JAX-RS 2.1 Reloaded

Santiago Pericas-Geertsen
JAX-RS Co-Spec Lead

Agenda

- Reactive Extensions
- Server-Sent Events
- Non-Blocking IO

Reactive Extensions

Asynchronous Processing in 2.0

Server side:

- Using @Suspended and AsyncResponse
- Resume execution on a different thread

Client side:

- Future<T>
- InvocationCallback<T>

Example Using Future<T>

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target(“http://…”);

Future<String> f =
    target.request().async().get(String.class);

// some time later …

String user = f.get();
```

What’s the problem here?

Example using InvocationCallback<T>

```
WebTarget target = client.target("http://…");

target.request().async().get(
   new InvocationCallback<String>() {
        @Override
        public void completed(String user) {
            // do something
        }
        @Override
        public void failed(Throwable t) {
            // do something
        } });
```

Example using InvocationCallback<T>

```
target1.request().async().get(new
    InvocationCallback<String>() {
        public void completed(String user) {
            target2.request().header(“user”, user).async().get(
                new InvocationCallback<String>() {
                    public void completed(String quote) {
                       System.out.println(quote);
                    }
                    public void failed(Throwable t) {
                       // do something
                    }
                }); }
        public void failed(Throwable t) {
            // do something
        }}));
```

Some use cases for Async Computations

- Compose two or more asynchronous tasks
- Combine the output of two or more asynchronous tasks
- Consume value of asynchronous task
- Wait for all tasks in a collection to complete
- Wait for any of the tasks in a collection to complete

And many more …

Meet CompletionStage<T> in JAX-RS

```
CompletionStage<String> cs1 =
    target1.request().rx().get(String.class);

CompletionStage<String> cs2 =
    cs1.thenCompose(user ->
      target2.request().header(“user”, user)
         .rx().get(String.class));

cs2.thenAccept(quote -> System.out.println(quote));
```

What about other Rx libraries?

```
Client client =
  client.register(ObservableRxInvokerProvider.class);

Observable<String> of =
    client.target("forecast/{destination}”)
          .resolveTemplate("destination", "mars”)
          .request()
          .rx(ObservableRxInvoker.class)
          .get(String.class);

of.subscribe(System.out::println);
```

Register a Provider

Override default Invoker

Server-Sent Events Summary

- New invoker to support Rx
- Default support for `CompletionStage`
- Extension mechanism for other Rx libraries

Server-Sent Events

Server-Sent Events

- Originally W3C (HTML5), now maintainted by WHATWG
- HTTP-based protocol for one-way server to client messaging
- Special media type `text/event-stream`

Client API

Publisher

```
try (SseEventSource source =
         SseEventSource.target(“http://…”).build()) {
   source.subscribe(System.out::println);
   source.open();
   Thread.sleep(500);
} catch (InterruptedException e) {
   // falls through
}
```

Server API

```
@GET
@Produces(“text/event-stream”)
public void eventStream(
    @Context SseEventSink eventSink,
    @Context Sse sse) {
    executor.execute(() -> {
        try (SseEventSink sink = eventSink) {
            eventSink.onNext(sse.newEvent("event1"));
            eventSink.onNext(sse.newEvent("event2"));
            eventSink.close();
        } catch (IOException e) {
            // handle exception
        }  });}
```

Subscriber

Broadcasting (1 of 2)

```
@Path("/")
@Singleton
public class SseResource {

    private final Sse sse;
    private final SseBroadcaster sseBroadcaster;

    public SseResource(@Context Sse sse) {
        this.sse = sse;
        this.sseBroadcaster = sse.newBroadcaster();
    }
…
```

One Publisher

Lifecycle controlled by App

Broadcasting (2 of 2)

```
@GET @Path(“subscribe”)
@Produces(MediaType.SERVER_SENT_EVENTS)
public void subscribe(@Context SseEventSink eventSink) {
    eventSink.onNext(sse.newEvent("welcome!"));
    sseBroadcaster.subscribe(eventSink);
}

@POST @Path(“broadcast”)
@Consumes(MediaType.MULTIPART_FORM_DATA)
public void broadcast(@FormParam("event") String event) {
    sseBroadcaster.broadcast(sse.newEvent(event));
} … }
```

Many Subscribers

Reactive Extensions Summary

- New types `SseEventSink`, `SseEventSource`, `Sse` and `SseBroadcaster`
- `Sse` and `SseEventSink`’s lifecycle controlled by runtime
- Singleton scope useful for broadcasting

Non-Blocking IO

Motivation

- Certain apps need more control over IO
- Higher throughput is hard with blocking IO
- Precedence with `StreamingOutput`

StreamingOutput in JAX-RS

```
@GET
public Response get() {
    return Response.ok(new StreamingOutput() {
        @Override
        public void write(OutputStream out) throws … {
            out.write(…);
        }
    }).build();
}
```

Still blocking!

Direct access to stream

First NIO Proposal

