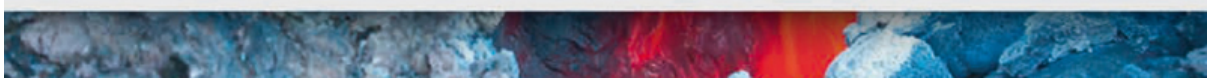
Michael Inden

Der Java-Profi: Persistenzlösungen und REST-Services

Datenaustauschformate,
Datenbankentwicklung
und verteilte Anwendungen

dpunkt.verlag



Inhalt

Cover

Titel

Impressum

Inhaltsverzeichnis

Vorwort

Kapitel 1: Einstieg in XML und JSON

1.1 Basiswissen XML

1.1.1 Bestandteile und Aufbau eines XML-Dokuments

1.1.2 Validierung eines XML-Dokuments

1.2 XML-Verarbeitung mit JAXP

1.2.1 Einfaches Parsing mit SAX

1.2.2 Komplexere Parsing-Aufgaben mit SAX

1.2.3 Parsing mit DOM

1.2.4 Verarbeiten und Speichern mit DOM

1.2.5 StAX als Alternative zu SAX oder DOM?

1.2.6 SAX, StAX oder DOM?

1.2.7 XPath im Überblick

1.2.8 XSLT im Überblick

1.2.9 XMLEncoder und XMLDecoder im Überblick

1.3 XML-Verarbeitung mit JAXB

1.3.1 Schritt 1: Passende Java-Klassen erstellen

1.3.2 Schritt 2: Marshalling und Unmarshalling

1.3.3 JAXB: Stärken und Schwächen

1.4 JAXB und StAX in Kombination

1.4.1 Rekonstruktion von Objekten mit JAXB und StAX

1.4.2 Vergleich zu SAX, DOM und JAXB

1.4.3 On-the-Fly-Modifikation von Objekten

1.5 JSON – das bessere XML?

1.5.1 Crashkurs JSON

1.5.2 JSON mit Java verarbeiten

1.5.3 JSON vs. XML

1.6 Weiterführende Literatur

Kapitel 2: Einführung in Persistenz und relationale Datenbanken

2.1 Grundlagen zur Persistenz

2.1.1 Beschränkungen einfacher Persistenzlösungen

2.1.2 Modelle zur Persistierung von Objekten

2.1.3 Speicherung von Daten in relationalen Datenbanken

2.2 Abbildung zwischen Objekt- und Datenbankmodell

2.2.1 Abbildung von Referenzen

2.2.2 Abbildung von Assoziationen und Aggregationen

2.2.3 Abbildung von Vererbung

2.3 Das Datenbanksystem HSQLDB im Kurzüberblick

2.4 SQL-Grundlagen

2.4.1 DDL – Definition von Tabellen

2.4.2 DQL – Datenabfrage

2.4.3 DML – Datenmanipulation

2.5 Ausfallsicherheit und Replikation

2.6 Weiterführende Literatur

Kapitel 3: Persistenz mit JDBC

3.1 Datenbankzugriffe per JDBC

3.1.1 Schritte zur Abfrage von Datenbanken

3.1.2 Besonderheiten von ResultSet

3.1.3 Abfrage von Metadaten

3.1.4 Probleme bei der Ausführung von Statements
3.1.5 Das Interface PreparedStatement
3.1.6 Transaktionen in JDBC
3.2 Grundlagen zum ORM mit JDBC
3.2.1 Rekonstruktion von Objekten
3.2.2 Zugriffe mit einem Data Access Object (DAO)
3.3 Weiterführende Literatur
Kapitel 4: Persistenz mit JPA
4.1 Grundlagen zum ORM und zum JPA
4.2 Einführung in JPA an einem Beispiel
4.2.1 Definition persistenter Klassen
4.2.2 Die Konfigurationsdatei persistence.xml
4.2.3 Datenbankzugriffe per JPA in Java SE
4.2.4 Lebenszyklus von Entitäten (Entity Lifecycle)
4.2.5 Datenbankmodell
4.2.6 Vorteile der konfigurativen Persistenz
4.3 JPQL im Überblick
4.3.1 Syntax von JPQL
4.3.2 Besondere Arten von Queries
4.3.3 Abfragen mit JPQL ausführen
4.3.4 Typsichere Abfragen und das Criteria API
4.4 DAO-Funktionalität mit JPA
4.4.1 CRUD-Funktionalität
4.4.2 Einsatz des DAO
4.5 Fortgeschritteneres ORM mit JPA
4.5.1 Abbildung von Assoziationen
4.5.2 Abbildung von Vererbungshierarchien
4.5.3 Verarbeitung der Typen aus JSR-310: Date and Time

4.5.4 Bean Validation im Einsatz
4.6 Transaktionen und Locking
4.6.1 Isolationslevel und Effekte
4.6.2 Problemkontext
4.6.3 Optimistic Locking
4.7 Caching in JPA
4.8 Fazit
4.9 Weiterführende Literatur
Kapitel 5: NoSQL-Datenbanken am Beispiel von MongoDB
5.1 Einführung und Überblick
5.2 Einführung MongoDB
5.2.1 Analogie von CRUD (RDBMS) zu IFUR (MongoDB)
5.2.2 Komplexere Abfragen
5.2.3 MongoDB und Transaktionen
5.3 Ausfallsicherheit und Skalierbarkeit
5.3.1 Hintergrundwissen: Formen der Skalierung
5.3.2 Ausfallsicherheit und Replica Sets
5.3.3 Skalierung und Sharding
5.3.4 Anmerkungen zu Replica Sets und Sharding
5.4 MongoDB aus Java ansprechen
5.4.1 Einführendes Beispiel
5.4.2 Daten einfügen und auslesen
5.4.3 Verarbeitung komplexerer Daten
5.4.4 Einfaches JSON-basiertes Object/Document Mapping
5.4.5 Object/Document Mapping mit Spring Data MongoDB
5.5 Fazit
5.6 Weiterführende Literatur
Kapitel 6: REST-Services mit JAX-RS und Jersey

6.1 REST im Kurzüberblick
6.1.1 Einführendes Beispiel eines REST-Service
6.1.2 Zugriffe auf REST-Services
6.1.3 Unterstützung verschiedener Formate
6.1.4 Zugriffe auf REST-Services am Beispiel von MongoDB
6.2 Ein REST-Service mit CRUD-Funktionalität
6.2.1 MIME-Types und unterschiedliche Datenformate
6.2.2 HTTP-Kommandos und CRUD-Funktionalität
6.3 Tipps zum Design von REST-Interfaces
6.3.1 Varianten der Rückgabe und Error Handling bei REST
6.3.2 Wertübergabe als @QueryParam oder @PathParam
6.3.3 Paging bei GET
6.4 Fortgeschrittene Themen
6.4.1 Einsatz von Request- und Response-Filtern
6.4.2 Security im Kontext von REST
6.4.3 Testen mit restfuse
6.5 Fazit
6.6 Weiterführende Literatur

Kapitel 7: Entwurf einer Beispielapplikation

7.1 Iteration 0: Ausgangsbasis
7.2 Iteration 1: Zentrale Verwaltung von Highscores
7.3 Iteration 2: Verwaltung von XML
7.4 Iteration 3: Bereitstellen als REST-Service
7.5 Iteration 4: Web-GUI mit HTML und JavaScript
7.6 Iteration 5: Protokollierung von Aktionen mit MongoDB
7.7 Fazit zum Abschluss der Iterationen

