Des Kaisers neue Kleider

Migration nach Java 8

Thorsten Göckeler

Lambdas, das neue Date-Time-API und ein Nashorn bringen frischen Schwung in die Entwicklung mit Java. Wie schwierig ist aber die Migration zu Java 8? Wie gut unterstützen die IDEs mich als Entwickler jetzt, und wird die Anwendung gar langsamer? Sind die Build-Werkzeuge bereits vorbereitet, kann ich jetzt produktiv mit Java 8 sein? Das werden wir anhand der Spring-Anwendung Pet Clinic erproben.

Motivation

Die meisten Systeme entstehen nicht neu, sie sind bereits da. Im Laufe der Zeit werden sie erweitert und verbessert und laufen meist länger als erwartet. Manchmal werden sie auf eine neue oder eine andere technische Basis gehoben.

Jede Migration bietet ihre eigenen Überraschungen, mit denen der Aufwand ungeplant höher wird. Für den Umstieg auf Java 8 wollen wir Überraschungen vermeiden. Deshalb nehmen wir uns zur Übung der Spring Pet Clinic an, einer realen und für jeden zugänglichen Anwendung. Real deshalb, weil sie einige Eigenschaften aufweist, die bei professionell entwickelter Software typisch sind: Sie verwendet Spring, greift auf eine Datenbank mittels JDBC, JPA beziehungsweise Spring JPA zu, hat eine Weboberfläche, wird mit Maven gebaut und verfügt über ausreichend Tests. Ferner verwendet sie die JODA-Bibliothek. Zusätzlich ist sie gut dokumentiert.

Nebenbei soll natürlich geklärt werden, wie aufwendig beziehungsweise wie schnell wir als Entwickler eine wirkliche Anwendung auf Java 8 migrieren können und wie wir die neuen Möglichkeiten nutzen können, um den vorhandenen Code besser lesbar zu machen und eventuell auch schneller.

Vorbereitung

Die Schritte einer Migration sollten sich eigentlich immer ähneln, in der Praxis gibt es dann meistens doch einige Abweichungen. Die erfolgversprechendsten Schritte lauten:

- Lauffähige Software demonstrieren lassen.
- Denselben Stand aus den Quelldateien bauen und testweise installieren.
- Entwicklungsumgebung aufbauen bzw. konfigurieren.
- Eine Sache verändern und lauffähig bekommen.

Unter [Pet] existiert eine Demonstration der Anwendung. Von [Git] lässt sich der aktuelle Stand klonen und mit `mvn tomcat7:run` bauen und ausführen. Aus der Entwicklung sind wir auch sehr vertraut mit der folgenden Herangehensweise:

- Software kompilierbar bekommen.
- Auf einem System verteilen.
- Starten und die Fehler anschauen.
- Fehler beseitigen und mit dem zweiten Schritt fortfahren.

Dieses Vorgehen ist sehr pragmatisch und oft sogar erfolgreich. Manchmal ist es sogar die einzige Möglichkeit, besonders wenn es zu aufwendig ist, die jetzige Laufzeitumgebung nachzustellen. Allerdings können wir so kaum sicherstellen, dass diese Version funktional mit der bestehenden Anwendung identisch ist, es sei denn, die Anwendung verfügt über einen funktionierenden Regressionstest. In der Praxis trifft das eher selten zu. Wenn sich die Migration als etwas komplizierter herausstellt, tritt ein weiterer Nachteil in den Vordergrund: Es gibt keinen lauffähigen letzten Stand.

Versuchen wir nun, die Spring Pet Clinic mit Java 8 als Laufzeitumgebung zu starten, gelingt uns das nicht. In den gängigen Foren werden unterschiedliche Ursachen angegeben: Der angegebene Tomcat 7 kommt mit Java 8 nicht zurecht. Der verwendete Eclipse-Compiler enthält einen Bug. In der Datei `web.xml` müssen Änderungen gemacht werden.

