Studienarbeit

Datenzugriffskomponenten

mit

JPA 2.1

von: Nils Grünewald

FWP-Fach: Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Gutachter: Herr Michael Theis

10.06.2016

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

München, den 10.06.2016

Nils Grünewald

Gliederung

Einleitung

Die meisten heutigen Anwendungen verarbeiten Daten und stellen sie späteren Nutzern von Anwendungen zur Verfügung. Das Erzeugen von Daten bzw. Entities kann einfach in Java-Code umgesetzt werden, jedoch stellt hier das Abschalten der Applikation ein Problem dar, da jeglicher Zwischenspeicher dann gelöscht wird. Daher wird eine Datenbank verwendet, um die Entities zu permanent speichern. Hierfür kann über JDBC eine Verbindung zwischen dem Java-Code und der Datenbank hergestellt werden.

Das Aufwändige hierbei ist, dass für jede Entity ein Data-Access-Object implementiert werden muss.

Durch die Einführung von JPA (Java Persistence API) wurde dies vereinfacht. Diese API definiert für Frameworks eine Reihe von standardisierten Anfragen an die Datenbank, sodass Entwickler sich kaum mehr um die Datenbank-Schicht kümmern brauchen.

Viele Operationen, die bisher in SQL geschrieben werden mussten, können nun als simpler Java-Code geschrieben werden. Ebenso bringt die aktuelle Version eine eigene SQL-ähnliche Sprache (JPQL) und eine Criteria-API mit, auf die in dieser Arbeit näher eingegangen werden soll.

Gängige JPA-Frameworks wären:

- EclipseLink

- Hibernate

Alle in dieser Arbeit vorgeführten Beispiele wurden mit EclipseLink 2.5.0-RC1, MySQL 5.7, MySQL-Connector 6.0.2 und EJB 3.2 entwickelt.

Schichtenmodell

Nach heutigem Standard wird Software nach der sogenannten Drei-Schichten-Architektur aufgebaut. Die drei Schichten bauen sich auf aus[1]:

- der Präsentationsschicht (UI Layer)

- der Anwendungsschicht (Business Logic Layer)

- und der Datenhaltungsschicht (Data Layer).

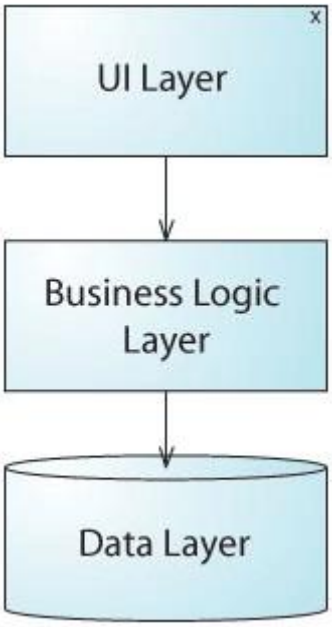

Abbildung 1 [2]

Die erste dient dazu, mit dem Benutzer zu interagieren und die Daten sinnvoll darzustellen. Sie kann mit beiden darunter liegenden Schichten kommunizieren, jedoch darf sie keine Daten aus der dritten Schicht bearbeiten oder speichern. Die zweite, auch Business-Schicht genannt, verarbeitet logische Schritte, Bedingungen und Rechenoperationen. Sie bietet der GUI-Schicht diverse Methoden für den Zugriff zum lesen/schreiben auf die Datenhaltungsschicht, welche dann die tatsächliche Transaktionen vornimmt.

JPA setzt genau an dieser Stelle an. Es soll dem Entwickler die Zugriffe auf die Datenbank erleichtern.

Konfiguration

JPA-Framework einbinden

Um JPA zu benutzen, muss dieses Framework eingebunden werden, was manuell oder über Maven geschehen kann. Es empfiehlt sich jedoch letzteres zu benutzen, da hier alle Dependencies bequem nachgeladen und verwaltet werden können.