Anhang

A Einführung Gradle

A.1 Projektstruktur für Maven und Gradle

A.2 Builds mit Gradle

B Client-Server-Kommunikation und HTTP im Überblick

B.1 Client-Server-Kommunikation

B.2 Basiswissen HTTP

C Grundlagenwissen HTML

C.1 Basiswissen HTML

C.1.1 HTML am Beispiel

C.1.2 Interaktivität und Formulare

D Wissenswertes zu JavaScript

D.1 Grundlagen zur Sprache

D.2 Modifikation von HTML

D.3 JSON-Verarbeitung

D.4 REST-Services ansprechen

Literaturverzeichnis

Index

4 Persistenz mit JPA

In diesem Kapitel stelle ich Ihnen für die Nutzung relationaler Datenbanken (RDBMS) in Java-Applikationen das ***JPA*** (***Java Persistence API***) als Variante zu JDBC vor. Während man bei JDBC noch mit relativ wenig Abstraktion und dadurch ziemlich nah am Datenbankmodell operiert, bietet JPA eine bessere Abstraktion. Vor allem, wenn man in JDBC ein ORM (Object-Relational Mapping), also die Abbildung zwischen Objektund Datenbankwelt, realisieren möchte, erfordert dies einiges an Arbeit für die Implementierung der Datenbankzugriffe, insbesondere auch eine recht gute Kenntnis von SQL. Dagegen erleichtert JPA das ORM, weil dieses weitestgehend deklarativ mithilfe von Annotations geregelt wird. Zudem werden in JPA für Datenbankzugriffe keine SQL-Anweisungen mehr geschrieben, sondern automatisch durch JPA generiert. Dabei werden Besonderheiten verschiedener Datenbanksysteme beachtet und durch JPA unter dem einheitlichen API verborgen, sodass sie für uns als Entwickler keine Rolle mehr spielen. Insgesamt erleichtert JPA die Persistenz enorm. Schauen wir uns dies nun genauer an.

Motivation für JPA

JPA definiert einen gemeinsamen Standard für Java SE und Java EE, der viele Ideen bereits etablierter Persistenz-Frameworks sowie die damit gesammelten Erfahrungen integriert. Die Notwendigkeit der Spezifikation einer standardisierten Schnittstelle für die Persistenz in Java ergab sich, weil die zuvor genutzten Techniken JDBC (Java SE) oder Entity Beans, neuerdings nur noch Entities genannt (Java EE), aufwendig oder umständlich zu handhaben sind, besonders für komplexere Objektmodelle. Als Abhilfe wurden diverse Persistenz-Frameworks (allen voran Hibernate) entwickelt und von vielen Entwicklern anstelle von JDBC bzw. Entity Beans genutzt.[1]

Der Vorteil von JPA gegenüber einem proprietären Persistenz-Framework besteht darin, dass man sich auf einen wohldefinierten Standard abstützt. Dadurch sollte man bei Bedarf (relativ problemlos) zu einer anderen JPA-Implementierung oder einer anderen Datenbank wechseln können. Mit JDBC wird Letzteres durch die Besonderheiten proprietärer SQL-Dialekte stark erschwert, weil diese trotz der gebotenen Abstraktion immer wieder

durchscheinen und für Inkompatibilitäten bei einem Wechsel des RDBMS sorgen würden.

Vorarbeiten für JPA mit Hibernate

Bevor wir uns JPA widmen, sollte noch erwähnt werden, dass JPA nur einen Standard definiert, aber keine Implementierung vorgibt. Bekannte Realisierungen sind Hibernate und Apache OpenJPA sowie EclipseLink als Referenzimplementierung.

In diesem Buch wollen wir stellvertretend Hibernate als JPA-Implementierung nutzen. Es steht frei zum Download unter `http://hibernate.org/orm/downloads/` zur Verfügung. Am einfachsten ist es, wenn Sie die Abhängigkeiten in Ihren Mavenoder Gradle-Build einbinden. Für Gradle notieren Sie etwa Folgendes in den Dependencies:

```
// JPA
compile 'org.hibernate.javax.persistence:hibernate-jpa-2.1-
api:1.0.0.Final'

// Hibernate
compile 'org.hibernate:hibernate-entitymanager:4.3.10.Final'
```

Die transitiv benötigten Bibliotheken werden von Gradle automatisch heruntergeladen und in den `Classpath` aufgenommen. Mit der Angabe der obigen Zeilen haben wir alle Voraussetzungen erfüllt, um die folgenden Beispiele mit JPA und Hibernate nachvollziehen zu können.

4.1 Grundlagen zum ORM und zum JPA

In diesem Abschnitt werden einige Grundlagen sowohl zum ORM als auch zum JPA vorgestellt. Anschließend führe ich zentrale Begriffe zum JPA ein.

Wissenswertes zum OR-Mapping

Bekanntermaßen ist bei der Speicherung von Objekten in relationalen Datenbanken die Aufgabe zu lösen, wie Objekte auf Tabellen einer Datenbank abgebildet werden. Dabei sind zum einen Objekte und deren Beziehungen in Form von Datensätzen in Tabellen zu speichern. Zum anderen müssen aus Datensätzen auch Objekte (re-)konstruiert werden. Diese Konvertierung, genauer das bereits erwähnte ORM, lässt sich mit etwas Mühe selbst programmieren. Das wird aber umso komplexer, aufwendiger und

fehleranfälliger, je verzweigter der Objektgraph ist, d. h., je mehr Objekte und Assoziationen sowie Vererbungsbeziehungen zwischen diesen rekonstruiert werden müssen. Mögliche Probleme eines mit JDBC programmierten ORMs und daraus resultierende Aufwände zur Lösung haben wir ansatzweise beim Erstellen einiger Varianten von Zugriffsmethoden im vorherigen Kapitel über JDBC kennengelernt.

Wenn man statt »purem« JDBC alternativ JPA einsetzt, so fällt sowohl die Umsetzung von Assoziationen als auch der Einsatz objektorientierter Konzepte leichter: Man arbeitet anstelle von Datenbanktabellen mit Objekten und kann somit problemlos alle Verknüpfungen zwischen Objekten verwenden, ohne sich viele Gedanken zur Persistierung machen zu müssen: Die zur Abbildung von Objekten und ihrer Beziehungen auf Datenbanktabellen benötigten SQL-Anweisungen werden durch die jeweilige JPAImplementierung automatisch generiert. Gleiches gilt für die SQL-Anweisungen, die für Objektrekonstruktionen notwendig sind. Insgesamt entfällt eine Menge an Source-code für beide Richtungen der Abbildung. JPA versteckt neben den SQL-Anweisungen auch die Details verschiedener SQL-Dialekte und verbirgt dadurch einiges an Komplexität beim Einsatz verschiedener Datenbanken.

Manchmal kommt es allerdings sowohl beim Einsatz von Persistenz-Frameworks als auch in JPA zu unerwartetem Programmverhalten, Exceptions oder Performance-Problemen durch suboptimal generierte SQL-Kommandos. Die Performance kann leiden, wenn viele referenzierte Daten verwaltet werden. Als erste Abhilfe kann die geschickte Wahl des Zugriffs auf referenzierte Daten derlei Probleme mildern. Beispielsweise kann man bestimmen, ob referenzierte Objekte sofort (Eager Loading) oder erst bei Bedarf und tatsächlichem Zugriff (Lazy Loading) aus der Datenbank eingelesen werden sollen. Des Weiteren lässt sich dafür festlegen, wie weit Referenzierungen verfolgt und ob die dadurch referenzierten Daten geladen werden sollen.

Zudem können sich Probleme durch unerwartetes Programmverhalten und Exceptions ergeben, weil die von JPA generierten SQL-Kommandos aufgrund von Problemen im Datenbankmodell Fehler auslösen. Zur fundierten weiteren Analyse der Datenbankzugriffe ist es deshalb sinnvoll, sich die von JPA erzeugten SQL-Kommandos ausgeben zu lassen. Glücklicherweise ist dies einfach möglich.

