JAX-RS 2.1 in Action

Markus Karg, Java User Group Pforzheim

Es hat lange gedauert, bis Oracle JAX-RS 2.1 endlich fertig hatte. Doch der Schein trügt: Was rein deklarativ als einfaches Minor-Release daherkommt, wird Microservice-Autoren freudig stimmen.

```
mvn archetype:generate -DarchetypeArtifactId=jersey-quickstart-grizzly2 -DarchetypeGroupId=org.glassfish.jersey.
archetypes -DinteractiveMode=false -DgroupId=com.me -DartifactId=demo -Dpackage=com.me -DarchetypeVersion=2.26-b09
```

Listing 1

Ja, es hat wirklich sehr, sehr lange gedauert, bis JAX-RS 2.1 endlich fertig war, und ja, es ist nur die Hälfte der angesagten Themen drin. Aber nein, das macht gar nichts, denn was da nun tatsächlich an Features auf dem Tisch liegt, kann sich aufgrund des effektiven Nutzwerts wirklich sehen lassen. Es hat sich ja in den letzten Jahren architektonisch viel getan in der Software-Welt, gerade „Microservices" und das „Internet of Things" sind Themen, die im Cloud-Umfeld allgegenwärtig im Raum stehen – auch bei Java-Projekten. Die neuen Features von JAX-RS 2.1 gestalten solche Projekte wesentlich effizienter gegenüber dem originalen Release 2.0. Am einfachsten lässt sich dies mit einem kleinen Beispielprogramm demonstrieren.

Stellen wir uns vor, wir hätten ein Sensor-Netzwerk, beispielsweise zur Messung von Temperaturdaten. So etwas kennt man von den beliebten Wetterseiten im Internet. Natürlich können wir im Rahmen dieses Artikels nicht dem Deutschen Wetterdienst Paroli bieten, uns jedoch rein als Gedankenmodell vorstellen, wir möchten einen Service erstellen, der unsere Sensordaten per REST aggregiert zur Verfügung stellt und beispielsweise den Mittelwert aller Sensoren einmal pro Sekunde aktualisiert an die Clients sendet. Ja, sendet, denn mit JAX-RS 2.1 können wir nun endlich „pushen".

Fertig … los!

Fangen wir ganz langsam an, in „old school" JAX-RS 2.0 mit Java 7. Damit es nicht ganz so viel Hackerei wird, benutzen wir Maven sowie den Archetyp der JAX-RS-Referenz-Implementierung Jersey *(siehe „https://jersey.github.io")* und verzichten ansonsten ganz bewusst auf IDEs, Plug-ins, Wizards und den ganzen anderen Kram, der wenig bringt, aber schick aussieht. Tatsächlich entstand der Code in *Listing 1* komplett in Bash mit dem vi-Editor und der Autor hat die IDE kaum vermisst.

Nach wenigen Sekunden befindet sich auf unserer Platte (hat überhaupt noch jemand eine echte Platte?) im Ordner „demo/" das lauffähige Grundgerüst einer richtigen (wenngleich sinnentleerten) JAX-RS-2.- Anwendung, die sich per „mvn compile exec:java" erstellen und starten lässt und der wir mittels cURL die ersten (gleichfalls sinnlosen) Daten entlocken können: „curl http://localhost:8080/myapp/myresource". Diese beiden Befehlszeichen (für Maven und cURL) bitte gut merken. Wir brauchen sie nun laufend und es wäre sicherlich mühselig, sie ständig zu wiederholen.

Übrigens, wer sich fragt, wie man denn auf diese URL kommt: Dazu muss man keineswegs in den Quellcode schauen. Beim Start schreibt die Anwendung netterweise die fix programmierte Root-URL auf „STDOUT"; ein „GET" auf diese, etwa mit cURL, offenbart das gesuchte Geheimnis in Form von WADL – und einiges mehr. WADL ist zwar leider nicht Teil von JAX-RS, was seine diesbezügliche Nützlichkeit allerdings keineswegs mindert.

