SEBASTIAN DASCHNER

JAX-RS.next: A First Glance

A look at what's coming next in JAX-RS 2.1

JAX-RS is the standard for developing RESTful HTTP web services in an enterprise application. It offers a productive yet powerful way of defining REST resources by annotating plain old Java objects (POJOs) that contain the business logic without needing to write the HTTP plumbing by hand. Due to the fact that RESTful web services—or HTTP services implementing some of the REST constraints—are commonly used in enterprise projects and also because of its productive programming model, the JAX-RS standard is widely adopted among Java developers.

Java EE 7 comes with JAX-RS 2.0 (JSR 339), whereas Java EE 8, which is currently in progress, will contain JAX-RS 2.1 (JSR 370).

This article explains some of the new concepts and features the specification's update will deliver. These include support for server-sent events (SSEs), better integration with other Java EE specifications, and integration of reactive programming functionality, as well as non-blocking I/O on the client side. To follow along, you need some experience with JAX-RS.

Note: The specification and all the topics and examples this article covers are still subject to change. This article, then, is a preliminary explanation of what very likely will be included in the next JAX-RS version.

Server-Sent Events

SSEs are a new technology that is part of HTML5. SSEs automatically receive updates from a server via HTTP, and they offer an easy-to-use, one-way streaming communication protocol that perfectly fits use cases that have broadcast updates—for example, social media updates, stock prices, or news feeds.

The server pushes UTF-8 text-based data as content type `text/event-stream` to a client that previously connected to the streaming endpoint and, therefore, registered for the events. The format of the events looks as follows:

```
data: This is a message

event: namedmessage
data: This message has an event name

id: 10
data: This message has an id which will be sent as
 'last event ID' if the client reconnects
```

This approach of asynchronous events over an established connection competes with the more powerful WebSockets standard, which supports bidirectional communication. The main advantages of SSEs, however, are that they are compliant with HTTP technology, because they use HTTP as the communication protocol (which makes it easy to integrate them into existing networks or developer tools), and they natively support event IDs and reconnects.

JAX-RS 2.1 will support SSEs for both JAX-RS resources and the clients.

The server side of SSEs. The following code shows an SSE-enabled JAX-RS endpoint.

```
@Path("events-examples")
@Singleton
public class EventsResource {

    @Inject
    SseContext sseContext;

    private SseBroadcaster sseBroadcaster;
    private int lastEventId;
    private List<String> messages = new ArrayList<>();

    // see methods initSse(), itemEvents(), newMessage()
    // and createEvent() below
}
```

The `EventsResource` class is a singleton Enterprise JavaBean (EJB). It contains the `SseBroadcaster`, which is connected to all clients. `SseContext` is an injectable context object responsible for creating SSE broadcasters, outputs, and events. The `messages` represent the history of all created messages. The `sseBroadcaster` is initialized and enhanced with optional `SseBroadcaster.Listeners` in the `@PostConstruct` method.

```
@PostConstruct
public void initSse() {
    sseBroadcaster = sseContext.newBroadcaster();

    sseBroadcaster.register(
     new SseBroadcaster.Listener() {
        @Override
        public void onException(SseEventOutput output,
                Exception exception) {
            // ...
        }

        @Override
        public void onClose(SseEventOutput output) {
            // ...
        }
    });
}
```

The clients connect against the following JAX-RS resource:

```
@GET
@Produces(MediaType.SERVER_SENT_EVENTS)
@Lock(LockType.READ)
public SseEventOutput itemEvents(
        @HeaderParam(HttpHeaders.LAST_EVENT_ID_HEADER)
        @DefaultValue("-1") int lastEventId) {
    final SseEventOutput eventOutput =
            sseContext.newOutput();

    if (!sseBroadcaster.register(eventOutput)) {
        // try to make the client reconnect after
        //     5 seconds
        throw new ServiceUnavailableException(5L);
    }

    if (lastEventId >= 0) {
        // replay messages
        try {
            for (int i = lastEventId;
                    i < messages.size(); i++) {
                eventOutput.write(createEvent(
                        messages.get(i), i + 1));
            }
        } catch (IOException e) {
            throw new InternalServerErrorException(
                "Could not replay messages ", e);
        }
    }
```

```
    return eventOutput;
}

private OutboundSseEvent createEvent(String message,
        int id) {
    return sseContext.newEvent()
            .id(String.valueOf(id))
            .data(message)
            .build();
}
```

The SseEventOutput return type tells the JAX-RS implementation to keep the client connection open and capable of sending events through the broadcaster. Content type text/event-stream is used for SSEs. The SSE standard specifies that the Last-Event-ID header controls the event stream of the previously received events; that is, if SSE is supported by the server, the server resends the events created after lastEventId.