Wenn wir nachvollziehen wollen, welche Änderung tatsächlich notwendig ist, dann ist spätestens jetzt der Zeitpunkt gekommen, um auf die zuerst aufgeführte Herangehensweise umzustellen. In diesem Fall ist das besonders einfach: Ein JDK 1.6 oder ein JDK 1.7 als Laufzeitumgebung verwenden – indem die Umgebungsvariable `JAVA_HOME` entsprechend gesetzt wird –, und schon steht die Anwendung unter `http://localhost:9966/petclinic/` zur Verfügung, siehe auch Abbildung 1.

Abb. 1: Spring Pet Clinic

Als letzten Schritt der Vorbereitung importieren wir die Anwendung in eine IDE unserer Wahl, zum Beispiel Eclipse (Kepler, SP2), IntelliJ IDEA (13+) oder Netbeans (8+). Ältere Versionen sind keine guten Voraussetzungen, um auf Java 8 zu wechseln.

Gleicher Code, neues Java

Die Anwendung läuft unter Java 6. Die Migration auf einen neuen Webcontainer verspricht viel Arbeit, aber ein Tomcat 7 läuft mindestens mit einem Java 7. Eine Umstellung im `pom.xml` später läuft die Anwendung einwandfrei auf Java 7, siehe Listing 1.

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.0</version>
    <configuration>
      <source>1.7</source>
```

```
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
```

Listing 1: Maven mit Java 7

Nachdem dieser Schritt erfolgreich war, wagen wir den nächsten Schritt und bleiben bei der Sprachspezifikation von Java 7, lassen die Anwendung aber unter Java 8 laufen. Und nun wartet die erste Überraschung auf uns: Das Tomcat-Maven-Plug-in startet einen Tomcat 7.0.30. Die Anwendung startet nicht, weil die JSP-Seiten nicht kompiliert werden können. Wenn wir die Anwendung aber manuell in einen neueren Tomcat 7 legen, zum Beispiel den aktuellen Tomcat 7.0.47, dann starten der Server und die Anwendung ohne weitere Probleme. Zurzeit gibt es noch kein neueres Plug-in, sodass wir dieses entweder ausbauen können oder vorerst ignorieren.

Eclipse

In den Entwicklungsumgebungen ist dies nicht weiter tragisch. In Eclipse zum Beispiel lässt sich unter „Preferences | Server | Runtime Environments" der externe Tomcat 7 konfigurieren und mittels WTP aktuell halten. In den anderen genannten Entwicklungsumgebungen ist es ähnlich, siehe weiter unten.

Bevor wir weitere Änderungen planen, sollten wir Eclipse für die Verwendung von Java 8 vorbereiten – ab Luna ist das nicht mehr notwendig, dann ist Java 8 genauso integriert wie in Netbeans und IDEA. Als Erstes erzwingen wir, dass Eclipse mit einem Java 8 als Laufzeitumgebung startet. Hierzu ändern wie die Datei `eclipse.ini` im entsprechenden Installationsverzeichnis ab, wie in Listing 2 fett markiert. Je nach Bedarf können wir auch gleich den Standard-Zeichensatz auf UTF-8 forcieren und den Speicher etwas erhöhen. Zum ersten Mal treffen wir hier auf Java 8: Die beliebten Einstellungen **`-XX:PermSize`** und **`-XX:MaxPermSize`** sind hinfällig, es gibt keinen PermGen Space mehr.

