HowTo: bestehende Anwendung auf Java 11 umstellen

Diese Anleitung soll euch durch die Aufrüstung eurer Anwendung auf Java 11 führen. Falls ihr auf neue Probleme stoßt, die hier noch nicht beschrieben sind, bitte einfach kommentieren oder bei JBS melden, damit wir unterstützen bzw. die Anleitung erweitern können.

erster Schritt: Runtime aufrüsten

Wenn ihr eure Anwendung umstellt, könnt ihr zunächst sehr einfach die Laufzeit umstellen. Dieser Schritt ist ziemlich unkritisch und erfordert nicht viel Arbeit. Dafür habt ihr schon einmal den Vorteil der optimierten Garbage Collection und Team PCS freut sich, weil auf Dauer das Java 8 Image entfallen kann.

Für eure lokale Laufzeit müsst ihr den OpenJDK 11 für eure JBoss Runtime wählen. Diese habt ihr ab dem Entwicklerarbeitsplatz 2.6.0 zur Verfügung.

Für die Laufzeit im Container müsst ihr lediglich diesen Block in eurer install.yml einfügen:

install.yml

```
java:
  runtime:
    version: '11'
```

Die Änderungen im Memory Management sind dann beispielsweise in Instana sehr gut sichtbar (hier zwei unterschiedliche Anwendungen):

Java 8

Memory Pools

Not all memory pools are considered to be part of the JVM heap. Usually only `Eden`, `Survivor` and `Old` are part of the heap. Depending on the configuration of the JVM it may resize any of these pools.

Pool ↑	Initial	Maximum	Value
Code Cache	2,44 MiB	240,00 MiB	51,04 MiB
Compressed Class Space	0,00 B	248,00 MiB	17,71 MiB
Metaspace	0,00 B	256,00 MiB	137,66 MiB
PS Eden Space	26,00 MiB	339,50 MiB	15,40 MiB
PS Old Gen	68,00 MiB	683,00 MiB	70,58 MiB
PS Survivor Space	4,00 MiB	1,00 MiB	336,58 kiB

Java 11

Memory Pools

Not all memory pools are considered to be part of the JVM heap. Usually only `Eden`, `Survivor` and `Old` are part of the heap. Depending on the configuration of the JVM it may resize any of these pools.

Pool ↑	Initial	Maximum	Value
CodeHeap 'non-nmethods'	2,44 MiB	5,56 MiB	1,51 MiB
CodeHeap 'non-profiled nmethods'	2,44 MiB	117,22 MiB	37,75 MiB
CodeHeap 'profiled nmethods'	2,44 MiB	117,22 MiB	8,39 MiB
Compressed Class Space	0,00 B	248,00 MiB	20,99 MiB
Metaspace	0,00 B	256,00 MiB	172,77 MiB
PS Eden Space	26,00 MiB	340,00 MiB	1,05 MiB
PS Old Gen	68,00 MiB	683,00 MiB	80,26 MiB
PS Survivor Space	4,00 MiB	512,00 kiB	96,00 kiB

zweiter Schritt: Compilezeit umstellen

Hier wirds spannend, denn mit der geänderten Compilezeit bekommt ihr die ganzen neuen Features zur Verfügung. Hier wartet aber mehr Arbeit und es gibt einige Einschränkungen:

- Bibliotheken, die von anderen konsumiert werden, dürfen erst auf Java 11 umgestellt werden, wenn alle Konsumenten bereits auf Java 11 umgestellt wurden. Eure base-Projekte müssen daher vermutlich noch etwas warten. Andernfalls erhalten diese **zur Laufzeit** eine vergleichbare Fehlermeldung:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError: de/gothaer/rvk/domain/exceptions
/VermittlerLesenException has been compiled by a more recent version of the Java Runtime (class
file version 55.0), this version of the Java Runtime only recognizes class file versions up to 52.0
```

- Nutzt ihr ODM, dann dürft ihr eure XOM-Projekte ebenfalls nicht auf Java 11 umstellen.
- Nutzt ihr JMockit für eure Tests, dann gibt es ein Problem mit der aktuellen Version des jacoco-maven-plugins aus der master-pom. Die Problematik ist hier beschrieben: https://github.com/jacoco/jacoco/issues/896. Leider gibt es keinen fix für JMockit. Allerdings scheint das Projekt auch tot, wenn man sich die Aktivität auf Github anschaut. Vielleicht ist jetzt ein guter Zeitpunkt zum Umstieg auf Mockito.

Für beide Ausnahmen gilt aber, dass ihr die Compilezeit auch nur in dedizierten Submodulen auf Java 8 runterdrehen könnt, während die anderen Module auf Java 11 umgestellt werden.

Gesteuert wird die Compilezeit eurer Projekte über die Property **java.compiler.version**. Die steuert das maven-toolchain-plugin aus der base-master-pom. Hier erfahrt ihr mehr dazu.

In eurer POM auf oberster Ebene setzt ihr zur Umstellung die Property auf 11, um den Java 11 Compiler auszuwählen:

```
<properties>
    <java.compiler.version>11</java.compiler.version>
</properties>
```

Falls ihr ein Submodul im Projekt habt, dass ihr noch nicht umstellen könnt (siehe oben), dann konfiguriert ihr in diesem die gleiche Property nur mit Wert 8.

Anschließend könnt ihr mit einem Maven-Install und einem Update-Projects prüfen, ob ihr auf Compilefehler lauft. Spätestens zur Laufzeit fallen aber fehlende Dependencies auf. Für zahlreiche der Service Clients (beispielsweise der ORG-Client) benötigt ihr Dependencies, die mit Java 11 nicht mehr standardmäßig ausgeliefert werden. Zu diesen Bibliotheken (je eine aus javax mit der Spezifikation und eine aus com.sun mit der Implementierung) gehören die nachfolgenden Beispiele, die ihr jetzt manuell mit einbinden müsst:

JAXB

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-core</artifactId>
    <version>2.3.0.1</version>
</dependency>
```

JAX-WS Runtime

```
<dependency>
    <groupId>javax.xml.ws</groupId>
    <artifactId>jaxws-api</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.ws</groupId>
    <artifactId>jaxws-rt</artifactId>
    <version>2.3.1</version>
</dependency>
```

Achtet beim Einbinden auf passende Versionen, andernfalls erhaltet ihr ggfs. zur Laufzeit Fehlermeldungen vom Classloader.

Falls ihr ein GWT Projekt aktualisiert müsst ihr im gwt-maven-plugin die Einstellung SourceLevel ergänzen und ggfs. das Maven-Plugin für den GWT-Build anpassen (siehe GWT - Builds mit Maven).

```
<sourceLevel>1.11</sourceLevel>
```

SOAP Services: WSDL- und Client-Generierung

Bei der Generierung der **WSDL** mit dem **cxf-java2ws-plugin** ist darauf zu achten, dass auch dabei die Java 11 Runtime genutzt werden muss. Bei der Verwendung von "Run as Maven build..." ist im **Reiter JRE** entsprechend die "Alternate JRE" auszuwählen.

Die **SOAP Clients** sind in der Regel Basisdienste, die erstmal auf Java 8 verbleiben (s.o.). Sollten diese so alt sein, dass sie noch dem überholten Verfahren folgen, dass per **JAXB-Binding-Declaration** die originalen Schnittstellen-Klassen oder gar fachlichen Modell-Klassen im Client verwendet werden müssen, dann ist darauf zu achten, dass die betrofffenen Module, in denen sich diese Klassen befinden, auch noch weiterhin mit Java 8 kompiliert werden müssen.

weiterführende Informationen

- Einen interssanten Artikel zur Umstellung und zur Toolunterstützung seitens Java 11 selbst findet man hier.