Im Folgenden nun wollen wir dieser Anwendung zunächst einen (zugegebenermaßen minimalen) Sinn einhauchen, um daraufhin Schritt für Schritt das Ganze mit den neuen Fähigkeiten von JAX-RS 2.1 anzureichern. Der Sinn, wie gesagt, soll sein, Sensoren abzufragen. Zwar haben wir keine echten Sensoren zur Hand, doch wir können diese mit einem kurzen Stück Java-Code simulieren *(siehe Listing 2)*. Auch die automatisch generierte Ressourcen-Klasse bauen wir kurz und schmerzlos um, sodass unnötiger Ballast wie Kommentare und „@Produces"-Annotationen durch die sinnvolle Abfrage des Sensors ersetzt werden *(siehe Listing 3)*. Das ist sicherlich kein Meisterwerk, aber es erfüllt seinen Zweck: Nach dem erneuten Kompilie-

```
@XmlRootElement public class SensorValue {
  public final long time = System.currentTimeMillis();
  public final double value =  100 * Math.random();

  @Override public String toString() {
     return "Time: " + time + " Value: " + value;
  }
}
```

Listing 2

```
@Path("myresource") public class MyResource {
    @GET public SensorValue getSensorValue() {
        return readSensor();
    }

    private static SensorValue readSensor() {
        return new SensorValue();
    }

}
```

Listing 3

```
@GET public Optional<SensorValue> getSensorValue() {
    return readSensor();
}

private static Optional<SensorValue> readSensor() {
    return Optional.of(new SensorValue());
}
```

Listing 4

```
public class InstantXmlAdapter extends
XmlAdapter<String, Instant> {
    public String marshal(Instant v) {
        return Optional.ofNullable(v).
map(Instant::toString).orElse(null);
    }

    public Instant unmarshal(String v) {
        return Optional.ofNullable(v).
map(Instant::parse).orElse(null);
    }
}
```

Listing 5

```
@Provider public class OptionalWriterInterceptor implements WriterInterceptor {
  public void aroundWriteTo(WriterInterceptorContext context) {
     if (context.getType().equals(Optional.class)) {
        Optional<?> optional = (Optional<?>) context.getEntity();
        Class<?> type = (Class) ((ParameterizedType) context.getGenericType()).getActualTypeArguments()[0];
        context.setEntity(optional.orElse(null));
        context.setType(type);
        context.setGenericType(type);
     }
     context.proceed();
  }
}
```

Listing 6

ren und Starten bestätigt cURL uns den Empfang eines kurzen XML-Dokuments mit Timestamp und zufällig simuliertem Sensorwert.

Java 8 ohne Klimmzüge

Was jedoch ärgert, ist der Typ „long" für den Timestamp. Seit Java 8 gibt es dafür einen vernünftigen Daten-Typ („Instant"), und den werden wir verwenden. Dann existieren noch Sensoren, die zwar immer mit einem reden, aber nicht immer einen sinnvollen Wert liefern. So etwas hätten wir in obiger Anwendung per „null" oder mit einer Exception umgesetzt. Da wir den Fehlercode nicht brauchen und da „null" immer die Angst einer „NullPointerException" mit sich bringt, wäre es schöner, hier die Java-8-Klasse „Optional" nutzen zu können.

Wir brauchen also Java 8, und das geht so: Zunächst ersetzen wir im POM kurzerhand den Wert „1.7" durch „1.8", damit Maven weiß, was wir haben wollen. Danach bauen wir den Sensor um auf Instant („public final Instant time = Instant.now();") und die Ressource auf „Optional" *(siehe Listing 4)*. Damit weiterhin XML herauskommt, brauchen wir einen JAXB-Adapter, das war schon in JAX-RS 2.0 so *(siehe Listing 5)*.

Bis hier dürfte das Ganze den meisten JAX-RS-Erfahrenen geläufig sein. Experten wissen natürlich, dass die Einführung von „Optional" ein paar mehr Tricks erfordert: Dazu brauchen wir einen „WriterInterceptor", sonst weiß kein existierender „MessageBodyWriter", was er tun soll – und wir müssten praktisch alle Zieldaten-Formate neu programmieren, einmal mit und einmal ohne „Optional". Da dieses Unterfangen höchst uneffektiv wäre, rüsten wir kurzerhand den Support für „Optional" nach, wodurch alle „MessageBodyWriter" dieser Welt bleiben können, wie sie sind *(siehe Listing 6)*.