```
-startup
plugins/org.eclipse.equinox.launcher_1.3.0.v20130327-1440.jar
-vm
C:/Programme/Java/jdk1.8/bin/javaw.exe
...
-vmargs
-Dfile.encoding=UTF-8
-Dosgi.requiredJavaVersion=1.6
-Xms384m
-Xmx768m
```

Listing 2: eclipse.ini

Wenn wir unter „Preferences | Java | Java Installed JREs" das Java 8 JDK eingerichtet haben, benötigt Kepler noch eine Frischzellenkur für einige Plug-ins, damit die neuen Sprachmittel auch verstanden werden. Im Eclipse Marketplace finden sich schnell die Patches für Eclipse, m2e und WTP, siehe Abbildung 2.

Mit diesen Vorarbeiten können wir nun im `pom.xml`, ähnlich wie in Listing 1 bereits beschrieben, mittels der Versionskennung „1.8" auf Java 8 umstellen, lassen die Anwendung bauen und entweder über WTP oder direkt auf dem Tomcat 7 laufen. Bei den meisten Anwendungen dürften nun einige Stolpersteine zu erwarten sein, in diesem Fall aber nicht. Die Spring Pet Clinic ist auf Java 8 migriert und läuft einwandfrei auf dem Tomcat 7. Die Voraussetzungen waren allerdings optimal, denn es wurde bereits Spring 4.0.3 eingesetzt, die erste Version, die gegen Java 8 getestet wurde und diese Sprachversion unterstützt.

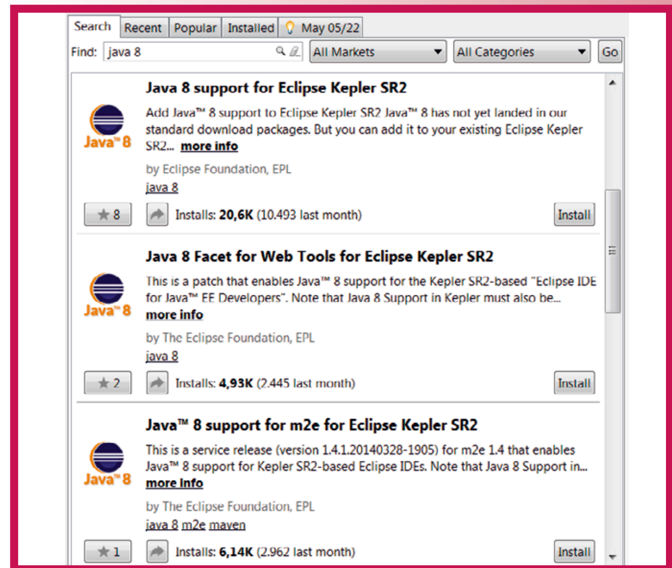

Abb. 2: Eclipse Java 8 Patches

IntelliJ IDEA, Netbeans

Die Integration von Maven und Java 8 in IDEA ist hervorragend. Entwicklungsumgebung aufmachen, Maven-Projekt importieren, fertig. Die gewünschten Java-Laufzeitumgebungen sollten vorher schon installiert sein, IDEA übernimmt automatisch alle Änderungen aus der `pom.xml`, auch wenn die in einem externen Texteditor erfolgen. Das gilt auch für Projekte, die mit Gradle gebaut werden. Einziger Wermutstropfen: Die Zusammenarbeit mit Anwendungsservern ist der Ultimate Edition vorbehalten. In der Community Edition ist deshalb Kopieren ohne Unterstützung der IDE notwendig.

Netbeans gibt sich ebenfalls unkomplizierter als Eclipse. Das Projekt lässt sich entweder direkt mittels „File | Open Project" öffnen oder von Eclipse importieren. Zusätzlich zu den bereits installierten Anwendungsservern lassen sich andere externe Server über „Tools | Servers" konfigurieren. Im Projekt selbst können wir dann über Rechtsklick, „Properties | Run" den zu verwendenden Server angeben und die Anwendung über „Run" ausführen.

Jenkins, SonarQube, Codequalität

Im professionellen Umfeld sind Werkzeuge zum kontinuierlichen Bauen der Software, die Verwaltung der entsprechenden Artefakte und die Überprüfung der Codebasis hinsichtlich qualitativer Vorgaben nicht mehr wegzudenken. Eine Umstellung ohne diese Unterstützung dürfte in den meisten Umfeldern praktisch nicht durchführbar sein.