Hierfür wird der Maven-Konfigurationsdatei pom.xml innerhalb des Bereiches für die Dependencies die Dependency der groupId `org.eclipse.persistence` und der artifactId `eclipselink` hinzugefügt. In diesem Beispiel wird EclipseLink mit der Versionsnummer `2.5.0-RC1` als Framework benutzt:

```
<dependencies>
	<dependency>
		<groupId>org.eclipse.persistence</groupId>
		<artifactId>eclipselink</artifactId>
		<version>2.5.0-RC1</version>
	</dependency>
</dependencies>
```

Des weiteren wird ein passender Treiber zur Verbindung mit der Datenbank benötigt. Für MySQL würden hierfür die Dependencies um den MySQL-Connector erweitert:

```
<dependencies>
	<dependency>
		<groupId>org.eclipse.persistence</groupId>
		<artifactId>eclipselink</artifactId>
		<version>2.5.0-RC1</version>
	</dependency>
	<dependency>
		<groupId>mysql</groupId>
		<artifactId>mysql-connector-java</artifactId>
		<version>6.0.2</version>
	</dependency>
</dependencies>
```

Einstellung der persistence.xml

Im Ressourcen-Ordner META-INF wird eine XML-konforme Datei namens persistence.xml generiert:

```
<?xml version="1.0" encoding="UTF-8"r?>
<persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
</persistence>
```

Darin wird eine Persistence-Unit definiert, in welcher alle Eigenschaften dieser Unit definiert werden. Die Eigenschaften bauen sich auf aus dem Namen dieser Unit, den Verbindungsdaten zur Datenbank und den Zugriffsrechten darauf. Ebenso werden hier alle Entities registriert.

```
	<persistence-unit name="jpa-demo">

	<class>model.Book</class>

	<properties>
		<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/jpadb?serverTimezone=UTC" />
		<property name="javax.persistence.jdbc.user" value="root" />
		<property name="javax.persistence.jdbc.password"
value="wA_20*ä" />
		<property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver" />
		<property name="eclipselink.logging.level" value="FINE" />
		<property name="eclipselink.ddl-generation" value="create-
tables" />
	</properties>

	</persistence-unit>
</persistence>
```

- Unter *javax.persistence.jdbc.url* wird die URL eingetragen, unter der die Datenbank zu erreichen ist
- Unter *javax.persistence.jdbc.user* & *javax.persistence.jdbc.password* werden die Zugangsdaten zu der Datenbank gespeichert
- Der *javax.persistence.jdbc.driver* dient JPA zur Verbindung zur Datenbank. Hier wird der entsprechende Connector definiert, der hierfür notwendig ist.
- *eclipselink.logging.level* ist für das Loggen der erzeugten SQL-Befehle zuständig. Immer wenn JPA auf die Datenbank zugreift, generiert es SQL-Befehle. Diese können z.B. durch die Einstellung FINE in der Konsole zurück gegeben werden.
- Auf die Einstellung *eclipselink.ddl-generation* wird im nächsten Kapitel eingegangen.

In diesem Beispiel wurde der Persistence-Unit jpa-demo die Klasse model.Book hinzugefügt. Die URL baut sich auf aus:

- jdbc:mysql://

- Domain (hier localhost)

- Portnummer, auf die der SQL-Server läuft

- Datenbankname

?serverTimezone=UTC dient nur dazu, die Zeitzone festzulegen.

DDL-Generation

JPA bringt verschiedene Optionen zur Generierung von DDL-Code (Data Definition Language) während der Laufzeit mit. So kann zB während der Laufzeit erst die Datenbankstruktur geändert werden, oder neue Tabellen erzeugt.

Festgesetzt wird dies in der folgenden Property der Persistence-Unit:

```
<property name="eclipselink.ddl-generation" value="..." />
```

Der Tabelle können die Möglichkeiten entnommen werden[3]:

Value	Beschreibung
none	Dies ist als Standardwert vorbelegt. Hier können keine Tabellen erzeugt werden
create-tables	Für jede Entity wird eine Tabelle erzeugt. Sollte es die zugehörige Tabelle schon geben, wird diese im Normalfall weiterhin verwendet.
create-or-extend-tables	Das selbe wie „create-tables“ mit der Möglichkeit zur Veränderung der Tabelle.
drop-and-create-tables	Für Entwicklung gedacht, nicht in produktiv: Zu jeder Entity werden alle eventuell existierende Tabellen gelöscht und anschließend neu erzeugt.