Wichtige Bestandteile von JPA

Bevor wir uns JPA etwas genauer anschauen, werfen wir einen kurzen Blick auf ein paar wichtige Komponenten:

- **Entity** – In der Datenbank zu verwaltende Objekte werden durch eine sogenannte ***Entity*** beschrieben. Das ist ein ganz normales Java-Objekt, für das Martin Fowler die Abkürzung ***POJO*** (***Plain Old Java Object***) geprägt hat. Eine Entity ist also ein POJO, das persistent gemacht werden soll. In der Regel wird eine solche Entity auf eine Tabellenzeile einer relationalen Datenbank abgebildet (vgl. Abschnitt 2.1). Das erfordert die genaue Kenntnis des Aufbaus und führt bei selbst gestrickten Persistenzlösungen mit JDBC dazu, dass für CRUD-Operationen jeweils passende SQL-Anweisungen bereitgestellt werden müssen. Das ist in JPA nicht nötig, weil dort Änderungen durch das eigene Programm nur in der Objektwelt erfolgen, und es Aufgabe der JPA-Implementierung ist, die Änderungen mit dem Stand in der Datenbank abzugleichen.
- **Objektrelationale Metadaten** – Die Abbildung von Objekten auf Tabellen und von Attributen auf Spalten wird durch sogenannte ***objektrelationale Metadaten*** beschrieben. Die hierfür erforderlichen Einstellungen kann man mithilfe von Annotations oder aber durch Konfigurationsdateien vornehmen. In beiden Varianten wird für die Konfiguration von sinnvollen Standardeinstellungen ausgegangen, wodurch oftmals kaum Konfigurationsaufwände notwendig sind. Das ist eine große Erleichterung im Gegensatz zum direkten Einsatz von JDBC. Dort muss für Änderungen an den Attributen eines Objekts die Aktualisierung aller korrespondierenden, betroffenen Spalten in der Datenbank explizit realisiert werden. Die Vereinfachung ergibt sich insbesondere im Vergleich zur Entity-Bean-Konfiguration (J2EE), die eine Plage für jeden Entwickler ist.
- **Persistenz-Konfiguration** – Die Konfiguration der Anbindung an und die Abbildung auf die Datenbank geschieht in einer XML-Datei namens `persistence.xml`. Dort werden die Verbindungseinstellungen zur Datenbank sowie weitere Dinge konfiguriert, etwa welche Klassen persistiert werden sollen.
- **Java Persistence Query Language** – Abfragen werden in JPA nicht in SQL formuliert, sondern mit der ***Java Persistence Query Language*** (***JPQL***). Zwar ähnelt deren Syntax derjenigen von SQL-Abfragen beziehen sich aber auf Entitäten und nur indirekt auf Tabellen. Während bei SQL die Werte von

Spalten zurückgeliefert werden, geben Abfragen in JPA zu den Datenbankwerten korrespondierende Objekte zurück, nachfolgend für `Person`-Objekte gezeigt:

```
// SQL
SELECT name, vorname, geburtstag FROM Personen

// JPQL
SELECT p FROM Person p
```

Ein Vorteil ist zudem, dass die in JPQL formulierten Abfragen eine Abstraktion vom konkret verwendeten DBMS und dem dortigen Datenbankmodell schaffen.

4.2 Einführung in JPA an einem Beispiel

In diesem Unterkapitel gebe ich eine Einführung in JPA und die Persistierung von Objekten. Dabei lernen wir zunächst Entitäten kennen und entwickeln dann ein Beispiel. Dieses kann als Grundgerüst für weitere JPA-Applikationen dienen.

4.2.1 Definition persistenter Klassen

Instanzen von Klassen, die in der Datenbank gesichert werden sollen, nennt man ***Entitäten***. Entitätsklassen werden in JPA mithilfe einfacher POJOs realisiert. Zur Verarbeitung mit JPA muss eine Klasse lediglich folgende Anforderungen erfüllen:

1. Sie muss mit der Annotation `@Entity` versehen werden.[2]
2. Sie muss ein Identitätsattribut besitzen, das den Primärschlüssel darstellt.
3. Sie muss einen Konstruktor ohne Parameter (mitunter auch No-Arg-Konstruktor oder Defaultkonstruktor genannt) definieren.

Als Beispiel soll die Entitätsklasse `Person` auf die Tabelle `PersonenJPA`[3] abgebildet werden. Dazu nutzen wir zum einen die Annotation `@Entity` als Markierung zur Verarbeitung der Klasse per JPA. Durch die Annotation `@Table(name="<dbTableName>")` stellen wir eine Verknüpfung mit der korrespondierenden Tabelle her, wobei der Platzhalter `<dbTableName>` für die zu verwendende Tabelle steht. Im Beispiel nutzen wir `@Table(name="PersonenJPA")` für eine Verbindung zur Tabelle `PersonenJPA`.

Neben diesen ersten Angaben zu den objektrelationalen Metadaten kommen diverse weitere Annotations zum Einsatz: Mit `@Id` wird das Identitätsattribut gekennzeichnet. Dieses entspricht dem Primärschlüssel. Durch die Annotation `@GeneratedValue` wird der Primärschlüssel automatisch durch die Datenbank vergeben. Schließlich fehlt noch eine Verknüpfung zwischen Attributen und den Spalten der zuvor spezifizierten Tabelle. Das lässt sich einfach per Annotation `@Column(name="<dbColumnName>")` beschreiben, wobei der Platzhalter `<dbColumnName>` dem gewünschten Spaltennamen für das jeweilige Attribut entspricht. Mit diesem Wissen erstellen wir folgende Entitätsklasse:

```
@Entity
@Table(name = “PersonenJPA”)
public class Person implements Serializable
{
    @Id
    @GeneratedValue
    private Long id;

    @Column(name = “Vorname”)
    private String  firstName;

    @Column(name = “Name”)
    private String  lastName;

    @Column(name = “Geburtstag”)
    private Date    birthday;

    private Person() {}

    public Long getId()
    {
        return id;
    }

    // …
}
```

Bekanntermaßen verwenden wir in diesem Buch englische Klassennamen und deutsche Tabellennamen sowie in Englisch benannte Attribute und deutsche Spaltennamen. Weil in JPA standardmäßig angenommen wird, dass Klassennamen und Tabellennamen sowie Attributnamen und Spaltennamen übereinstimmen, wird hier eine Konfiguration nötig. Deshalb geben wir mit den Annotations `@Table` und `@Column` – wie schon im Listing genutzt – jeweils Abbildungsvorschriften in Form von Annotations an.

Info: Convention over Configuration

Für viele Frameworks und auch für Persistenzlösungen vor EJB 3 mussten diverse Konfigurationsdateien erstellt und gepflegt werden, um Einstellungen vorzunehmen. Ziel bei der auch in JPA genutzten ***Convention over Configuration*** ist es, die Konfigurationsaufwände zu reduzieren. Ohne diese müssen Einstellungen immer wiederholt angegeben werden, obwohl sie nicht vom Standardverhalten abweichen oder aus den Gegebenheiten nach einer festen Regel ableitbar sind! Legt man jedoch fest, dass eine fehlende Angabe der Standardeinstellung entsprechen oder analog zu einer Regel ermittelt werden soll, so lassen sich die Konfigurationsaufwände häufig drastisch reduzieren, da nur noch diejenigen Einstellungen angegeben werden müssen, die vom Standard abweichen.