Schon haben wir sowohl eine lesbare Uhrzeit als auch die Unterstützung für optionale Rückgabewerte aus Java 8. Der Trick an JAX-RS 2.1 ist also nicht, dass alle möglichen neuen Klassen direkt eingebaut sind (das sind nur wenige), sondern, in Analogie zu anderen erweiterbaren Frameworks wie JAXB, JSON-B und JPA, dass die in JSR 370 erarbeitete Spezifikation erzwingt, dass JRE 8 als Unterbau vorhanden sein muss – denn nur so kann sich die Anwendung darauf verlassen, die neuen JRE-8-Klassen zu finden. Ansonsten würde schließlich das Laden der Ressourcen-Klasse ebenso fehlschlagen wie das Laden des Interceptor oder des JABX-Adapters.

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-binding</artifactId>
</dependency>
```

Listing 7

```
curl -H Accept:application/json http://localhost:8080/
myapp/myresource
{"time":"2017-07-09T15:19:23.288Z","val
ue":57.3497420050259}
```

Listing 8

Good bye, JAXB!

Apropos JAXB-Adapter: Falls es noch nicht bekannt sein sollte, JAXB wird zukünftig nicht mehr automatisch im JRE enthalten sein. Zwar ist es in Java 9 noch da, doch dank des Projekts „Jigsaw" ist es optional – man muss es also extra einschalten. Ob dies in Java 10 noch der Fall sein wird, ist fraglich, denn offiziell ist JAXB ab Java 9 als „deprecated for removal" markiert. Es ist also damit zu rechnen, dass es in Java 10 oder Java 11 verschwunden ist. Die Lösungsstrategie darf sich jeder selbst aussuchen, hier einige zur Auswahl:

- In Java 9 per „--add-modules java.xml.bind" schlicht JAXB wieder an der Kommandozeile einschalten oder die eigene Anwendung mit einer entsprechenden Klausel in „module-info.java" ausstat-