Entities

Eine Entity ist ein materielles oder immaterielles Objekt, das in der Datenbank abgebildet werden soll. Zwischen mehreren Entitäten können Beziehungen zueinander herrschen. Normalerweise wird vor dem Implementieren ein ER-Diagramm (Entity-Relationship) erstellt, das diese Beziehungen verdeutlicht. Die Eigenschaften dieser Entitäten werden Attribute genannt. [4]

Um eine JPA-Entity zu erstellen, wird in Java eine Klasse erstellt. Dies hat unbedingt eine Klasse zu sein, andere Typen wie Interfaces werden nicht akzeptiert. Die Mindestanforderungen sind:

Klasse:

- sie muss nach außen hin sichtbar sein (public/protected)

- Deklaration der Entity als solche

- Kennzeichnung des Primärschlüssels als ID

Konstruktor:

- es wird ein ein Default-Konstruktor (Konstruktor ohne Parameter) benötigt.

- Default-Konstruktor muss public oder protected sein

Datenbanktabelle:

- Die zugehörige Tabelle muss eine Spalte für Primärschlüssel beinhalten

- Entity benötigt ebenso eine Variable für die ID (Primärschlüssel)

Die Kennzeichnung als Entity und Primärschlüssel wird über Annotationen geregelt.

Für die Entity wird die Klasse mit @Entity annotiert, für den Primärschlüssel die entsprechende Variable mit @ID. Idealerweise sollte die ID einen Long-Wert darstellen.

Hier ein Beispiel für eine Entity Book:

```
package model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Book {

	@Id
	@GeneratedValue(strategy= GenerationType.IDENTITY)
	private int b_id;

	private String b_title;
	private String b_author;

	public Book() {
	}

	public Book(int b_id, String b_title, String b_author) {
		super();
		this.b_id = b_id;
		this.b_title = b_title;
		this.b_author = b_author;
	}
```

```
	public int getId() {
		return b_id;
	}

	public void setId(int b_id) {
		this.b_id = b_id;
	}

	public String getTitle() {
		return b_title;
	}

	public void setTitle(String b_title) {
		this.b_title = b_title;
	}

	public String getAuthor() {
		return b_author;
	}

	public void setAuthor(String b_author) {
		this.b_author = b_author;
	}
}
```

Die zugehörige Tabelle wäre wie in der Abbildung aufgebaut:

B_ID	B_AUTHOR	B_TITLE
1	Michael Theis	Aktuelle Java Technologien
2	Nils Grünewald	JPA 2.1
NULL	NULL	NULL

Die Annotation `@GeneratedValue(strategy= GenerationType.IDENTITY)` sorgt dafür, dass die ID automatisch bei Erstellung einer Entity generiert wird. Hierfür gibt es diverse Strategien, die je nach angewandter Datenbank unterschiedlich eingesetzt werden können.

Der EntityManager

Transaktionen mit der Datenbank werden in Java-SE Applikationen über den EntityManager gesteuert. Somit dient er als Steuerzentrale in JPA und bietet verschiedene Funktionen an, um Entities in der Datenbank zu suchen, speichern, aktualisieren oder zu löschen.

Er wird über die EntityManagerFactory generiert, welche wiederum über die Persistence erzeugt und über die vorher erwähnte PersitenceUnit konfiguriert wird.

Der EntityManager verwaltet im Hintergrund den PersistenceContext. Dies ist eine Art Zwischenspeicher für sämtliche verwalteten JPA-Entities. Da der EntityManager nicht threadsicher ist, werden Transaktionen bei JavaEE-Applikationen vom EJB-Container übernommen.