Für die Konfiguration der Persistenz wird als Standard angenommen, dass die Namen von Klassen und Tabellen sowie von Attributen und Spalten übereinstimmen. Ist das der Fall, wird nur noch die Angabe benötigt, welche Klassen tatsächlich auf Tabellen abgebildet werden sollen. In der Praxis weichen häufig doch noch ein paar Dinge mehr vom Standard ab. Existieren z. B. mehrere Klassen gleichen Namens in verschiedenen Packages, so muss explizit eine Abbildung auf die gewünschte Tabelle spezifiziert werden. Trotzdem entsteht durch Convention over Configuration deutlich weniger Aufwand als bei einer allumfassenden Konfiguration.

Variante: Zugriffsmethoden annotieren

In JPA kann man Attribute (wie im obigen Beispiel) oder aber die korrespondierenden `get()`-Methoden annotieren. Zwar ist es möglich, diese Varianten innerhalb einer Klasse miteinander zu mischen, das sollte man aber besser vermeiden, um Fehlern und Missverständnissen vorzubeugen. Falls man trotzdem Zugriffsmethoden annotieren möchte, müssen diese den Namenskonventionen für JavaBeans genügen und die Namen dem Schema `get<Attributname>()` bzw. `set<Attributname>()` entsprechen. Demnach könnte man die Klasse `Person` wie folgt annotieren:

```
@Entity
@Table(name=“PersonenJPA”)
public class Person implements Serializable
{
    private Long id;
    private String firstName;
    private String lastName;
    private Date birthday;
```

```
    @Id
    @GeneratedValue
    public Long getId()             { return id; }

    @Column(name="Vorname")
    public String getFirstName()    { return firstName; }

    // …
}
```

Meinung: Attribute oder Zugriffsmethoden annotieren?

Ich bevorzuge die Variante der annotierten Attribute, da für mich die Deklaration bzw. Definition von Attributen noch zum eher beschreibenden Teil der Klasse gehört. Dort kann ich zentral die Annotations zum Datenbankzugriff verwenden und habe sie mit einem Blick sichtbar und nicht über den gesamten Sourcecode und damit den Realisierungsteil der Klasse verstreut.

4.2.2 Die Konfigurationsdatei `persistence.xml`

Zur Persistierung benötigen wir neben Entitätsklassen insbesondere eine Verbindung zur Datenbank. Die dafür notwendigen Informationen werden für JPA in einer XMLKonfigurationsdatei namens `persistence.xml` abgelegt. ***Diese muss in einem Verzeichnis `META-INF` liegen, das über den `Classpath` erreichbar ist.*** Die Datei selbst hat für unser Beispiel folgenden Aufbau:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence 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"
version="2.1">

    // <!- Symbolischer Name zur Identifikation ->
    <persistence-unit name="java-profi-PU" transaction-
type="RESOURCE_LOCAL">

        // <!- Hibernate als JPA-Implementierung ->
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        // <!- Voll qualifizierter Name zu persistierender Entitätsklassen
->
        <class>ch04.jpa.firstexample.Person</class>
        <exclude-unlisted-classes>true</exclude-unlisted-classes>

        // <!- Konfiguration der Datenbankanbindung ->
        <properties>
```

```
            // <!- Datenbankverbindungseinstellungen ->
            <property name="javax.persistence.jdbc.driver"
                      value="org.hsqldb.jdbcDriver" />
            <property name="javax.persistence.jdbc.user" value="sa" />
            <property name="javax.persistence.jdbc.password" value="" />
            <property name="javax.persistence.jdbc.url"
                      value="jdbc:hsqldb:hsql://localhost/java-profi" />

            // <!- Achtung: Provider-spezifisch: hier für Hibernate ->
            <property name="hibernate.dialect"
                      value="org.hibernate.dialect.HSQLDialect" />

            // <!- Ausgabe der SQL-Kommandos aktivieren ->
            <property name="hibernate.show_sql" value="true" />

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

Im Anschluss beschreibe ich die wesentlichen Bestandteile dieser Datei für Anwendungen im Kontext von Java SE. Zuvor möchte ich Sie noch darauf hinweisen, dass es für Java EE ein paar Abweichungen gibt, auf die der nachfolgende Praxishinweis eingeht.

In der Konfigurationsdatei `persistence.xml` befinden sich die zuvor für JDBC genutzten Datenbankverbindungsparameter sowie weitere Konfigurationseinstellungen durch die Angabe von `property`-Einträgen. Insbesondere sehen wir die Angabe des Persistence-Providers und der zu verwaltenden Klasse `Person` inklusive des Package-Pfads. Diese Informationen bilden eine sogenannte ***Persistence Unit*** (Eintrag `persistence-unit`). Eine solche umfasst nicht nur die Konfiguration der Datenbankverbindung, sondern vor allem auch Einträge für die Entitätsklassen. Zur späteren Referenzierung aus einer Java-Applikation besitzt eine Persistence Unit einen eindeutigen Namen, hier `java-profi-PU`.

Es mag Ihnen vielleicht so vorkommen, als ob die bisher gezeigte Realisierung des Datenbankzugriffs mit JPA aufwendiger als mit JDBC ist. Möglicherweise kann für kleinere Projekte initial mit JPA ein wenig mehr Arbeit anfallen. Das gilt allerdings schon nicht mehr, wenn man für JDBC die Verbindungseinstellungen aus Properties liest. Des Weiteren vereinfacht sich die Implementierung der Persistenz durch den Einsatz von JPA mit zunehmender Projektgröße und Anzahl zu persistierender Klassen, da dann für jede weitere Klasse lediglich ein weiterer Eintrag in der Datei `persistence.xml` notwendig wird sowie natürlich auch einige Annotations in der Klasse selbst. Nachdem Sie die initiale Konfiguration vorgenommen haben, können Sie (fast) vergessen, dass

Sie die Daten in einer Datenbank verwalten. Sie arbeiten vorwiegend auf Objekten. Eine JDBC-Anbindung erfordert dagegen für jede weitere zu persistierende Klasse immer wieder einiges an Sourcecode zur Datenbankanbindung. Schnell ist dabei ein Fehler gemacht.

Hinweis: Die Konfigurationsdatei `persistence.xml` für Java EE

Für Anwendungen im Umfeld von Java EE gibt es im Vergleich zu Java SE einige Abweichungen für die Konfigurationsdatei `persistence.xml`. Zunächst einmal betrifft dies den `transaction-type`, der für Java EE den Wert `JTA` besitzt. Zudem müssen die zu verwaltenden Klassen nicht explizit angegeben werden, sondern es werden alle im `Classpath` verfügbaren, mit `@Entity` annotierten Klassen einbezogen. Schließlich wird die Verbindung zur Datenbank nicht wie bei Java SE feingranular spezifiziert, sondern man verweist bei Java EE auf eine vom Application Server verwaltete Datasource. Details zu den beschriebenen Abweichungen finden Sie in der am Kapitelende angegebenen weiterführenden Literatur.

4.2.3 Datenbankzugriffe per JPA in Java SE

Nach Abschluss der notwendigen konfigurativen Arbeiten stelle ich Ihnen nun den Ablauf beim Zugriff auf die Datenbank per JPA vor. Das Ganze folgt gewöhnlich einem recht ähnlichen Schema – wobei der Übersichtlichkeit halber auf ein ausgeklügeltes Exception Handling verzichtet wurde. Allerdings wird die Freigabe von Ressourcen durch geeignete Aufrufe in einem `finally`-Block sichergestellt.

