void onSubscribe(final Flow.Subscri
        this.subscription = subscription;
    }

    @Override
    public void onNext(final String item) {
        sendSignal(ReactiveSignal.next(item));
    }

    @Override
    public void onError(final Throwable throwa
        sendSignal(ReactiveSignal.error(throwa
        try {
            session.close(new CloseReason(Clos
        } catch (IOException e) {
            LOGGER.log(Level.SEVERE, e, () ->
        }
    }

    @Override
    public void onComplete() {
        sendSignal(ReactiveSignal.complete());
        try {
            session.close(new CloseReason(Clos
        } catch (IOException e) {
            LOGGER.log(Level.SEVERE, e, () ->
        }
    }

    private void sendSignal(ReactiveSignal sig
        session.getAsyncRemote().sendObject(si
    }
}
```

Notice that I am using WebSocket's AsyncRemote to send messages asynchronously. This is necessary because it's forbidden to block threads in reactive pipelines.

The only thing missing now is starting Helidon SE as a WebSocket server. Once that's done, every created endpoint/subscriber is subscribed to the new publisher supplied by the `StreamFactory` when a client connection is created. That way, the upstream subscription is ready when the first request signal from the downstream client arrives over WebSocket.

```
StreamFactory streamFactory = new StreamFacto

TyrusSupport tyrusSupport = TyrusSupport.buil
        .register(
```

```
                 ServerEndpointConfig.Builder.c
                         WebSocketServerEndpoin
                         .encoders(List.of(Reac
                         .decoders(List.of(Reac
                         .configurator(new Serv
                             @Override
                             public <T> T getEn
                                     throws Ins
                                 T endpointInst
                                 if (endpointIn
                                     WebSocketS
                                             (W
                                     //Endpoint
                                     streamFact
                                 }
                                 return endpoin
                             }
                         })
                         .build())
         .build();

Routing routing = Routing.builder()
        .register("/ws", tyrusSupport)
        .build();

WebServer.builder(routing)
        .build()
        .start();
```

That's it for the server side!

The WebSocket client endpoint as a publisher

The next step is to use `Flow.Publisher` for the client side, so the application has something to subscribe to. There are many specification rules for the publisher to comply with, but the most pressing issues are to signal subscriber methods serially (rule 1.3 of using the Reactive Streams spec for the JVM) and to not block signals from downstream (rules 3.4 and 3.5). I will leverage Helidon SE's SequentialSubscriber class as a wrapper for the actual subscriber, to defend it from wildly asynchronous signals coming over a WebSocket. To ensure the request/cancel signals are nonblocking and nonobstructing, I will simply use WebSocket's AsyncRemote to send the signals upstream, as was done on the server side.

```
public class WebSocketClientEndpoint extends

   private static final Logger LOGGER = Logge

   private Session session;
   private Flow.Subscriber<? super String> su

   @Override
   public void onOpen(final Session session,
       this.session = session;
       session.addMessageHandler(new MessageH
           @Override
           public void onMessage(ReactiveSign
               switch (signal.type) {
```

```
                    case ON_NEXT:
                        subscriber.onNext(sign
                        break;
                    case ON_ERROR:
                        subscriber.onError(sig
                        break;
                    case ON_COMPLETE:
                        subscriber.onComplete(
                        break;
                    default:
                        subscriber.onError(new
                }
            }
        });
    }

    @Override
    public void onError(final Session session,
        Optional.ofNullable(subscriber).ifPres
        LOGGER.log(Level.SEVERE, thr, () -> "C
        super.onError(session, thr);
    }

    @Override
    public void onClose(final Session session,
        subscriber.onComplete();
        super.onClose(session, closeReason);
    }

    @Override
    public void subscribe(final Flow.Subscribe
        Objects.requireNonNull(subscriber, "su
        // Notice usage of Helidon's Sequentia
        // to get around difficulties with spe
        this.subscriber = SequentialSubscriber
        subscriber.onSubscribe(this);
    }

    @Override
    public void request(final long n) {
        sendAsyncSignal(ReactiveSignal.request
    }

    @Override
    public void cancel() {
        sendAsyncSignal(ReactiveSignal.cancel(
    }

    private void sendAsyncSignal(ReactiveSigna
        try {
            //reactive means no blocking
            session.getAsyncRemote().sendObjec
        } catch (Exception e) {
            subscriber.onError(e);
        }
    }
}
```

Now I just have to connect and request something, reusing the same encoder for serializing messages. This time the test application is creating only one connection, so I can instantiate the client endpoint myself.

