„Java 8 ist ein gutes Release“ – Interview mit Stephen Colebourne ▸23

S&S

JAVA Mag

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 6.2014

Java™ magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Android meets BaaS
Jetzt wird’s interaktiv! ▸114

Simply JNI
Eine Brücke zur Insel ▸10

Crashkurs CQRS
Ein Command geht seinen Weg ▸93

JavaFX 8

Für Java 8 aufgehübscht ▸29

Tipps für JavaFX in Enterprise-Anwendungen ▸44

JavaFX für die Energiewende ▸51

JavaFX-Fallbeispiel: Anatomie eines Scrum Boards ▸73

Ökosystem: TestFX, DataFX, MarvinFX ▸58, 66

4 194586 709801 06

Java 8 Streams: Einführung in das Java 8 Stream API ▸18

Softwarequalität: Besser und entspannter entwickeln ▸106

Continuous Delivery in der Cloud: Umgebungen fallen vom Himmel ▸85

Aufbau einer JPA-Anwendung

Karaf 3.0 im Enterprise

Nachdem im letzten Artikel die Grundlagen vorgestellt wurden, geht es im Folgenden um die Enterprise-Features von Apache Karaf, speziell wie eine JPA-Anwendung mit Karaf aufgebaut werden kann. Damit die JPA-Anwendung in Karaf getestet werden kann, werden auch eigene Karaf-Kommandos hierfür erstellt.

von Achim Nierbeck

Da in diesem Artikel die Apache-Karaf-Besonderheiten im Vordergrund stehen, basiert das JPA-Beispiel auf dem von OpenJPA bereitgestellten Tutorial. Dieses wird adaptiert, um im OSGi-Kontext eingesetzt werden zu können.

Karaf vorbereiten für JPA

Mit Apache Karaf 3.0 ist es einfacher geworden, eine JPA-Anwendung zu erstellen. Zusätzlich zu den bisher auch schon von 2.3 bekannten Features für JPA und JTA, die durch das Apache-Aries-Projekt bereitgestellt werden, gibt es mit Version 3.0 zwei JPA implementierende Features: Hibernate und OpenJPA. In diesem Artikel soll OpenJPA zum Einsatz kommen. Zu Beginn starten wir den Karaf-Container mit *bin/karaf*. Im Anschluss werden die Features JPA, Transaction und JNDI installiert.

Hierdurch werden 33 weitere Bundles installiert, die alle benötigten Bundles enthalten, um eine JPA-Anwendung zu betreiben. Karaf stellt ebenfalls ein Feature für OpenJPA bereit, das an dieser Stelle allerdings nicht installiert wird, da es hiermit noch ein Problem gibt (dieses ist aber ab Version 3.0.1 bereits gelöst).

JPA-Tutorial anpassen

Die im OpenJPA-Tutorial [1] verwendeten zwei Entitätsklassen liegen in dem hier verwendeten Beispiel im Basispackage *de.inovex.javamagazin.jpa*. Die Tutorial-*InventoryEntityBroker*-Klasse wird in das Subpackage *de.inovex.javamagazin.jpa.broker.impl* verschoben und in *InventoryEntityBrokerImpl* umbenannt. Passend zu der Klasse wird ein Interface erstellt. Die meisten IDEs unterstützen hierzu eine Generierung. Dieses Interface liegt nun im Package *de.inovex.javamagazin.jpa.broker* (Listing 1).

Auch die im OpenJPA-Tutorial bereitgestellte *persistence.xml*-Datei muss für den Einsatz im Apache-Karaf-Container angepasst werden. So kann im Karaf auf JTA-Transaktionen zugegriffen werden. Auf die Datenbankverbindung wird per JNDI zugegriffen. Dazu ist es notwendig, eine DataSource als OSGi Service einzubinden.

Starten von Karaf

```
        __ __                  ____
       / //_/ ____ __________/ __ /
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

Apache Karaf (3.0.0)
Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown Karaf.
karaf@root()> feature:install transaction jpa jndi
```

Artikelserie

Teil 1: Grundlagen und Neuigkeiten
Teil 2: Aufbau einer JPA-Anwendung
Teil 3: Webvisualisierung im Zusammenspiel mit Pax Web