Grundgerüst

Zunächst schauen wir auf ein Grundgerüst, wie es für alle folgenden Beispiele immer wieder in ähnlicher Form genutzt wird, wobei lediglich die Angabe der verwendeten Persistence Unit variiert. Die zur Persistenz auszuführenden Aktionen werden in einer Methode `performJPAActions()` gekapselt. Damit ergibt sich folgende `main()`-Methode:

```
public static void main(final String[] args) throws Exception
{
    // Applikationsstart: Datei “persistence.xml” wird automatisch
eingelesen
    final EntityManagerFactory entityManagerFactory =
                        Persistence.createEntityManagerFactory(“java-
profi-PU”);
```

```
    // Zur Verarbeitung von Daten => EntityManager bietet Zugriff auf
Datenbank
    final EntityManager entityManager =
                        entityManagerFactory.createEntityManager();

    try
    {
        // Aktionen auf der DB ausführen
        performJPAActions(entityManager);
    }
    finally
    {
        // EntityManager nach Datenbankaktionen wieder freigeben
        entityManager.close();

        // Freigabe am Ende der Applikation
        entityManagerFactory.close();
    }
}
```

Die Verknüpfung des Programms mit der Datenbank wird mithilfe konkreter Realisierungen der Interfaces `EntityManagerFactory` und `EntityManager` aus dem Package `javax.persistence` geregelt. Die statische Methode `createEntityManagerFactory(String)` der Klasse `javax.persistence.Persistence` erzeugt eine `EntityManagerFactory` zum Zugriff auf die Datenbankkonfiguration. Dazu übergibt man den Namen der gewünschten Persistence Unit als Methodenparameter. Dieser muss mit den zuvor in der Datei `persistence.xml` im Element `persistence-unit` hinterlegten Konfigurationsdaten übereinstimmen – wir haben dort den Namen "`java-profi-PU`" angegeben. Da die Erzeugung einer `EntityManagerFactory`-Instanz in der Regel ein recht aufwendiger Vorgang ist, wird man dies normalerweise einmalig beim Start der Applikation durchführen. Eine `EntityManagerFactory`-Instanz stellt einen Zugriffspunkt für die Datenbank und ihre Konfiguration dar. Der eigentliche Datenbankzugriff erfolgt mithilfe von `EntityManager`-Instanzen, die persistenzbezogene Funktionalitäten anbieten, wie wir sie zuvor ähnlich mit JDBC genutzt haben.

Von JDBC kennen wir auch, dass wir zu gegebener Zeit zuvor belegte Ressourcen wieder freigeben sollten. Auch hier vereinfacht der Einsatz von JPA die Handhabung. Lediglich die schwergewichtige `EntityManagerFactory`-Instanz sowie die kürzer lebenden und leichtgewichtigen `EntityManager`-Instanzen besitzen `close()`-Methoden, die zur Freigabe der davon belegten Systemressourcen dienen. Beim `EntityManager` wird zudem die Verbindung der Entitäten zur Datenbank aufgehoben und alle bislang ungesicherten Änderungen in der Datenbank persistiert.

Auszuführende Aktionen

Als einführendes Beispiel zur Persistenz mit JPA schauen wir uns an, wie wir ein Objekt erzeugen und danach in der Datenbank speichern können. Zur Speicherung dient die Methode `persist()` des `EntityManager`s. Zudem ist es bei JPA wichtig zu wissen, dass Änderungen von Entitäten im Kontext von Transaktionen ausgeführt werden müssen – nur Lesezugriffe benötigen keinen Transaktionskontext. Zugriff auf eine Transaktion erhält man über die Methode `getTransaction()`. Ähnlich zu JDBC startet man eine Transaktion mit `begin()` und bestätigt den Abschluss durch Aufruf von `commit()` bzw. ein Rückgängigmachen per `rollback()`. Wir implementieren das Speichern einer Entität mit JPA durch Aufruf von `persist()` wie folgt:

```
private static void performJPAActions(final EntityManager entityManager)
{
    try
    {
        entityManager.getTransaction().begin();

        executeStatements(entityManager);

        entityManager.getTransaction().commit();
    }
    catch (final Exception e)
    {
        entityManager.getTransaction().rollback();
    }
}

private static void executeStatements(final EntityManager entityManager)
{
    final Person michael = new Person(“Micha-JPA”, “Inden”, new Date(71,
1, 7));
    entityManager.persist(michael);
}
```

Listing 4.1 *Ausführbar als* **’FIRSTJPAEXAMPLE’**

Viele der folgenden Beispiele basieren alle auf dem gezeigten Grundgerüst und der Transaktionsverwaltung in `performJPAActions()`. Varianten in der Implementierung findet man in der Methode `executeStatements()`, die den Anwendungscode für das jeweilige Beispiel enthält.

In diesem Fall wird dort ein `Person`-Objekt erzeugt und in der Datenbank gespeichert. Das ist zwar schon ein erster Schritt in Richtung Persistenz, jedoch fehlen uns noch die Mittel, um die typischen CRUD-Operationen auszuführen, also neben dem Erzeugen auch Daten zu suchen, zu modifizieren und zu

löschen. Als Basis lernen wir deshalb zunächst grundlegende Methoden eines `EntityManager`s kennen.

Hinweis: Abweichungen für die Beispiele bei Java EE

Die zuvor in den Listings gezeigten Aktionen wie das Ermitteln und Konstruieren der `EntityManagerFactory` sowie des `EntityManager`s, aber auch die explizite Verwaltung von Transaktionen sind für Java SE nötig. Im Kontext von Java EE wird all dies durch den Application Server erledigt (JTA-verwaltete Transaktionen). Somit wäre dort nur die Methode `executeStatements()` relevant.

Methoden eines `EntityManager`s

Bevor ich auf die wesentlichen Methoden eines `EntityManager`s eingehe, muss zunächst der Begriff ***Persistence Context*** eingeführt werden. Darunter versteht man die Menge von Instanzen der Entitätsklassen, die von einem `EntityManager` verwaltet werden. Man kann sich dies wie eine Art logischen und zeitlichen Bereich vorstellen, in dem die Werte der Objekte mit denen der Datenbank abgeglichen werden. Solange ein `EntityManager` die Objekte verwaltet, werden diese automatisch konsistent mit der Datenbank gehalten.[4] Das ist ein großer Vorteil von JPA gegenüber JDBC: ***Als Programmierer modifiziert man Objekte, und die dazu benötigten Datenbankkommandos werden automatisch durch den JPA-Provider erzeugt und zu passender Zeit ausgeführt.*** Bei Bedarf können Objekte aus der »Obhut« eines `EntityManager`s entfernt und später wieder hinzugefügt werden. Das erleichtert z B. den Datenaustausch in verteilten Applikationen zwischen Client und Server.

Im Folgenden gebe ich einen kurzen Überblick über einige praxisrelevante Funktionalitäten eines `EntityManager`s:

- `persist(Object entity)` – Speichert die Entity in der Datenbank und überführt sie in einen vom `EntityManager` verwalteten Zustand.
- `find(Class<T> entityClass, Object primaryKey)` – Sucht basierend auf dem übergebenen Primärschlüssel nach einem Datensatz in der Datenbank und gibt diesen als verwaltete Entity zurück.
- `detach(Object entity)` – Löst für die übergebene Entity deren Verbindung zum `EntityManager`.

- `merge(T entity)` – Diese Methode dient dazu, vom `EntityManager` losgelöste Instanzen wieder in den vom `EntityManager` verwalteten Persistence Context aufzunehmen.
- `remove(Object entity)` – Entfernt eine Entity aus dem Persistence Context und später aus der Datenbank (sofern nicht nochmals `persist()` aufgerufen wird).
- `refresh(Object entity)` – Gleicht die Werte der Entity mit denen aus der Datenbank ab.