```
return Response.ok().entity(
        out -> {
            try {
                final int n = in.read(buffer);
                if (n >= 0) {
                    out.write(buffer, 0, n);
                    return true;    // more to write
                }
                in.close();
                return false;       // we're done
            } catch (IOException e) { … }
        }).build();
```

Write handler

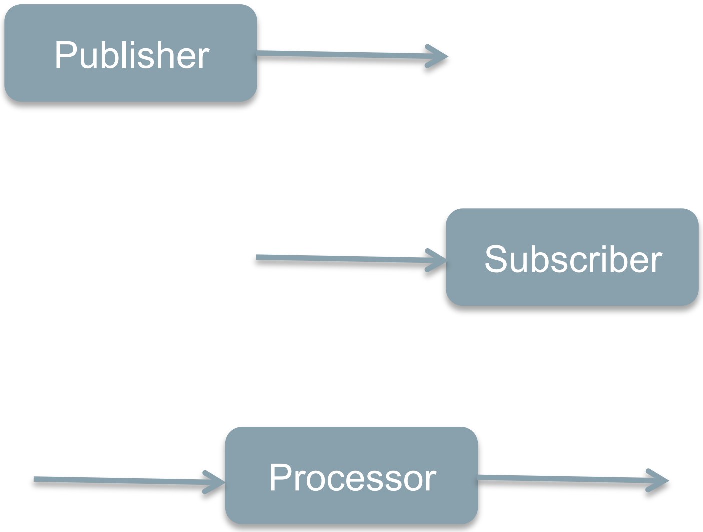
First Proposal Limitations

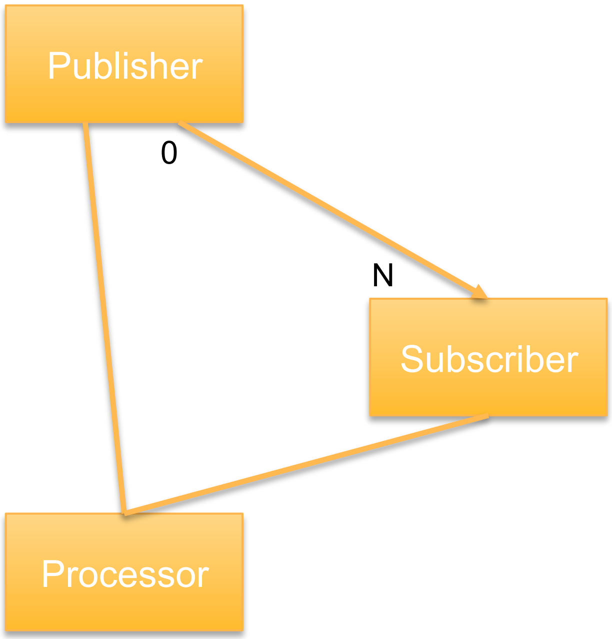
- Byte streams are not sufficient in JAX-RS
  - We want POJOs!
- What about JAX-RS readers, writers and interceptors?
- Poor integration with other third-party libraries and APIs

Current Proposal: Flows

- Standardized in Java 9
  - Originaly appeared in Reactive Streams
- Not just bytes, POJOs too
- Possibility of integration with third-party libraries and APIs

Simple Concepts





Why and when to NIO?

- Non-blocking code tends to be more involved
- Beneficial for large payloads
    - Large payloads often involve *collections*
    - In Flow terms, a collection of Pojos is a `Publisher<Pojo>`

Processing a collection

Asynchronous by nature

```
@POST
@Consumes(“application/json”)
void process(Publisher<Pojo> pojos,
             @Suspended AsyncResponse response) {
    pojos.subscribe(new Subscriber<Pojo> {
        public onNext(Pojo pojo) {
            // process single pojo
        }
        public onComplete() {
            return response.ok().build();
        }
        …
    } }
```

MessageBodyReader for Pojo?

- MessageBodyReader for Pojo or Publisher<Pojo>?
- We need a new type of Reader: NioBodyReader
  - Knows how to process a collection of Pojos
  - May use MessageBodyReader for each Pojo
- *(Ditto for MessageBodyWriters …)*

Pojo Processor in REST

```
@POST
@Consumes(“application/json”)
@Produces(“application/json”)
Publisher<Pojo> process(Publisher<Pojo> pojos,
                  @Suspended AsyncResponse response) {
    Processor<Pojo> pp = new MyProcessor(response);
    pojos.subscribe(pp);
    return pp;
}
```

Processors are publishers!

Third-party Libraries

```
@GET
@Produces(“application/json”)
Publisher<Pojo> process(
        @QueryParam(“predicate”) String predicate) {
    return DB.get(Pojo.class).filter(predicate);
}
```

Third-party Library

What about Filters and Interceptors?

- Need special support for NIO
  - For example, additional methods and contexts
  - Discussion still ongoing
- May impact how NioBodyReader/NioBodyWriter are defined

And NIO Clients?

- Will be supported as well
- Need re-usability of readers and writers for NIO
- Likely using a new client Invoker

NIO Client

```
WebTarget resource = target("…").path("pojos");
Publisher<Pojo> pojos =
    resource.request().nio().get(Pojo.class);

pojos.subscribe(new Subscriber<Pojo> {
    public onNext(Pojo pojo) {
        // process single pojo
    }
    …
});
```

NioBodyReader

Tentative Vocabulary

- Publisher = Source
- Subscriber = Sink
- *Does this ring a bell?*

SSE is a special case of Flow!

- Where what flows are SSE protocol messages
- Initial SSE proposal may be impacted
  - Return a Source instead of injecting a Sink

Non-Blocking IO Summary

- Based on Flows
- New readers/writers for collections
- Support for Flows without depending on Java 9
- Support for Pojos, not just byte streams
  - `Publisher<Pojo>` and `Publisher<ByteArray>`

Public Release In Progress

- Support for NIO and Flows
- Some minor changes to SSE possible
- Stay tuned!