```
@GET @Produces(MediaType.SERVER_SENT_EVENTS)
public void getSensorValue(@Context SseEventSink sseEventSink, @Context Sse sse, @HeaderParam("X-Accept") MediaType
mediaType) {
  timer.schedule(
    () -> sseEventSink.send(sse.newEventBuilder().mediaType(mediaType).data(readSensor().get()).build()).thenRu-
nAsync(() -> getSensorValue(sseEventSink, sse, mediaType)),
    1, TimeUnit.SECONDS
  );
}
```

Listing 9

ten. Annahme: Java 9 wird diese Möglichkeiten in der vorliegenden Form im Final Release enthalten, wonach es bis Redaktionsschluss aussieht.
- In Java 10 als eigenständige Bibliothek der Anwendung hinzufügen wie jede andere Third-Party-Bibliothek auch. Annahme: Oracle führt JAXB, derzeit zu finden auf https://github.com/javaee/jaxb-v2, als selbstständiges Projekt weiter oder übergibt es, ähnlich wie im Falle des MVC-API, einem neuen Maintainer. Nach der kürzlichen Oracle-Ankündigung zur Öffnung von Java EE ist davon auszugehen.
- Oder einfach zugunsten von modernen Formaten wie JSON auf XML verzichten.

Die Cloud spricht JSON

Stichwort JSON: Neben vielen kleineren Änderungen (wie beispielsweise dem Verzicht auf die ohnehin seit JAX-RS 2.0 sinnlose Methode „getSize" in „MessageBodyWritern") bringt JAX-RS 2.1 auch ein paar tolle, größere Sachen mit. Gerade im Hinblick auf Microservices und Cloud-Anwendungen muss an dieser Stelle natürlich erwähnt werden, wie leicht es für den Anwendungsprogrammierer wird, von XML auf JSON – die „Lingua Franca" der Cloud-Welt – umzustellen oder es parallel zu nutzen, denn XML weiter zu unterstützen, erfordert keine Ressourcen. JSON bekommt man nun quasi geschenkt. Derzeit muss es in der Referenz-Implementierung allerdings noch manuell eingeschaltet werden, was über simples Auskommentieren der entsprechenden Dependency im POM geschieht *(siehe Listing 7)*. Das war's! *Listing 8* zeigt den kurzen Test.

Augenscheinlich erfolgreich! Theoretisch könnte man nun also an dieser Stelle XML ausbauen. Die einzige Stelle, an der XML explizit in unserer Anwendung genannt ist, ist die Annotation „@XmlRootElement" über dem SensorValue-POJO. Wird diese entfernt, ist XML effektiv stillgelegt.

Push it!

Das nächste größere Schmankerl ist die Unterstützung für Server-Sent Events (SSE, nicht zu verwechseln mit WebSockets, Comet oder Long Polling). Diese Technologie erweitert reines HTTP um die Möglichkeit, nicht nur eine, sondern beliebig viele Responses pro Request zu senden – und zwar zu einem beliebigen, späteren Zeitpunkt. Die Push-Möglichkeit ist gerade im IoT-Umfeld sinnvoll, etwa um unnötige Network-Roundtrips zu vermeiden.

Nicht verschwiegen werden soll, dass auch JAX-RS nichts daran ändern kann, dass SSE durch die Hintertür Sessions wieder einführt, die man ja in REST-Anwendungen gerade nicht möchte. Wen dies jedoch nicht weiter stört, der bekommt nun das Handwerkszeug in Form der injizierten Ereignisrutsche „SseEventSink" und der zugehörigen Event-Factory „Sse" geliefert – muss sich dann prinzipiell aber um alles Weitere selbst kümmern: JAX-RS-Ressourcen-Methoden müssen „void" als Ergebnis-Typ deklarieren und alle Responses, auch die initiale, selbst produzieren. Ebenso muss sich die Anwendung selbst darum kümmern, wann eine Response geschickt wird, etwa mittels einer Schleife, eines Timers oder wenn ein anderer Client Daten hochgeladen hat *(siehe Listing 9)*. Hierzu gibt es ein Broadcast-API, das in unserem Beispiel aber nicht vorkommt. Um das auszuprobieren, muss allerdings bei Nutzung von Jersey der SSE-Support im POM eingeschaltet sein *(siehe Listing 10)*.

Das Code-Beispiel macht augenscheinlich, dass für SSE einiges Beiwerk notwendig ist. Zum einen darf der initiale Thread, der die SSE-Session ursprünglich öffnet, nicht blockiert werden. Die Erzeugung der Responses erfolgt daher im Beispiel über einen „ScheduledExecutor". Auf eine initiale Response wird der Einfachheit wegen verzichtet, sie könnte aber auf technisch gleichem Wege erfolgen, wenn man den initialen Delay auf „0" setzt statt auf „1 Sekunde". Die Schleife wird im Beispiel über die Reaktion auf das Ende des Sendevorgangs gebildet, indem der „CompletionStage", die von „SseEventSink.send()" geliefert wird (und die per Hintergrund-Thread sendet), befohlen wird, einfach wieder die gleiche Methode („getSensorValue") aufzurufen.

Jeder einzelne Event, der dem Client gesendet werden soll, kann einen anderen MIME-Type haben. Möchte man HTTP-konform dem Client die Wahl über den MIME-Type per Accept-Header geben, steht einem die SSE-Spezifikation im Weg, die besagt, dass der initiale SSE-Session-Aufbau nur erfolgt, wenn der Client per „Accept:text/event-stream" dies fordert. Zwar können mehrere Accept-Header-Values im Request enthalten sein, dies jedoch nur, um dem Server die Auswahl über einen Typ zu lassen, was hier aber explizit ja eine SSE-Session sein soll. Damit steht der Accept-Header nicht mehr

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-sse</artifactId>
</dependency>
```

Listing 10

```
curl -N -H X-Accept:application/json http://localhost:8080/myapp/myresource
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:--  0:00:01 --:--:--
0data: {"time":"2017-07-09T17:58:05.303Z","value":23.254565347099575}

100    70    0    70    0     0     31      0 --:--:--  0:00:02 --:--:--
31data: {"time":"2017-07-09T17:58:06.307Z","value":88.88601882225326}

100   139    0   139    0     0     43      0 --:--:--  0:00:03 --:--:--
43data: {"time":"2017-07-09T17:58:07.310Z","value":17.351233704968173}

100   209    0   209    0     0     49      0 --:--:--  0:00:04 --:--:--
49data: {"time":"2017-07-09T17:58:08.315Z","value":23.409041205977122}
```

Listing 11

```
@GET public CompletionStage<SensorValue> getAver-
ageSensorValue() {
  CompletionStage<SensorValue> sensorA =
readSensor("A");
  CompletionStage<SensorValue> sensorB =
readSensor("B");
  return sensorA.thenCombine(sensorB,
SensorValue::average);
}
```

Listing 12

für den gewünschten MIME-Type der Events zur Verfügung. Unser Beispiel befreit sich aus diesem Dilemma, indem es HTTP-konform einen „X-Accept"-Header erfindet. Daher läuft es auch nur dann fehlerfrei, wenn cURL nun statt mit „-H Accept" mit „-H X-Accept" aufgerufen wird *(siehe Listing 11)*.