CRUD-Beispiel

Nachdem wir nun einige grundlegende Methoden eines `EntityManager`s kennengelernt haben, wollen wir eine einfache CRUD-Applikation implementieren, die zwei `Person`-Objekte erzeugt, diese persistiert, dann eine Änderung ausführt und eine Person schließlich wieder löscht:

```
private static void executeStatements(final EntityManager entityManager)
{
   final Person michael = new Person(“Michael”, “Inden”, new Date(71, 1,
7));
   final Person werner = new Person(“Werner”, “Inden”, new Date(40, 0,
31));

   // CREATE
   entityManager.persist(michael);
   entityManager.persist(werner);

   // READ
   final Long id = werner.getId();
   final Person wernerFromDb = entityManager.find(Person.class, id);

   // UPDATE
   wernerFromDb.setFirstName(“Dr. h.c. Werner”);

   // DELETE
   entityManager.remove(michael);
}
```

Listing 4.2 *Ausführbar als* **’JPACRUDEXAMPLE’**

Führen wir das Programm JPACRUDEXAMPLE aus, so werden von Hibernate folgende (für das Beispiel relevante) Aktionen protokolliert:

```
Hibernate: insert into PersonenJPA (id, Geburtstag, Vorname, Name) values
(
      default, ?, ?, ?)
Hibernate: insert into PersonenJPA (id, Geburtstag, Vorname, Name) values
(
```

```
      default, ?, ?, ?)
Hibernate: update PersonenJPA set Geburtstag=?, Vorname=?, Name=? where
id=?
Hibernate: delete from PersonenJPA where id=?
```

Die SQL-Abfrage

```
SELECT * FROM PersonenJPA
```

liefert erwartungskonform folgendes Ergebnis:

```
ID Geburtstag              Vorname             Name
2  1940-01-31 00:00:00.0   Dr. h.c. Werner     Inden
```

4.2.4 Lebenszyklus von Entitäten (Entity Lifecycle)

Wir haben einen ersten Einblick in die Arbeit mit einem `EntityManager` gewonnen und gesehen, wie sich damit CRUD-Operationen realisieren lassen. Doch was bedeutet es, wenn man diese Operationen auf den Entitäten ausführt? Tatsächlich besitzen diese einen definierten Lebenszyklus und die Methoden bewirken diverse Zustandsübergänge zwischen folgenden Zuständen:

- NEW – Eine Instanz einer Entitätsklasse wurde neu erzeugt und ist noch keinem `EntityManager` zugeordnet, wird folglich auch noch nicht mit der Datenbank abgeglichen.
- MANAGED – Eine Instanz einer Entitätsklasse ist Bestandteil eines Persistence Context und wird von einem `EntityManager` verwaltet. Änderungen an der Entity werden mit der Datenbank zu bestimmten Zeitpunkten abgeglichen, insbesondere am Ende einer Transaktion.
- REMOVED – Geht eine Entity in diesen Zustand über, so wird sie später aus der Datenbank entfernt, und zwar entweder wenn `flush()` oder aber `close()` aufgerufen wird oder wenn die Transaktion erfolgreich endet.
- DETACHED – Eine Entity im Zustand detached bedeutet, dass diese nicht mehr Bestandteil des Persistence Context ist und damit auch nicht mehr vom `EntityManager` verwaltet wird.

Diese Zustände sowie mögliche Übergänge dazwischen sind in Abbildung 4-1 visualisiert.

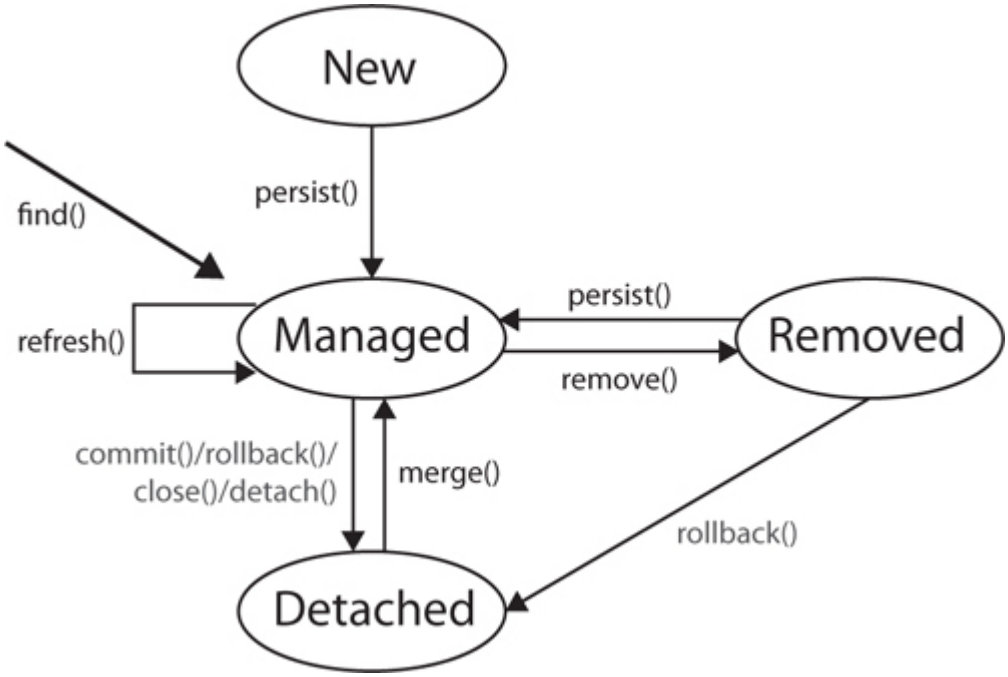


Abbildung 4-1 *Lifecycle von Entitäten in JPA*

Abgleich zwischen Datenbank und den Änderungen an einer Instanz

Zuvor erwähnte ich, dass Entitäten und die Werte der Datenbank automatisch konsistent gehalten werden, solange ein `EntityManager` die Objekte verwaltet. Das stimmt mit einer Ausnahme: Änderungen in der Datenbank durch konkurrierende Schreibzugriffe anderer Threads oder Benutzer werden nicht automatisch zum Objekt übertragen.

Für den beschriebenen Fall sind die Daten der Entitäten eventuell nicht mehr konsistent mit der Datenbank. Daher existiert die Methode `refresh(Object entity)`. Damit kann man die Attribute einer Entität mit den derzeit in der Datenbank gespeicherten Werten aktualisieren. Andersherum kann man den Abgleich der in den Instanzen der Entitätsklassen gespeicherten Werte mit der Datenbank durch einen expliziten Aufruf von `flush()` forcieren.

Der automatische Abgleich zwischen Datenbank und den Änderungen an einer Instanz einer Entitätsklasse kann durch einen Aufruf von `detach(Object entity)` temporär unterbrochen werden. Man kann sich fragen, wozu das gut sein soll, wo doch genau dieser Abgleich so praktisch ist. Darauf geht der nachfolgende Praxishinweis ein. Sollen Änderungen wieder automatisch mit der Datenbank abgeglichen werden, so kann eine zuvor abgekoppelte Instanz einer Entitätsklasse durch einen Aufruf von `merge(T entity)` als neue Entity (Rückgabe

von `merge()`) in die Verwaltung des `EntityManager`s übernommen werden – später dazu mehr.

Hinweis: Datenaustausch mithilfe von Detached Objects

In vielen Anwendungen sollen Daten über Applikationsschichten transportiert werden, etwa aus der Datenbank zum GUI. In der Regel möchte man aber nicht, dass sich im GUI vorgenommene Änderungen direkt in der Datenbank widerspiegeln. Zur Abkopplung verschiedener Schichten kann man zum Datenaustausch sogenannte ***Data Transfer Objects*** (***DTOs***) nutzen.