Die Verwaltung der Artefakte durch Nexus oder Artifactory geschieht neutral gegenüber der Sprache, sodass hier keine Einschränkungen vorliegen. Die Programme zum kontinuierlichen Bauen – besser als Continuous Integration (CI) bekannt – haben auch nur wenig eigene Abhängigkeiten zur Sprachversion. Zum Beispiel bietet Jenkins in der aktuellen Version von sich aus an, das JDK 1.8 herunterzuladen, und

läuft ohne erkennbare Hindernisse auf der neuen JVM. Auch in älteren Versionen kann das aktuelle JDK eingebunden werden.

Auf der Seite der Werkzeuge zur Überprüfung der Codequalität muss hingegen mehr Aufwand investiert werden, die neuen Sprachmittel müssen verstanden werden. Trotzdem ist die Unterstützung schon sehr fortgeschritten: SonarQube in der neuesten Version 4.3 analysiert bereits Java 8, somit kann die Überprüfung auch in Jenkins einfach integriert werden. Dies gilt allerdings nur für die direkt in SonarQube konfigurierten Vorgaben.

PMD wurde bereits aktualisiert, und das Plug-in für SonarQube ebenfalls, ab der Version 2.2 des Plug-ins wird durch `sonar.java.source=8` von PMD 5.1.1 auch Java 8 verstanden. Checkstyle wurde noch nicht angepasst, sodass unser Code nur solange erfolgreich überprüft wird, wie keine Lamdas oder Default-Methoden eingesetzt werden. Die Unterstützung ist für diesen Sommer angekündigt, genauer nach dem „Google Summer of Code".

Ähnlich sieht es mit FindBugs aus – die Version 3.0.0 befindet sich in der Entwicklung, ein Release-Termin ist noch nicht absehbar. Hier wiederholt sich die Geschichte: Der Code wird von zwei Studenten in ihrer Freizeit gewartet, anderweitige Unterstützung existiert nicht, dementsprechend ist auch der Fortschritt. Somit sind diese beiden beliebten Werkzeuge und die darauf aufsetzenden Integrationen in die Entwicklungsumgebungen und in SonarQube nicht nutzbar. In SonarQube können wir uns im Moment nur damit behelfen, alle entsprechenden Regeln dieser Tools zu deaktivieren.

JODA ablösen

Die Anwendung ist migriert und läuft. Eigentlich ist nichts mehr zu tun. Allerdings ist absehbar, dass die referenzierten JODA-Bibliotheken nicht mehr lange aktualisiert werden, denn Java 8 bringt mit dem JSR 310 ein neues API für Datumszugriffe von dem führenden Kopf hinter JODA mit, siehe [ThreeTen]. Um auf der sicheren Seite zu sein, sollten wir diese Abhängigkeit gleich entfernen. Die Spring Pet Clinic verwendet die Bibliothek nur wenig. Dafür wird sie auch in den Webseiten verwendet. Die Klasse `DateTime` zur Abbildung von Tagen wird direkt mittels Hibernate persistiert. Das verspricht einige Komplikationen.

In der Oberfläche werden Datumswerte mittels der Bibliothek `joda-time-jsptags` formatiert ausgegeben. Diese Bibliothek oder ein ähnlicher Ersatz steht uns nach letztem Kenntnisstand nicht für die entsprechenden Klassen aus dem JSR 310 zur Verfügung. Damit wir diese Bibliothek trotzdem entfernen können, müssen wir ein wenig Code beisteuern: In der Anwendung findet sich nur die Verwendung in der Art von