The method registers the newly created output to the sseBroadcaster and then immediately resends the events if the corresponding HTTP header has been sent. After the output is registered to the broadcaster, the client—together with all other active clients—receives events, which are created using the SseContext injected to the resource class.

This process is shown in the following code:

```
@Schedule(second = "*/5", minute = "*", hour = "*",
        persistent = false)
@Lock(LockType.WRITE)
public void newMessage() {
    final String message =
        "It's now: " + LocalDateTime.now();
    messages.add(message);

    final OutboundSseEvent event = createEvent(
        message, ++lastEventId);

    sseBroadcaster.broadcast(event);
}
```

Here, the @Schedule method is called every five seconds to create new events and broadcast them to all connected clients. I store the messages for later reuse in case any reconnecting client asks to continue the event stream from the last received message.

The client side of SSEs. JAX-RS 2.1 will also include client-side functionality to consume SSEs. There are two ways of creating a connection to an endpoint.

SseEventSource offers functionality to open a connection to an SSE endpoint by registering an event listener and, thus, providing a reactive way of handling the events.

```
public class SseClient {

    private final WebTarget target =
            ClientBuilder.newClient().target("...");
    private SseEventSource eventSource;

    public void connect(Consumer<String> dataConsumer){
        eventSource = SseEventSource.target(target)
            .register(ev ->
               dataConsumer.accept(ev.readData()))
            .open();
    }

    public void disconnect() {
        if (eventSource != null)
            eventSource.close();
    }
}
```

The SseEventSource is created by calling the target method on a WebTarget, registering an SseEventSource.Listener, and opening the connection. After successfully opening the connection, the current thread continues and the listener—which in this case would be dataConsumer .accept—is called as soon as events arrive.

Consumers can call start() and stop() **to resume and pause the stream,** and all events that happen during the pause are replayed after resuming.

SseEventSource handles all required plumbing, including reconnecting after a connection loss, by sending an appropriate Last-Event-ID header and then handling Retry-After headers sent from the server appropriately.

If a more sophisticated way is needed, for instance, to manually control the Last-Event-ID header, you can manually request an SseEventInput from the server.

```
public class StatefulSseClient {

    private final WebTarget target =
            ClientBuilder.newClient().target("...");
    private final Consumer<String> dataConsumer;
    private String lastEventId;
    private SseEventInput eventInput;

    public StatefulSseClient(
            Consumer<String> dataConsumer) {
        this.dataConsumer = dataConsumer;
    }

    public void start() {
        eventInput = target
          .request(MediaType.SERVER_SENT_EVENTS)
          .header(HttpHeaders.LAST_EVENT_ID_HEADER,
                  lastEventId)
          .get(SseEventInput.class);

        new Thread(() -> {
            while (!eventInput.isClosed()) {
                final InboundSseEvent event =
                        eventInput.read();
                if (event != null) {
                    lastEventId = event.getId();
                    dataConsumer.accept(
                        event.readData());
                }
            }
        }).start();
    }

    public void stop() {
        if (eventInput != null &&
            !eventInput.isClosed())
            try {
                eventInput.close();
            } catch (IOException e) {
                // suppress
            }
    }
}
```

By requesting the input directly, all required information has to be sent manually and potential reconnects need to be handled manually. Therefore, the client sends the text/event-stream HTTP header and expects the response to be an SseEventInput type that is handled appropriately by the JAX-RS implementation. This event input is used to receive the actual incoming events.

Because the eventInput.read method blocks until some

event is being sent, you need to manually take care of threading. Therefore, you start a new thread that waits for the events and, thus, the execution in the `start` method can continue.

Consumers of the `StatefulSseClient` class can call `start()` and `stop()` to resume and pause the stream, and all events that happen during the pause are replayed after resuming.