Der Einsatz von DTOs vermeidet, Verbindungen der Datenbankschicht bis in das GUI herauszureichen. Ähnliches kann man auch mithilfe von Entitätsklassen erzielen. Allerdings sollte man diese nicht weiterreichen, solange sie noch in Verbindung mit einer Datenbank stehen. Denn dann würden möglicherweise im GUI vorgenommene Änderungen direkt in der Datenbank reflektiert, was vermutlich nicht beabsichtigt ist. Deshalb muss die Verbindung zunächst per Aufruf von `detach(Object entity)` aufgelöst werden, um die Überwachung von Änderungen am Objekt und den Abgleich mit der Datenbank durch eine `EntityManager`-Instanz zu beenden. Man erhält dann ein sogenanntes ***Detached Object***. Dieses stellt ein POJO dar und kann problemlos an Klienten übergeben und dort sogar verändert werden, ohne dass dies Auswirkungen auf die Datenbank hat. Tatsächlich besitzt JPA auch hier Voreinstellungen, die dabei helfen, Fehler zu vermeiden. Anstatt Objekte explizit per `detach(Object entity)` aus der Obhut eines `EntityManager`s entfernen zu müssen, geschieht dies automatisch, wenn entweder der `EntityManager` per `close()` geschlossen, die Transaktion beendet oder aber eine Entity serialisiert wird. Letzteres erleichtert den entfernten Methodenaufruf sowie eine Serialisierung von Session-Daten.

4.2.5 Datenbankmodell

Interessanterweise haben wir uns bis jetzt zwar mit der Persistenz von Java-Objekten mit JPA beschäftigt, aber dabei außer über die Verbindungsparameter zur Datenbank noch in keiner Weise das zu den Entitätsklassen korrespondierende Datenbankmodell in Form von Tabellen betrachtet. Bemerkenswert ist außerdem, dass sich sowohl das Programm FIRSTJPAEXAMPLE als auch JPACRUDEXAMPLE ohne Fehler ausführen lassen und jeweils Einträge in der Datenbank in der Tabelle `PersonenJPA` erzeugen. Woher stammt das Datenbankmodell?

Bevor wir versuchen das Ganze zu verstehen, nutzen wir das in Abschnitt 2.3 beschriebene Datenbankzugriffstool von HSQLDB. Damit wagen wir einen Blick in die Datenbank. Dort sehen wir, dass folgendes Datenbankmodell erzeugt wurde. Zudem finden wir nach dem Start der Programms FIRSTJPAEXAMPLE wie erwartet eine Zeile mit folgendem Inhalt (vgl. Abbildung 4-2).

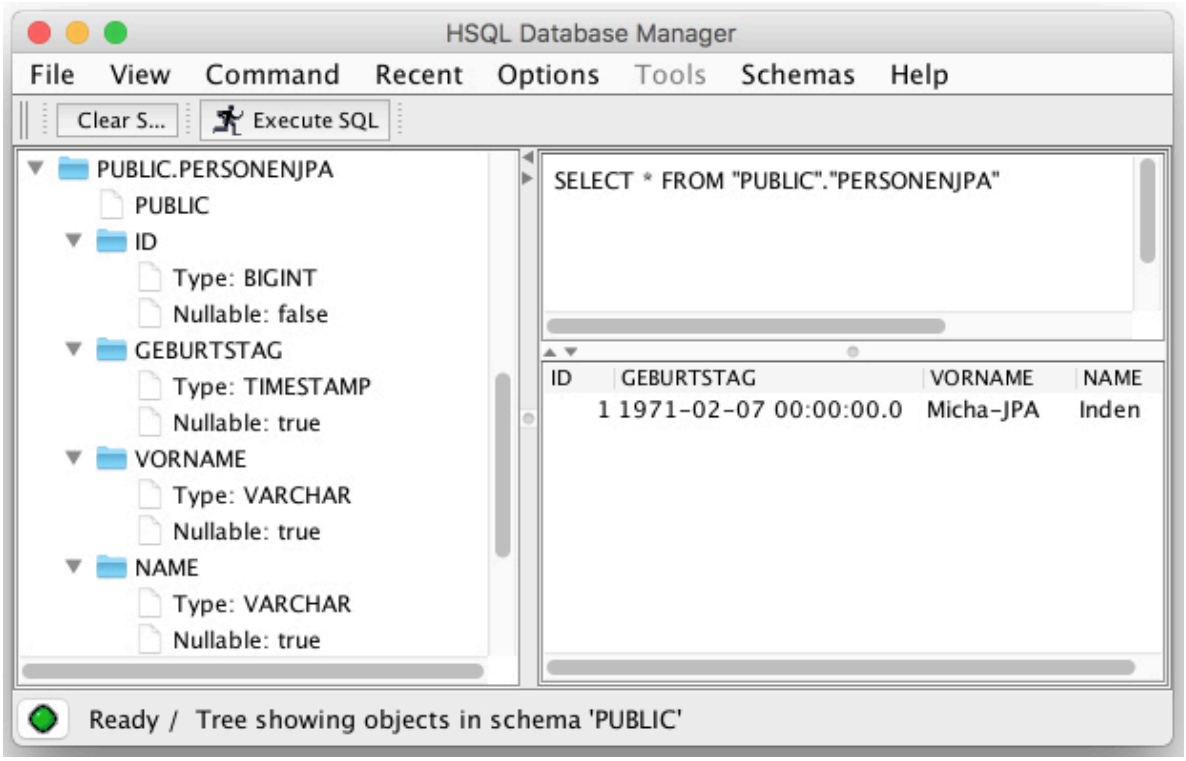


Abbildung 4-2 *Datenbankmodell von* FIRSTJPAEXAMPLE

Kommen wir zur Frage zurück, wer das Datenbankmodell erzeugt: Das geschieht durch den JPA-Provider, sofern wir in der Konfigurationsdatei `persistence.xml` über den Parameter

```
<property name="javax.persistence.schema-generation.database.action"
          value="drop-and-create"/>
```

die automatische Generierung von Tabellen eingeschaltet haben. Für die Beispiele dieses Buchs ist das sehr praktisch, in die Praxis wird man das Datenbankmodell aber nur einmalig erzeugen und bei Bedarf anpassen, wie dies in den folgenden Abschnitten kurz thematisiert wird.

Beim Start der Beispielapplikation FIRSTJPAEXAMPLE werden auf der Konsole folgende Ausgaben protokolliert, die das Erzeugen des Datenbankmodells zeigen:

```
Hibernate: drop table PersonenJPA if exists
Hibernate: create table PersonenJPA (id bigint generated by default as
identity
      (start with 1), Geburtstag timestamp, Vorname varchar (255), Name
```

```
varchar
      (255), primary key (id))
```

Erzeugung des Datenbankmodells und von DDL-Skripten

Seit JPA 2.1 ist es möglich, sich das Datenbankschema erzeugen zu lassen. Dazu gibt es folgende Aktionen:

- `none` – Das ist der Standard. Als Folge wird kein Schema generiert.
- `drop` – Bei dieser Einstellung werden die zu den Entitäten korrespondierenden Tabellen gelöscht.
- `create` – Bei Angabe von `create` werden zu den Entitäten korrespondierende Tabellen generiert. Existiert eine gleichnamige Tabelle bereits, so wird diese nicht gelöscht, sondern einfach beibehalten. Nachträgliche Änderungen im Objektaufbau werden somit möglicherweise nicht im Datenbankschema reflektiert.
- `drop-and-create` – Hier werden die Tabellen in der Datenbank zunächst gelöscht und dann neu erzeugt. Dadurch werden zwar Änderungen im schematischen Aufbau garantiert konsistent gehalten, jedoch führt auch jede Schemaänderung zu einem Löschen der Tabellen und somit zu einem Verlust der in der Datenbank gespeicherten Daten, sofern man dem nicht geeignet vorbeugt.