```
<joda:format value="${pet.birthDate}" pattern="yyyy-MM-dd"/>.
```

Diese Auszeichnung muss ersetzt werden, was ohne großen Aufwand mittels eines selbstgeschriebenen Tags auch funktioniert

```
<dateAndTime:date value="${pet.birthDate}" pattern="yyyy/MM/dd" />.
```

Die dafür notwendige Klasse `DateTag` und die Beschreibungsdatei dateAndTime.tld finden sich in der migrierten Anwendung unter [Bit].

Die eigentliche Leistung bei der Umstellung von JODA auf JSR 310 ist die Abbildung von `DateTime` auf die passende neue Klasse. Diese zentrale Klasse gibt es in dieser Form nicht mehr. In Listing 3 sind die möglichen Kandidaten aufgeführt, die sich in der Behandlung der Zeitzone unterscheiden. In diesem Fall werden nur Geburtstage verwendet, also eine Tagesangabe ohne Zeitzone. Das spricht für eine Migration auf `LocalDate`. An manchen Stellen muss zwischen `java.util.Date` und der entsprechenden neuen Klasse konvertiert werden, in diesem Fall lohnt sich der Artikel von Joachim Van der Auwera unter [Auw13].

```
LocalDate = "2014-07-18"
LocalDateTime = "2014-07-18T13:15:00"
ZonedDateTime = "2014-07-18T13:15:00 Europe/Berlin"
OffsetDateTime = "2014-06-18T13:15:00+02:00"
```

Listing 3: Zeitdefinitionen

Wie können wir diese neuen Typen in unseren Entities verwenden und ohne manuelle Konvertierungen direkt auf Datenbankfelder abbilden? Für JODA gibt es eine Bibliothek, die die Datentypen als sogenannte UserTypes verfügbar macht, wie sieht es für `LocalDate` und Konsorten aus? Diese Mühe hat sich das Jadira-Projekt [Jad] bereits gemacht. Die verfügbaren Bibliotheken ab der Version 3.1.0.GA erlauben eine sanfte Migration von JODA auf JSR 310. In Listing 4 steht der vorhandene Code mit den Annotationen für JODA.

```
@Entity
@Table(name = "pets")
public class Pet extends NamedEntity {
  @Column(name = "birth_date")
  @Type(type = "org.jadira.usertype.dateandtime.joda.PersistentDateTime")
  @DateTimeFormat(pattern = "yyyy/MM/dd")
  private DateTime birthDate;
```

Listing 4: JODA mit Jadira

Die Änderung ist denkbar einfach wie in Listing 5 zu sehen. Hierfür muss nur die Extended statt der Core-Version von Jadira im Build referenziert werden. Somit reduziert sich die Migration auf eine reine Fleißarbeit ohne große Umstrukturierungen im Code, und letztendlich bleibt keine Referenz auf JODA übrig.

```
@Column(name = "birth_date")
@Type(type = "org.jadira.usertype.dateandtime.threeten.
PersistentLocalDate")
@DateTimeFormat(pattern = "yyyy/MM/dd")
private LocalDate birthDate;
```

Listing 5: JSR 310 mit Jadira

Lambdas

Bisher können wir sagen, die Migration funktioniert, die Anwendung läuft unter Java 8, aber einen richtigen Nutzen haben wir aus der Umstellung noch nicht gezogen. Der Nutzen ergibt sich erst, wenn neue Anforderungen und neue Funktionalitäten gewünscht sind, oder wenn die Anwendung unter der vorherigen Java-Version nicht mehr sicher betrieben werden kann – wenn es nämlich keine Sicherheits-Updates für die Version gibt.

Können wir den Code auch besser lesbar oder sogar schneller machen? Bei dieser Anwendung lässt es sich nur andeuten, dafür sind die Datenmengen zu gering beziehungsweise ist die Logik zu unspektakulär. Damit ergibt sich auch eine Handlungsempfehlung, die sinnvoll angesetzt werden kann: bei der Verarbeitung der meisten Daten beziehungsweise bei der komplexesten Logik.

Schleifen lassen sich natürlich etwas kürzer mit Lambdas ausdrücken, aber nicht jeder freundet sich mit den neuen Sprachmitteln so schnell an, dass er den Code versteht. Ein Beispiel

```
for (String item : items) {
  if (item.equals(name)) {
    return item;
  }
}
return null;
```

Diese immer wiederkehrende Aufgabe lässt sich auch in einer Zeile ausdrücken