Listing 1

```
public interface InventoryEntityBroker {
  // Item methods
  public abstract List<InventoryItem> getAllItems();
  public abstract InventoryItem getSingleItem(int id);
  public abstract void addItem(String name, String description, float
                                                        price, int categoryID);
  public abstract void updateItem(int id, String name, String description,
                                                    float price, int categoryID);
  public abstract void deleteItem(int id);
  // Category Methods
  public abstract List<InventoryCategory> getAllCategories();
  public abstract InventoryCategory getSingleCategory(int id);
  public abstract void addCategory(String name, String description);
  public abstract void updateCategory(int id, String name, String
                                                              description);
  public abstract void deleteCategory(int id);
}
```

JDBC DataSource anlegen

Zur Unterstützung von JDBC DataSourcen gibt es das *jdbc*-Feature, das per *feature:install jdbc* installiert wird. Es muss lediglich zusätzlich ein geeigneter Datenbanktreiber installiert werden. In diesem Artikel verwenden wir eine H2-Datenbank. Hierzu das H2-Datenbank-Bundle mittels *bundle:install -s mvn:com.h2database/h2/1.3.167* in der Karaf-Shell installieren.

Danach wird die passende DataSource ebenfalls in der Karaf-Shell angelegt: *jdbc:create -t H2 -url jdbc:h2:${karaf.data}/database/h2Test -u sa -d org.h2.jdbcx.JdbcDataSource javamagazin*. Dabei wird direkt im *deploy*-Verzeichnis eine Blueprint-Definition wie in Listing 2 im Apache-Karaf-Server erstellt.

Diese DataSource kann jetzt verwendet werden. Daraus ergibt sich für die *persistence.xml* eine Konfiguration, wie sie in Listing 3 gezeigt ist. Wichtig ist hierbei zum einen der *transaction-type*, der auf JTA gestellt ist, und zum anderen die Verbindung zur DataSource per JNDI. Der JNDI-*lookup*-Name setzt sich dabei wie folgt zusammen:

- *osgi:service*: Referenz eines OSGi-Dienstes; sobald das JNDI-Feature installiert ist, sind alle OSGi-Dienste per JNDI referenzierbar. Damit wird gekennzeichnet, dass es sich um einen solchen Dienst handelt.
- *javax.sql.DataSource*: Das Interface, unter dem der gesuchte Dienst registriert ist.
- *(osgi.jndi.service.name=jdbc/javamagazin)*: Dies ist ein Filter (LDAP-Syntax), damit nach genau einem Dienst zum Auflösen der DataSource gesucht wird.

Da Aries JPA nur das API bereitstellt, wird noch eine Implementierung benötigt. Hier hat sich Apache OpenJPA im Kontext von OSGi bis dato am besten geeignet gezeigt. In der Karaf-Shell muss das *openjpa*-Feature installiert werden. Da dieses erst mit Karaf 3.0.1 fehlerfrei funktioniert, ist im Beispielprojekt [2] ein fehlerfreies OpenJPA-Feature enthalten. Ist die Infrastruktur soweit vorbereitet, kann das eigene JPA Bundle mit *bundle:install -s mvn:de.inovex.javamagazin/Karaf-JPA/1.0.0-SNAPSHOT* installiert werden.

JPA Commandline Client

Zum Testen des JPA Bundles bietet es sich an, ein paar Shellkommandos für rudimentäre CRUD-Befehle zu erstellen. Hierzu erstellt man ein Maven-Projekt mit dem von Karaf bereitgestellten Archetype. Ein komplettes Beispiel ist unter [2] zu finden:

```
mvn archetype:create
    -DarchetypeArtefactId=karaf-command-archetype
    -DgroupId=de.inovex.javamagazin
    -DartifactId=Karaf-JPA-Command
    -Dversion=1.0.0-SNAPSHOT
```

Listing 2

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <bean id="dataSource" class="org.h2.jdbcx.JdbcDataSource">
    <property name="URL" value="jdbc:h2:/Users/anierbeck/apache-karaf-3.0.0/data/database/h2Test"/>
    <property name="user" value="sa"/>
    <property name="password" value="null"/>
  </bean>

  <service interface="javax.sql.DataSource" ref="dataSource">
    <service-properties>
      <entry key="osgi.jndi.service.name" value="jdbc/javamagazin"/>
    </service-properties>
  </service>
</blueprint>
```

Listing 3

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0" >
  <persistence-unit name="inventory" transaction-type="JTA">
    <provider>org.apache.openjpa.persistence.PersistenceProviderImpl</provider>
    <jta-data-source>osgi:service/javax.sql.DataSource/(osgi.jndi.service.name=jdbc/
                                                    javamagazin)</jta-data-source>
    <class>de.inovex.javamagazin.jpa.InventoryItem</class>
    <class>de.inovex.javamagazin.jpa.InventoryCategory</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <validation-mode>NONE</validation-mode>
    <properties>
      <property name="openjpa.jbdc.Schema" value="StoreSystem" />
      <property name="openjpa.Log" value="slf4j" />
      <property name="openjpa.RuntimeUnenhancedClasses" value="supported" />
      <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema"/>
    </properties>
  </persistence-unit>
</persistence>
```