Erzeugen eines EntityManagers:

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import model.Book;

public class CreateBook {
      public static void main( String[ ] args ) {
            EntityManagerFactory emfactory =
                  Persistence.createEntityManagerFactory( "jpa-demo" );
            EntityManager entitymanager = emfactory.createEntityManager( );

            ...

            entitymanager.close();
            emfactory.close();
      }
}
```

Am Ende jeder Transaktion, werden EntityManager und dessen Factory beendet. Beim Erzeugen der Factory ist zu beachten, dass hier der in der persistence.xml definierte Name der Persistence-Unit korrekt gesetzt wird:

```
Persistence.createEntityManagerFactory( "jpa-demo" );
```

Wurde der EntityManager erzeugt, bietet er die folgenden wichtigsten Funktionen an:

Funktion	**Beschreibung**
persist()	Speichert eine neu erstellte Entity in der Datenbank
merge()	Aktualisiert eine Entity in der Datenbank
find()	Sucht eine Entity aus der Datenbank anhand dessen ID
remove()	Löscht eine Entity aus der Datenbank

In den folgenden Beispielen wird davon ausgegangen, dass JAVA SE benutzt wird. Auf die Funktion `.getTransaction( )` wird im folgenden Kapitel eingegangen.

Erstellen einer neuen Entity:

```
public static void main( String[ ] args ) {
	...

	entitymanager.getTransaction( ).begin( );

	Book book = new Book();
	book.setTitle("Aktuelle Java Technologien");
	book.setAuthor("Michael Theis");
	entitymanager.persist( book );
	entitymanager.getTransaction( ).commit( );

	...
}
```

Suchen einer Entity:

```
public static void main( String[ ] args ) {
	...

	entitymanager.getTransaction( ).begin( );

	Book book = entitymanager.find( Book.class, 1 );

	System.out.println("--- Find Book ---");
	System.out.println( book.getId() );
	System.out.println( book.getTitle() );
	System.out.println( book.getAuthor() );

	...
}
```

Der Funktion `.find()` sind Klassentyp der Entity und dessen ID zu übergeben.

Updaten einer Entity:

```
public static void main( String[ ] args ) {
	...

	entitymanager.getTransaction( ).begin( );
	Book book = entitymanager.find( Book.class, 1 );
	book.setAuthor( "Michael Theis" );
	entitymanager.merge( book );
	entitymanager.getTransaction( ).commit( );

	...
}
```

Löschen einer Entity:

```
public static void main( String[ ] args ) {
	...

	entitymanager.getTransaction( ).begin( );
	Book book = entitymanager.find( Book.class, 2 );
	entitymanager.remove( book );
	entitymanager.getTransaction( ).commit( );

	...
}
```

Der Funktion `.remove()` sind Klassentyp der Entity und dessen ID zu übergeben.

Transaktionen

Wie in den gerade gezeigten Beispielen deutlich wurde, wird vor jeder Kommunikation mit der Datenbank eine Transaktion durch `entitymanager.getTransaction( ).begin( )` begonnen. Anschließend wird eine der oben genannten Funktionen des EntityManagers aufgerufen, wobei diese Transaktion noch nicht durchgeführt wird. Es wird lediglich hinterlegt, wie vorgegangen werden soll. Erst beim Aufrufen von `entitymanager.getTransaction( ).commit( )` wird die Transaktion ausgeführt, d.h. die Entity in der Datenbank gespeichert.

Bei den Transaktionen muss darauf geachtet werden, um was für eine Anwendung es sich hierbei handelt. Oben aufgeführte Beispiele bezogen sich auf eine Java SE Anwendung. Wird mit EJBs

(Enterprise JavaBeans) gearbeitet, kann das Öffnen und Schließen des EntityManagers sowie das Verwalten über `.getTransaction( )` weggelassen werden.

Stattdessen wird die Klasse mit `@Stateless` annotiert. Der EntityManager wird zwar erzeugt und mit `@PersistenceContext` annotiert, wird hier aber selbstständig verwaltet.

Hier die bereits vorgeführten Beispiele zum Erstellen, Finden, Updaten und Löschen einer Entity, doch diesmal über JavaBeans:

```
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import model.Book;

@Stateless
public class ManageBookEJB implements ManageBookEJBInt {

	@PersistenceContext
	private EntityManager em;

	@Override
	public void saveNewBook(Book book) {
		em.persist(book);
	}

	@Override
	public void findBook(int id) {
		em.find(Book.class, id);
	}


	@Override
	public void updateBook(Book book) {
		em.merge(book);
	}

	@Override
	public void removeBook(Book book) {
		em.remove(book);
	}
}
```

Anmerkung:

Um diese Annotationen nutzen zu können, wird EJB benötigt, welches sich ebenfalls über die Dependencies über Maven beziehen lässt:

```
<dependency>
	<groupId>javax.ejb</groupId>
	<artifactId>javax.ejb-api</artifactId>
	<version>3.2</version>
</dependency>
```

Welche Versionen verwendet werden, ist zunächst einmal irrelevant. Jedoch kann es bei älteren Versionen zu Inkompatibilitäten kommen, sollten einige Features nicht eingebaut sein. Selbiges gilt für die anderen in Maven importierten Dependencies.

JPQL-API & Criteria-API

Die größten Änderungen von JPA 2.0 zu JPA 2.1 betreffen die JPQL-API (Java Persistence Query Language) und die Criteria-API. Diese werden verwendet, um komplexere Datenbankabfragen durch zu führen. Im Gegensatz zu JPQL, wo Anweisungen SQL-ähnlich in Zeichenketten umgesetzt werden, wird Criteria in Java geschrieben. Dies hat den Vorteil, dass Tippfehler (z.B. „SELETC“ statt „SELECT“) noch vor dem Ausführen des Codes, also noch während des Programmierens, erkannt werden können. Im weiteren Verlauf soll näher auf die beiden APIs eingegangen werden.

JPQL

JPQL bietet die folgenden Anweisungen an:

- SELECT

- UPDATE

- DELETE

Hauptsächlich wird es verwendet, um komplexere Anweisungen durch zu führen. Das Erstellen von neuen Entities wird weiterhin über die Standard-JPA-Methoden persist() und merge() ausgeführt.

Das Finden von Entities aus Tabellen in JPA wurde bisher über die Methode find() erledigt. Hier stellt sich jedoch das Problem, dass die ID der Entity bereits bekannt sein muss. Kennt die Anwendung diese nicht, kann die Suche nach der passenden Entitiy über den SELECT-Befehl von JPQL, wo eine feinere Suche konfiguriert wird, erfolgen.

Die Query

Um eine JPQL-Query zu erzeugen, wird der bereits bekannte EntityManager verwendet. Der EntityManager stellt die Methode createQuery() zur Verfügung, mit der eine JPQL Anweisung erstellt werden kann:

```
TypedQuery<Book> query = entitymanager.createQuery( " … ", Book.class);
```

(An Stelle der Punkte wird hier die Query eingetragen, auf die im folgenden Kapitel eingegangen wird. Danach wird der Typ der Entity mitgegeben, um eine TypedQuery zurück zu bekommen.)

Das Ergebnis der Query lässt sich über die eigene Funktion der Query-Klasse getResultList() abrufen:

```
query.getResultList();
```

Das Ergebnis ist eine java.util.List Collection:

```
List<...> books = query.getResultList();
```

(An Stelle der Punkte wird hier die Klasse des erwarteten Ergebnisses eingetragen.)

Syntax einer SELECT-Abfrage

SELECT

Identifikationsvariable.Attribut

FROM

Entity AS Identifikationsvariable

[5, S. 626]

Die Entity repräsentiert hier nicht den Tabellennamen in der Datenbank, sondern den Klassennamen der Entity.

Im folgenden Beispiel werden aus allen Bücher-Entities das Attribut b_author abgefragt. Die Rückgabe sind alle Autoren, zu denen es ein Buch in der Datenbank gibt:

```
public static void main(String[] args) {
	...
	Query query = entitymanager.createQuery(
		"SELECT b.b_author FROM Book b" );
	...
}
```

Book stammt aus dem Klassennamen Book.class.

Zur Vermeidung von Tippfehlern empfiehlt es sich, den Klassennamen durch `.getSimpleName()` in Java-Code ausgeben zu lassen:

```
public static void main(String[] args) {
	...
	Query query = entitymanager.createQuery(
		""SELECT b.b_author FROM " + Book.class.getSimpleName() + " b""
	);
	...
}
```

Hier das Finden aller Bücher aus der Datenbank:

```
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

import model.Book;

public class FindBook {

	public static void main(String[] args) {

		EntityManagerFactory emfactory =
			Persistence.createEntityManagerFactory("jpa-demo");
		EntityManager entitymanager = emfactory.createEntityManager();

		Query query = entitymanager.createQuery( "SELECT b FROM "
			+ Book.class.getSimpleName() + " b" );
```

```
            @SuppressWarnings("unchecked")
            List<Book> books = query.getResultList();

            for (int i=0; i<books.size(); i++) {
                  System.out.println(books.get(i).getId());
                  System.out.println(books.get(i).getAuthor());
                  System.out.println(books.get(i).getTitle());
            }

            entitymanager.close();
            emfactory.close();
      }
}
```

Kürzen der SELECT-Anweisung:

Bei einer Abfrage nach allen Spalten einer Tabelle kann der * weg gelassen werden.

Wird keine Anweisung wie SELECT, UPDATE oder DELETE mitgegeben, wird automatisch die SELECT-Anweisung angehängt [5, S. 627]. Die folgenden drei Beispiele sind vollkommen identisch:

```
public static void main(String[] args) {
      ...
      Query query = entitymanager.createQuery(
            "SELECT b.* FROM Book b" );
      ...
}

public static void main(String[] args) {
      ...
      Query query = entitymanager.createQuery(
            "SELECT b FROM Book b" );
      ...
}

public static void main(String[] args) {
      ...
            Query query = entitymanager.createQuery(
            "FROM Book b" );
      ...
}
```

Suche verfeinern

Wie bei SQL bekannt, gibt es auch bei JPQL diverse Befehle wie die WHERE-Bedingung, die ORDER-BY Klausel oder das Verknüpfen von Anweisungen durch AND/OR, um die Suche bzw. das Ergebnis zu verfeinern.

Die Query zum Suchen eines Buches vom Autoren Max Mustermann würde lauten:

```
public static void main(String[] args) {
	...
	Query query = entitymanager.createQuery(
		"SELECT b.* FROM Book b WHERE b.b_author = 'Max Mustermann'" );
	...
}
```

Das Suchwort wird hier in Hochkommata geschrieben, da es sich hier um einen Vergleichswert handelt. Generell werden wie in SQL solche Zeichenketten und Zeit-/Datumsangaben in Hochkommata gesetzt.

Oben dargestelltes Beispiel kann noch durch Parameter erweitert werden, um SQL-Injections vorzubeugen. Dies bedeutet, dass eine Stelle, wo eine Variable vorkommt, durch einen Parameter ersetzt wird. An anderer Stelle wird dem Parameter dann der Wert zugewiesen. Hierbei werden unter zwei Kategorien unterschieden:

- Positional parameter
- Named parameter

Bei den ersteren wird an der Stelle, wo eine Variable definiert werden soll, lediglich die Position nummeriert. Bei der zweiten wird der Stelle ein Name gegeben. Dem Namen in der Query wird hierbei ein Doppelpunkt vorangestellt. Im Folgenden wird obiges Beispiel noch einmal mit beiden Parametern gezeigt:

Positional Parameter:

```
Query query = entitymanager.createQuery(
		"SELECT b.* FROM Book b WHERE b.b_author = ?1" );
query.setParameter(1, "Max Mustermann");
```

Named Parameter:

```
Query query = entitymanager.createQuery(
		"SELECT b.* FROM Book b WHERE b.b_author = :authorName" );
query.setParameter("authorName", "Max Mustermann");
```

Durch Anhängen einer ORDER BY-Klausel, kann das Ergebnis einer Suche mit mehreren Einträgen sortiert werden. Hier stehen die beiden Optionen ASC und DESC zur Verfügung. ASC (ascend) sortiert die Einträge in aufsteigender Reihenfolge, letzteres in absteigender (descend).

Hier werden alle Bücher nach Namen in alphabetischer Reihenfolge nach Autorennamen sortiert abgefragt:

```
public static void main(String[] args) {
	...
	Query query = entitymanager.createQuery(
		"SELECT b FROM Book b ORDER BY b.b_author ASC" );
	...
}
```

Durch AND und OR lassen sich verschiedene Bedingungen miteinander verknüpfen.

Die Rückgabe aller Bücher des Autoren Max und EJB heißen:

```
public static void main(String[] args) {
	...
	Query query = entitymanager.createQuery(
		"SELECT b FROM Book b WHERE b.b_author = 'Max' AND b.b_title =
'EJB'" );
	...
}
```

Die Rückgabe aller Bücher der Autoren Max oder Nils:

```
public static void main(String[] args) {
	...
	Query query = entitymanager.createQuery(
		"SELECT b FROM Book b WHERE b.b_author = 'Max' OR b.b_author =
'Nils'" );
	...
}
```

Criteria-API

Die Criteria-API wurde entwickelt, da JPQL nicht typsicher ist. Dies kommt daher, dass bei JPQL SQL ähnlicher Code in String-Format erstellt wird. Strings werden aber von IDEs nicht auf Korrektheit überprüft, weshalb Fehler erst zur Laufzeit des Programms auffallen. Um dem entgegen zu wirken, bietet die Criteria-API die Möglichkeit, die SQL Befehle in Java-Code zu schreiben.

Einfache Queries

Gehandhabt wird dies auf dieselbe Art wie JPQL, nur dass hier die Query, also der String in der `.createQuery()` Funktion generiert wird. Dazu bietet der EntityManager die Funktion `.getCriteriaBuilder()`, mit welcher der CriteriaBuilder ausgeführt werden kann.

Um auf das JPQL-Beispiel zum Finden aller Bücher aus der Tabelle zurück zu kommen:

```
public static void main(String[] args) {
	...
	Query query = entitymanager.createQuery(
		"SELECT b FROM " + Book.class.getSimpleName() + " b" );
	...
}
```

Dieses würde mit Criteria folgendermaßen aussehen:

```
public static void main(String[] args) {
	...
	Query query = entitymanager.createQuery(
		entitymanager.getCriteriaBuilder()
		.createQuery(Book.class)
	);
	...
}
```

bzw:

```
public static void main(String[] args) {
	...
	CriteriaBuilder criteriaBuilder = entitymanager.getCriteriaBuilder();
	CriteriaQuery<Book> criteriaQuery =
		criteriaBuilder.createQuery(Book.class);
	Query query = entitymanager.createQuery(criteriaQuery);
	...
}
```

Beide ergeben das selbe Ergebnis. Ersteres kann für einfache Abfragen verwendet werden. Für komplexere bzw. beim Verfeinern der Suche sollte der zweite Ansatz gewählt werden.

Suche verfeinern

Das Verfeinern der Suche lässt sich über die Funktionen `select()`, `.from()` und `.where()` der CriteriaQuery erreichen. Damit für jede dieser Funktionen nicht jedes mal erneut, die CriteriaQuery generiert werden muss, hat es sich bewährt, die Suche in einzelne Schritte zu unterteilen.

Folgendes Beispiel zur Suche aller Bücher des Autoren Nils Grünewald verdeutlicht dies:

```
public static void main(String[] args) {

        Query query = entitymanager.createQuery(
                entitymanager.getCriteriaBuilder()
                        .createQuery(Book.class).where(
                        entitymanager.getCriteriaBuilder()
                                .equal(entitymanager.getCriteriaBuilder()
                                        .createQuery(Book.class).from(Book.class)
                                                .get("b_author"), "Nils Grünewald"))
        );

}
```

Die Funktionen zur Verfeinerung der Suche lassen sich auslagern, sodass obiges Beispiel übersichtlicher wird:

```
public static void main(String[] args) {

        CriteriaBuilder criteriaBuilder = entitymanager.getCriteriaBuilder();
        CriteriaQuery<Book> criteriaQuery = criteriaBuilder.createQuery(Book.class);

        Root<Book> root = criteriaQuery.from(Book.class);
        Path<Object> b_author = root.get("b_author");
        Predicate predicate = criteriaBuilder.equal(b_author, "Nils Grünewald");

        criteriaQuery.select(root).where(predicate);
        Query query = entitymanager.createQuery(criteriaQuery);

}
```

Wie obigem Beispiel entnommen werden kann, wird beim Predicate die eigentliche Suche definiert:

Predicate predicate = criteriaBuilder.equal (b_author, "Nils Grünewald");

Eine weitere Verfeinerung lässt sich somit über den CriteriaBuilder erreichen. Die angebotenen Funktionen lauten [5, S. 659] :

`.between()`	`.equal()`	`.exist()`	`.ge()`
`.greaterThan()`	`.greaterThanOrEqualTo()`		`.greatest(Expression<X>)`
`.gt()`	`.isEmpty()`	`.isFalse()`	`.isMember()`
`.isNotEmpty()`	`.isNotMember()`	`.isNotNull()`	`.isNull()`
`.isTrue()`	`.le()`	`.lessThan()`	`.lessThanOrEqualTo()`
`.like()`	`.not()`	`.notEqual()`	`.notLike()`

Fazit

In dieser Arbeit wurde dem Leser ein schneller und einfacher Einstieg in JPA gewährt sowie JPQL und die Criteria-API näher gebracht. Hierbei handelte es sich lediglich um die Grundlagen. So kann die Arbeit eines Entwicklers durch die Verwendung von NamedQueries noch weiter vereinfacht werden. Ebenso sei erwähnt, dass sich mittels JPQL diverse Joins verfassen lassen.

Die Einarbeitung und Konfiguration kann eventuell anfangs komplizierter wirken, jedoch zahlt sich die Verwendung von JPA spätestens beim Entwickeln der Anwendung aus.

Bei großen rechenintensiven Anwendungen müsste möglicherweise noch auf die Performance geachtet werden. Schließlich ist generierter Code nicht unbedingt performanter als eigenständig entwickelter. Dem entgegen wirken kann man etwa durch das eigenständige Anlegen von Tabellen, also dem ddl-generation Wert „*none*“.

Quellen

[1] „savorles-kap7a.ppt - savorles-kap7a.pdf“. [Online]. Verfügbar unter: http://www4.in.tum.de/~sihling/savorles/pdf/savorles-kap7a.pdf. [Zugegriffen: 14-Mai-2016].

[2] „Difference between MVC and 3-tier architecture“. [Online]. Verfügbar unter: http://www.c-sharpcorner.com/blogs/difference-between-mvc-and-3tier-architecture. [Zugegriffen: 30-Mai-2016].

[3] „ddl-generation | EclipseLink 2.5.x Java Persistence API (JPA) Extensions Reference“. [Online]. Verfügbar unter: http://www.eclipse.org/eclipselink/documentation/2.5/jpa/extensions/p_ddl_generation.htm. [Zugegriffen: 08-Juni-2016].

[4] „Instanz :: entity :: ITWissen.info“. [Online]. Verfügbar unter: http://www.itwissen.info/definition/lexikon/Instanz-entity.html. [Zugegriffen: 07-Juni-2016].

[5] A. Salvanos, *Professionell entwickeln mit Java EE 7*, 1. Aufl. Rheinwerk Computing, 2014.