```
return items.stream()
  .filter(i -> i.equals(name))
  .findFirst()
  .orElse(null);
```

Als Faustregel sollte gelten: Wenn ein Lambda eine Codestelle deutlich lesbarer macht, dann bitte anwenden. Ansonsten den Code belassen, wie er ist, er wird nämlich meistens nicht schneller, sondern etwas langsamer. Gerade die leichte Verwendung von parallelen Streams lädt dazu ein, sie schnell herunter zu schreiben – meistens ist das aber die langsamste Variante von allen.

Statistik

Apropos langsam – der Compiler mit der 7 ist immer noch der schnellste. Der Build mit der 8. Ausgabe von Java hat durchgehend etwas länger gebraucht, allerdings reden wir hier von ein bis zwei Sekunden. In der Anwendung war kein Unterschied messbar, es war auch keiner spürbar.

Interessanterweise hatte ich persönlich das Gefühl, Eclipse würde schneller starten. Das kann durchaus an der neuen Speicherverwaltung liegen. Die Anwendung war nach dem Entfernen von JODA und dem Hinzufügen der speziellen UserType-Bibliothek etwas größer als vorher, was so nicht zu erwarten war. Der Migrationsaufwand der Codebasis war eher gering, nach vier Stunden waren die gröbsten Hürden schon beseitigt.

Nicht zu unterschätzen ist der Aufwand für die Aktualisierung der verwendeten Tools und Bibliotheken. Selbst für dieses überschaubare und einigermaßen aktuelle Projekt beträgt die initiale Investition zwei bis drei Arbeitstage. Erfreulich ist bereits die Anzahl der Treffer im Internet, wenn der Fortschritt etwas ins Stocken kommt und Hilfe benötigt wird. Auf der Seite des migrierten Projektes unter [Bit] wird überdies eine Liste mit den Links zu den in diesem Artikel aufgeführten oder erwähnten Produkten, Tools und Bibliotheken bereitgestellt.

Fazit

Die Umstellung auf Java 8 funktioniert überraschend einfach. Die Entwicklungsumgebungen sind bestens vorbereitet, auch wenn die Unterstützung in Eclipse noch etwas mühselig nachinstalliert werden muss. Aktuelle Anwendungsserver sind auf die Erweiterungen der Sprache vorbereitet, wobei die Liste noch überschaubar ist: JBoss Wildfly 8, Tomcat 7.0.54+ sowie Jetty 9. Glassfish kommt in der Version 4.0.1 später in diesem Jahr noch hinzu.

Die gängigen Build-Werkzeuge wie Maven und Gradle funktionieren bereits reibungslos, auch viele Produktivitätswerkzeuge wie Jenkins sind heute einsetzbar. Ein Upgrade auf neuere Versionen ist meistens Voraussetzung. Einige Einschränkungen müssen heute noch gemacht werden, denn nicht alle Projekte und Produkte können oder wollen zeitnah kompatibel zu Java 8 werden, zum Beispiel das beliebte FindBugs, fast alle stellen dies aber für die nächsten Monate in Aussicht.

Die Kernwerkzeuge funktionieren bereits heute einwandfrei, die meisten anderen werden im nächsten halben Jahr folgen. Wir können bereits jetzt produktiven Code auf Java 8 migrieren, mehr Freude werden wir damit am Ende des Sommers haben.

Literatur und Links

[Auw13] J. Van der Auwera, Von Date zu JSR 310, `http://blog.progs.be/542/date-to-java-time`

[Bit] migrierte Anwendung, `https://bitbucket.org/goeckeler/petclinic-on-java8`

[Git] Spring Pet Clinic Sourcen, `https://github.com/spring-projects/spring-petclinic`

[Jad] Jadira User Types, `http://jadira.sourceforge.net/usertype.extended/index.html`

[JSR310] Java Specification Request 310: Date and Time API, `https://jcp.org/en/jsr/detail?id=310`

[Pet] Spring Pet Clinic, `http://demo-spring-petclinic.cfapps.io/`

[ThreeTen] JSR-310 Date and Time API, `http://www.threeten.org/`

JavaSPEKTRUM ist eine Fachpublikation des Verlags:

SIGS DATACOM GmbH
Lindlaustraße 2c • 53842 Troisdorf
Tel.: 0 22 41 / 23 41-100 • Fax: 0 22 41 / 23 41-199
E-Mail: info@sigs-datacom.de • www.javaspektrum.de

SIGS DATACOM
FACHINFORMATIONEN FÜR IT-PROFESSIONALS

Dipl.-Inform. Thorsten Göckeler entwirft seit über zwanzig Jahren Softwaresysteme, seit 2000 überwiegend in Java, und ist durchaus ein Entwickler mit Migrationshintergrund. Leicht agil unterwegs liegt sein Augenmerk auf dem Mehrwert, den der Code erzeugt.
E-Mail: thorsten.goeckeler@iteratec.de