Der zusätzliche Parameter „-N" verbietet es cURL, die Ausgabe zu puffern und hat weder mit JAX-RS noch mit SSE zu tun: Damit wird lediglich erreicht, dass wir die eingetroffenen Responses sofort auf der Konsole sehen können und nicht warten müssen, bis der entsprechende STDOUT-Puffer voll ist.

Reaktivität

Temperatur-Sensoren sind nicht sonderlich flott: Je nach Modell kann die Abfrage schon eine Weile dauern. Dies fällt beim Pushen nicht auf, da die empfangende Anwendung ja letztendlich gar nicht weiß, wann die Anfrage an den Sensor gestartet wurde. Wenn wir jedoch nicht nur einen einzigen Sensor abfragen, sondern beispielsweise den Mittelwert einer ganzen Sensoren-Batterie (etwa um den Schnitt einer größeren, überwachten Fläche zu erhalten), und wenn wir die Messung zudem explizit anstoßen, werden wir die ewige Wartezeit sicherlich ärgerlich finden. Zudem ist in so einem Fall von langsamer Query anzumerken, dass der Front-End-Thread (also der, der die Netzwerkkarte des Servers bedient) blockiert – und davon hat der Webserver leider nicht unbegrenzt viele. Entsprechend findet keine Kommunikation mehr statt, der Server fühlt sich überlastet an. Gut, ein Raspberry PI ist da jetzt sowieso anderweitig am Limit. Insbesondere in Cloud-Umgebungen stellt dies allerdings ein spürbares Skalierungshemmnis dar.

Um Front-End-Threads von Back-End-Threads zu entkoppeln, gab es schon in JAX-RS 2.0 die Möglichkeit der asynchronen Programmierung. Diese ist aber zugegebenermaßen recht umständlich. Mit dem reaktiven API von JAX-RS 2.1 wird dies sehr, sehr viel übersichtlicher, *Listing 12* ist zur Veranschaulichung auf zwei Sensoren reduziert.

Interessant dabei ist, dass „CompletionStage" nicht die einzige Klasse ist, mit der JAX-RS umzugehen weiß: Über diese Pflichtübung hinaus steht es jeder Implementierung frei, auch weitere reaktive Frameworks zu unterstützen, beispielsweise RxJava. Damit eine Anwendung jedoch portabel bleibt (also nicht nur mit Jersey, sondern beispielsweise auch mit CXF läuft), sollte man sich nicht auf diese spezielle Fähigkeit verlassen und lieber die entsprechende Bibliothek selbst mitbringen sowie das Ergebnis etwa von Observable auf CompletionStage umformen. Dazu gibt es im Web zahlreiche Beispiele. Freuen darf man sich bereits auf das nächste Release von JAX-RS: Dort wird es dann auch Unterstützung für das Flow-API geben, das mit Java SE 9 eingeführt wurde, womit SSE dann erst richtig Spaß macht.

Fazit

Mit JAX-RS 2.1 wurde wertvolle Detailarbeit am REST-API geleistet, das vor allem den Architekturwechsel von Application Server auf Microservices unterstützen soll. Mit der Konzentration auf die Aspekte Java 8, JSON, SSE und reaktive Programmierung folgt der neue Standard diesem Trend, ohne Vorhandenes über Bord zu werfen, das API unnötig aufzublasen oder die Portabilität zu opfern – Stichwort „WORA".

Wer Lust hat, kann das Ganze gerne mal auf einem Raspberry PI mit echter Wettersensorik ausprobieren. Tatsächlich würde es problemlos laufen, denn JAX-RS 2.1 ist leichtgewichtig genug, um nicht nur auf der Cloud-Seite, sondern auch auf der „Things"-Seite zu laufen, und der Raspberry PI ist zugegebenermaßen nicht gerade schwachbrüstig.

Markus Karg
markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.