```
public class Client {

    private static final Logger LOGGER = Logge

    public static void main(String[] args)
            throws URISyntaxException, Deploym

        ClientManager client = ClientManager.c
        WebSocketClientEndpoint endpoint = new

        Future<Session> sessionFuture = client
                ClientEndpointConfig.Builder
                        .create()
                        .encoders(List.of(Reac
                        .decoders(List.of(Reac
                        .build(),
                new URI("ws://localhost:8080/w

        //Wait for the connection
        sessionFuture.get();

        //Subscribe to the publisher and wait
        Multi.create(endpoint)
                .onError(throwable -> LOGGER.l
                .onComplete(() -> LOGGER.log(L
                .forEach(s -> System.out.print
                .await();
    }
}
```

The output should look like this, with 10 items coming in half-second intervals followed by the complete signal:

```
Received item> Message number: 0
Received item> Message number: 1
Received item> Message number: 2
Received item> Message number: 3
Received item> Message number: 4
Received item> Message number: 5
Received item> Message number: 6
Received item> Message number: 7
Received item> Message number: 8
Received item> Message number: 9
Jul 30, 2020 5:34:22 PM io.helidon.fs.reactiv
INFO: Complete signal received!
```

As you can see, I have subscribed to the WebSocket publisher with `forEach`, which requests `Long.MAX_VALUE` and means the subscriber is confident it is able to consume any number of items. Luckily, the upstream sends only 10 items and then finishes.

Handling error signals with Java

What if something goes wrong? Let's find out. First, let's make sure the client code is logging error signals and reports a problem.

```
Multi.create(endpoint)
        .onError(throwable -> LOGGER.log(Level
        .onComplete(() -> LOGGER.log(Level.INF
        .forEach(s -> System.out.println("Rece
        .await();
```

And now, to introduce a fault, I'll tell the `StreamFactory` to produce an error signal as the fourth item.

```
public class StreamFactory {

   public Multi<String> createStream() {
       return Multi.concat(Multi.just(1, 2, 3
               .map(aLong -> "Message number:
   }
}
```

Running the code shows the following results. I can thank the JSON-B adapter for providing a stack trace:

```
Received item> Message number: 1
Received item> Message number: 2
Received item> Message number: 3
Jul 31, 2020 4:30:11 PM io.helidon.fs.reactiv
SEVERE: Error from upstream!
java.lang.Throwable: BOOM!
  at app//io.helidon.fs.reactive.StreamFactor
  at app//io.helidon.fs.reactive.Server$1.get
  at app//org.glassfish.tyrus.core.TyrusEndpo
```

That concludes the Java-to-Java solution with reactive streams. How about getting a little polyglot?

Full-stack reactive coding with JavaScript

When there's a reactive WebSocket endpoint on the back end, why not try to connect it to the reactive pipeline on the front end?

Let's connect the application to a Reactive Extensions for JavaScript (RxJS) stream. To stay end-to-end reactive, I'll use reactive operators to map custom signals to the RxJS stream. The snippet below leverages the `takeWhile` operator for detecting the complete signal; when the `ON_COMPLETE` type arrives, the RxJS stream is completed.

For mapping the error signal, RxJS's `flatMap` equivalent, called `mergeMap`, is the most suitable. I'll use it for mapping any items of `ON_ERROR` type to stream errors and flatten errors into the main stream. In case the error is `ON_NEXT`, I'll just unwrap the item with flattening using `of(msg.item)`.

```
const { Observable, of, from, throwError} = r
const { map, takeWhile, mergeMap } = rxjs.ope
```

```
const { WebSocketSubject } = rxjs.webSocket;

const subject = new WebSocketSubject('ws://12

// Now I have to map the custom signals to Rx
subject.pipe(
   // Map the custom ON_COMPLETE to RxJS comp
   takeWhile(msg => msg.type !== 'ON_COMPLETE
   // Map the custom ON_ERROR to RxJS error s
   mergeMap(msg => msg.type === 'ON_ERROR' ?
)
.subscribe(
   // invoked for every item
   msg => onNext(msg),
   // invoked when error signal is intercepte
   err => console.log(JSON.stringify(err, nul
   // invoked when complete signal is interce
   () => console.log('complete')
);
```

When I connect this up, nothing happens. That's because the back end expects requests for a number of items, since it is backpressure-aware. So, I'll add a button to send a custom request signal:

```
const input = $("#input");
const submit = $("#submit");

submit.on("click", onSubmit);

function onSubmit() {
   subject.next({"requested":input.val(),"typ
}
```

You can use the entire working example, which is available from GitHub, to try to request any number of items and see the results visually. Since the stream is initialized for every new connection, after you deplete your 10 items and the stream is completed, simply reload the page to start again. **Figure 3** shows the user interface.

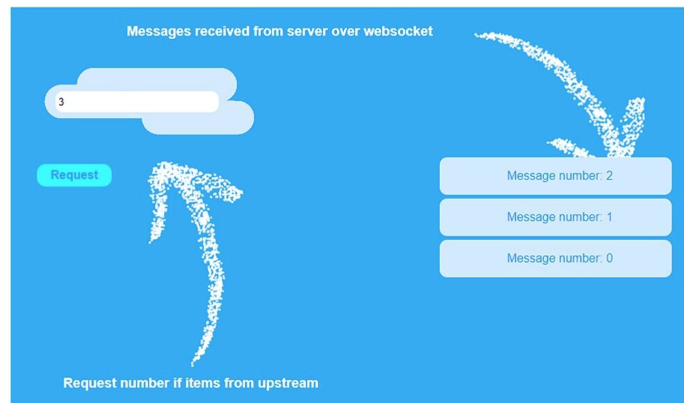

Figure 3. The user interface for the JavaScript front end

Handling error signals with JavaScript

Let's change `StreamFactory` again to produce an error signal as the fourth item:

```
public class StreamFactory {

    public Multi<String> createStream() {
        return Multi.concat(Multi.just(1, 2, 3
                .map(aLong -> "Message number:
    }
}
```

The custom error signal gets encoded to JSON. **Figure 4** shows the front end where it gets printed to the console:

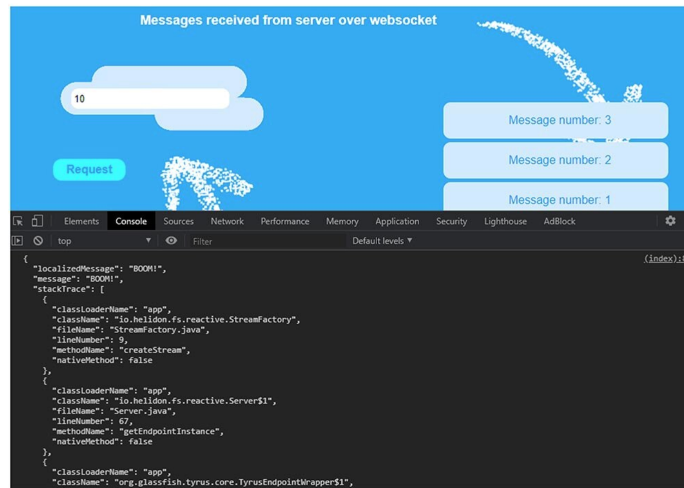

Figure 4. Reactive stream error handling with JavaScript and JSON-B

As you can see, the application has a whole exception with a Java stack trace, thanks to the fact that JSON-B is used for encoding the custom signals.

Conclusion

WebSockets are powerful enough to support reactive streams communications. That's good for relatively small or constrained application use cases. For streams that are expected to be long and that might have millions of items flowing through them, heavier tooling is required.

If you are interested in the topic, I can recommend MicroProfile Reactive Messaging, which is available in Helidon MP. You can also see the non-CDI API, which has been in Helidon SE since version 2.0.0.

Dig deeper

- "Helidon Takes Flight"
- "Microservices From Dev To Deploy, Part 1: Getting Started With Helidon"
- Project Helidon

- Overview of reactive programming
- “Reactive Programming with JAX-RS”
- “Reactive Programming with JDK 9 Flow API”
- “Going Reactive with Eclipse Vert.x and RxJava”
- The WebSocket API
- Helidon Full Stack Reactive on GitHub

Daniel Kec

Daniel Kec is a Java developer working for Oracle in Prague, where he focuses on the Helidon Project.

Share this Page