Reactive Programming

On the client side, another interesting new feature will be the integration of reactive computation types, such as `CompletionStage`, into the JAX-RS client.

As an example, I'll call several web services and combine the results with functionality that is available in JAX-RS 2.0 already.

Suppose I have a slow HTTP endpoint returning a randomly created response after a while, and suppose I want to consume this service several times and calculate a result from all responses. Polling the services sequentially and then constructing the result would lead to long wait times while the threads are blocked. To avoid that, I can use constructs such as `CompletableFuture`s together with Java 8 lambdas and streams.

Here's a JAX-RS 2.0 client-side example:

```
final Executor executor =
    Executors.newCachedThreadPool();

// endpoint takes 2 seconds to respond
//    with data like {"random":1}

final OptionalDouble average =
   IntStream
     .range(0, 10).parallel()
     .map(i ->
        CompletableFuture.supplyAsync(() ->
          target
            .request(MediaType.APPLICATION_JSON_TYPE)
            .get(JsonObject.class), executor)
            .thenApply(o -> o.getInt("random"))
            .join()
        ).average();

System.out.println(
    "average = " + average.getAsDouble());
```

Calling the endpoint via the `target` results in a 2-second wait time. To avoid this, I wrap the action in a `CompletableFuture` and extract the `total` value from the JSON data once the result is available. The `executor` is provided as a parameter so you can manage the threads for the 10 concurrent polls while most of the calling threads are waiting. The execution blocks when `join()` is called, but because I'm running the `IntStream` in parallel, all the results are available at roughly the same time—after about 2 seconds, rather than after 20 seconds, as they would be when polling one after the other. (The stream of results is then aggregated into the average value.)

This is what is possible today using JAX-RS 2.0 and `CompletableFuture` to wrap long-running jobs.

JAX-RS 2.1 will integrate reactive invokers for the client side. Constructing invocations is enhanced with `rx()` methods that wrap responses into a reactive invocation type and, optionally, use a specific `ExecutorService`. Here's an example:

```
final ExecutorService executor =
        Executors.newCachedThreadPool();

final OptionalDouble average = IntStream
        .range(0, 10).parallel()
        .map(i -> target
           .request(MediaType.APPLICATION_JSON_TYPE)
```

```
            .rx(executor)
            .get(JsonObject.class)
            .thenApply(o -> o.getInt("total"))
            .toCompletableFuture().join()
        ).average();

System.out.println("average = " +
        average.getAsDouble());
```

Calling the `rx()` method configures the builder to use an `RxInvoker`, and the `CompletionStage` is used as the default invocation type. Therefore, all subsequent actions, such as `get()`, return the responses wrapped in this corresponding reactive type. As in the previous example, I chain several actions together and `CompletionStage.toCompletableFuture` enables me to call `join()` as well.

This is one of the many useful improvements in the next release of JAX-RS.

Non-Blocking I/O

Another extension to both the client and the server side will be support for non-blocking I/O (NIO). By using NIO, the caller is guaranteed to be able to call certain methods for either sending data without being blocked or registering a reader that will eventually be called when data is available.

Server-side NIO. The JAX-RS server-side programming model will be enhanced so that developers can register `NioReaderHandler`s for responses or `NioWriterHandler`s for requests, respectively. These callback handlers are called when data needs to be read or written without blocking.

The following examples show how to read and write data in a non-blocking way in a JAX-RS resource:

```
@GET
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public Response download() {
    final InputStream in = // ...
    final byte[] buffer = new byte[1000];

    return Response.ok().entity(
      out -> {
            try {
                final int length = in.read(buffer);
                if (length >= 0) {
                    out.write(buffer, 0, length);
                    return true;
                }
                in.close();
                return false;
            } catch (IOException e) {
                throw new WebApplicationException(e);
            }
            }).build();
}
```

The `Response.ok().entity` method takes an `NioWriterHandler` as an argument and, thus, the lambda implementing the `NioWriterHandler.write(NioOutputStream)` method is called each time `out` is ready to accept data. The handler method is expected to return a boolean value indicating whether there is more data to write.