Listing 4

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" default-
                                                          activation="lazy">

  <reference id="entityBroker" interface="de.inovex.javamagazin.jpa.broker.
                                                      InventoryEntityBroker" />

  <command-bundle xmlns="http://karaf.apache.org/xmlns/shell/v1.1.0">
    <command>
      <action class="de.inovex.javamagazin.jpa.command.CreateCategory">
        <property name="broker" ref="entityBroker" />
      </action>
    </command>
  </command>
```

Listing 5

```
@Command(scope = "javamagazin", name = "CreateCategory", description = "CRUD
                                          commands for the JavaMagazin samples")
public class CreateCategory extends OsgiCommandSupport {
  @Option(name = "-d", aliases = { "--description" }, description = "A description to the
                                  category", required = false, multiValued = false)
  private String description;
  @Argument(name = "name", description = "Name of Category", required = true,
                                                          multiValued = false)
  private String name;
  ...
  protected Object doExecute() throws Exception {
    broker.addCategory(name, description);
    return null;
  }
}
```

Die sich daraus ergebende Projektstruktur enthält ein Beispielkommando und das dazugehörige *blueprint.xml*. Beides muss für den Zweck eines CLI-Interface zum Bearbeiten der Entitätsobjekte angepasst werden. Im *blueprint.xml* wird die Referenz auf den *InventoryEntityBroker*-Dienst hinzugefügt, der dem Shellkommandoobjekt injiziert wird.

Die zum *blueprint.xml* passende *Create*-Klasse sieht dann wie in Listing 5 aus. Mit der *@Command*-Annotation wird definiert, dass diese Klasse ein Karaf-Shellkommando ist. Mit *@Option* werden die optionalen Argumente definiert, das eigentliche Shellargument wird mit *@Argument* annotiert. Bei Aufruf des Kommandos wird die *doExecute*-Methode ausgeführt.

Nachdem das Maven-Modul erfolgreich gebaut worden ist, kann es jetzt im Karaf installiert werden. Im Anschluss stehen sofort die neuen Shellkommandos in der Karaf-Shell zur Verfügung. In der Shell mit *javamagazin* in die entsprechende Subshell wechseln. Wahlweise kann auf die Subshell verzichtet werden, und dann wird dem *createcategory*-Befehl ein *javamagazin:* vorangestellt.

Beide Bundles können auch durch das Feature installiert werden. Hierbei wird auch gleich das funktionierende OpenJPA-Feature, das als Abhängigkeit definiert ist, installiert. Dazu die Maven-Koordinate des Features mit folgendem Befehl dem Feature-Repository bekannt machen: *feature:repo-add mvn:de.inovex.javamagazin/Karaf-JPA-Feature/1.0.0-SNAPSHOT/xml/features*. Im Anschluss kann das Feature mittels *feature:install*-Karaf-JPA-Feature installiert werden. Danach stehen folgende Kommandos auf der Karaf-Shell zur Verfügung:

- *createcategory* – erstellt eine Kategorie
- *createitem* – erstellt ein Item
- *deletecategory* – löscht eine Kategorie
- *listcategory* – listet alle Kategorien in einer Tabelle
- *listitem* – listet alle Items in einer Tabelle

Ausblick

Da jetzt erfolgreich Daten persistiert werden können, geht es im nächsten Artikel um die Kombination der in diesem Artikel vorgestellten JPA-Komponente mit einer Webanwendung.

Achim Nierbeck ist Seniorentwickler/-architekt/-Scrum-Master bei der inovex GmbH und beackert das Feld der EE-Entwicklung schon seit über zwölf Jahren. Seit nunmehr drei Jahren beschäftigen ihn Apache Karaf PMC OSGi und vor allem das Thema Karaf.

Links & Literatur

[1] http://openjpa.apache.org/begin-using-openjpa---the-basics.html

[2] https://github.com/ANierbeck/Karaf-Sample

[3] OSGi Compendium Spezification 134-3 Subsystem Region