A similar handler approach is realized for reading uploaded data from the client's request:

```
@POST
@Consumes(MediaType.APPLICATION_OCTET_STREAM)
public void upload(@Context Request request) {
    final ByteArrayOutputStream out = // ...
    final byte[] buffer = new byte[1000];

    request.entity(
        in -> {
```

```
            try {
                if (in.isFinished()) {
                    out.close();
                } else {
                    final int length =
                        in.read(buffer);
                    out.write(
                        buffer, 0, length);
                }
                } catch (IOException e) {
                    throw new
                       WebApplicationException(e);
                }
        });
    }
```

The `Request.entity` method accepts an `NioReaderHandler.read(NioInputStream)` handler and, optionally, an `NioCompletionHandler` or `NioErrorHandler`, respectively, as an additional listener. Calling `isFinished()` on the input stream reveals whether more data is available.

By using these approaches, you can realize NIO by specifying asynchronous handlers for the data sent with the request or response.

Client-side NIO. Similar to the reactive integration in the client that was shown earlier, the NIO feature is realized as a separate invoker, callable by `nio()` on the `Invocation.Builder`, which returns an `NioInvoker` that is capable of handling non-blocking requests by `NioWriterHandler` or `NioReaderHandler` callbacks.

The following examples write and read data from the client side in a non-blocking way.

```
final InputStream in = // ...
final byte[] buffer = new byte[1000];

target.request().accept(
    MediaType.APPLICATION_OCTET_STREAM).nio().get(
        in -> {
            try {
                if (in.isFinished()) {
                    out.close();
                    // processing the output further...
                } else {
                    final int length = in.read(buffer);
                    out.write(buffer, 0, length);
                }
            } catch (IOException e) {
                throw new WebApplicationException(e);
            }
        });
```

```
target.request(
    MediaType.APPLICATION_OCTET_STREAM).nio().post(
        out -> {
            try {
                final int length = in.read(buffer);
                if (length >= 0) {
                    out.write(buffer, 0, length);
                    return true;
                }
                in.close();
                return false;
            } catch (IOException e) {
                throw new WebApplicationException(e);
            }
        });
final OutputStream out = // ...
final byte[] buffer = new byte[1000];
```

As on the server side, these builders also optionally accept an `NioErrorHandler`.

The JAX-RS implementation for both the client and the server takes care of the internal processing of the NIO by using `java.nio.*` features so that you don't need to worry about scheduling or handling the asynchronicity.

Version 2.1 of JAX-RS will ship with new features in Java EE 8.

Integration with Other Specifications

Besides the previously mentioned features that will affect the JAX-RS programming model, there will also be a few improvements in how the rest of the specifications work together within Java EE.

Because Java EE 8 will be the first version to have native support for JSON binding via JSON-B, it is crucial to be able to use that specification seamlessly with JAX-RS. As of today, binding XML to POJOs can be realized via Java Architecture for XML Binding (JAXB) mapping—most likely with a declarative approach using annotations. It works equally well to read from and write to POJOs that are mapped to JSON with JAX-RS using the JSON-B provider:

```
@Path("{id}")
@GET
public Example getExample(
    @PathParam("id") long id) {
    // ...
}

@POST
public void createExample(Example example) {
    // ...
}
public class Example {

    @JsonbTransient
    private long id;

    @JsonbProperty("hello")
    private String greeting;

    // getters & setters omitted
}
```

Another possible improvement is how other specifications that work with JAX-RS tackle Contexts and Dependency Injection (CDI), or JSR 330. Objects managed by the JAX-RS implementation are injected into the resource classes and methods mainly via `@Context` and field `Param`s.

Conclusion

As of today, JAX-RS 2.0 is a very usable and widely adopted specification in Java EE. Version 2.1, which will ship with Java EE 8, will be improved by the small enhancements and new features I've described here. Other changes that are already in the API's snapshot version can be found in the repositories on GitHub. `</article>`

Sebastian Daschner (@DaschnerS) is a Java EE freelancer based in Munich, Germany. He has more than six years of Java experience, contributes to various open source projects, is a Java Champion, and participates in the JCP as a JSR 370 Expert Group member. Daschner evangelizes computer science practices on his blog. When not working with Java, he loves to travel the world by plane or motorbike.

learn more

JSR 370 web page