Für Testumgebungen, die nach Schemaänderungen immer wieder schnell und frisch aufgesetzt werden können, kann die Variante `drop-and-create` passend sein. In Produktivsystemen wird man diese Einstellung eher selten nutzen. Häufig ist selbst die Option `create` dort nicht sinnvoll, weil die Verwaltung der Datenbankschemata von speziellen Administratoren vorgenommen werden (müssen).

Tipp: Datenbankmodellgenerierung mit Hibernate

In Hibernate war schon vor JPA 2.1 eine Generierung des Datenbankmodells möglich und wurde über folgendes Property in der Datei `persistence.xml` gesteuert:

```
<property name=“hibernate.hbm2ddl.auto” value=“create” />
```

DDL-Generierung

Zusätzlich lassen sich DDL-Skripte automatisch erzeugen, was nur ein paar Einträge in der Konfigurationsdatei `persistence.xml` erfordert:

```
<property name="javax.persistence.schema-generation.scripts.action"
          value="create"/>
<property name="javax.persistence.schema-generation.scripts.create-target"
          value="createDB.sql"/>
```

Die gezeigte Ergänzung in der Konfiguration führt dazu, dass eine SQL-Datei namens `createDB.sql` mit Kommandos zum Erzeugen von Tabellen automatisch beim Programmstart generiert wird:

```
create table PersonenJPA
             (id bigint generated by default as identity (start with 1),
              Geburtstag timestamp, Vorname varchar(255), Name
varchar(255),
              primary key (id))
```

Für unser einfaches Entitätenmodell mit nur einer Klasse ist kaum ein Gewinn zu erkennen. Wenn Sie jedoch für ein umfangreiches Domänenmodell mit einer Vielzahl an zu persistierenden Klassen die korrespondierenden Datenbanktabellen anlegen müssten, kann die gerade gezeigte DDL-Generierung sehr hilfreich beim initialen Aufbau des Datenbankmodells sein.

Anmerkung zur Versionierung und Datenmigration

Sich ändernde und neue Anforderungen sind eher die Regel als die Ausnahme. Somit wird sich Ihr Datenmodell vermutlich weiterentwickeln (müssen). Aber selbst wenn keine oder kaum Änderungen zu erwarten sind, empfiehlt es sich, eine Möglichkeit zur Korrektur und anschließenden Migration des Datenmodells zu besitzen. Dazu kann man eigene Tools bauen, die inkrementelle Updates auf dem Datenmodell ausführen. Früher, als es noch keine ausgeklügelten Migrationstools gab, habe ich damit gute Erfahrungen gemacht. Heutzutage sollte man von selbst gestrickten Lösungen aber eher absehen und lieber einen Blick auf folgende Tools werfen:

- Flyway – `http://flywaydb.org/`
- Liquibase – `http://www.liquibase.org/`

4.2.6 Vorteile der konfigurativen Persistenz

Anhand dieser Einführung erkennt man, dass mit JPA eine Trennung von Zuständigkeiten erreicht wird. Es ist Aufgabe des `EntityManager`s, die Verwaltung von Objekten und den Abgleich mit der Datenbank zu übernehmen. Der

Entwickler muss in der Regel lediglich konfigurativ eingreifen. Zudem ist es möglich, Entitäten als leichtgewichtige POJOs zu implementieren und auf das Schreiben von SQL-Anweisungen (nahezu) vollständig zu verzichten. Dies ist besonders aufgrund der Unterschiede in den SQL-Dialekten erstrebenswert.

Bislang haben wir lediglich Abfragen über die ID realisiert. Für viele Anwendungsfälle ist das aber nicht ausreichend: In der Praxis sind Abfragen fast immer komplexer und umfassen eine Vielzahl an Attributen und Bedingungen, beispielsweise, wenn man alle Personen eines gewissen Alters oder mit einem bestimmten Wohnort ermitteln möchte. Wie wir auch solche Abfragen mithilfe der in JPA integrierten und an SQL angelehnten Abfragesprache JPQL formulieren, lernen wir nun.

4.3 JPQL im Überblick

Wie eingangs schon erwähnt, steht JPQL für Java Persistence Query Language und ist die JPA-spezifische Abfragesprache, deren Syntax derjenigen von SQL ähnelt. Ich habe außerdem schon darauf hingewiesen, dass JPQL auf Entitäten und nicht wie SQL auf Tabellen einer Datenbank operiert. Ähnlich wie SQL werden in JPQL auch `UPDATE`- und `DELETE`-Kommandos unterstützt, nicht jedoch das Einfügen von Daten mit `INSERT`. Letzteres erfordert, wie aus den vorangegangenen Beispielen bekannt, einen Aufruf von `persist()` durch einen `EntityManager`.

4.3.1 Syntax von JPQL

Nachfolgend zeige ich zunächst das textuelle Formulieren von Abfragen, Kommandos sowie spezielle Arten von Queries in JPQL, bevor ich in Abschnitt 4.3.3 das Ausführen thematisiere.

Einfache Abfragen mit `SELECT`

Zum Einstieg in JPQL betrachten wir die aus SQL bekannten Abfragen mit `SELECT`:

```
final String allPersonsQuery = “SELECT p FROM Person p”;
```

Analog zu SQL kann man aber auch dezidiert auf einzelne Attribute zugreifen:

```
final String allFirstNamesSortedQuery = “SELECT p.firstName FROM Person p
” +
                                        “ORDER BY p.firstName”;
```

An dem Beispiel sieht man gut, dass SQL direkt auf Datenbanktabellen und Spalten arbeitet, während JPQL sich auf Java-Klassen und deren Instanzen bezieht.

Aggregationen in Abfragen

Wie schon in SQL kann man auch in JPQL verschiedene vordefinierte Funktionen innerhalb von Abfragen nutzen. Damit lassen sich etwa Statistiken berechnen oder Texte manipulieren. Hier eine Auswahl gebräuchlicher Funktionen:

- `COUNT` – Zählt die Anzahl an Einträgen einer Abfrage.
- `MAX` / `MIN` – Ermittelt das Maximum / Minimum der Werte einer Spalte.
- `AVG` – Berechnet den Durchschnitt der Werte einer Spalte.
- `SUM` – Summiert die Werte einer Spalte.
- `TRIM` – Entfernt Whitespaces am Anfang und Ende eines Strings.
- `LENGTH` – Ermittelt die Länge eines Strings.
- `UPPER` / `LOWER` – Wandelt in Groß-/Kleinschreibung um.

Zur Verdeutlichung betrachten wir zwei Beispiele. Wir ermitteln die Anzahl an Personen und berechnen das Durchschnittsalter dieser Menge mit JPQL wie folgt:

```
final String personCountQuery = “SELECT COUNT(p) FROM Person p”;

final String personAvgAgeQuery = “SELECT AVG(p.age) FROM Person p”;
```

Letztere Query greift auf das Attribut `age` zu. Bislang enthält unsere Entitätsklasse `Person` jedoch kein solches. Man könnte annehmen, dass man dies lösen könnte, indem man eine passende Zugriffsmethode `getAge()` bereitstellt. Doch das geht nicht: Der Grund ist, dass JPQL durch den JPA-Provider in SQL umgewandelt wird. Somit müssen wir die Klasse `Person` um das Attribut `age` erweitern.

Für dieses und alle anderen Beispiele zu JPQL werden wir jeweils eine eigene Persistence Unit sowie eine eigene Tabelle für Personen nutzen. Das erfordert eine entsprechende Angabe in der Annotation `@Table`. Wir modifizieren die Klasse `Person` wie folgt:

```
@Entity
@Table(name = “PersonenJPQL”)
public class Person implements Serializable
{
```

```
    // ..

    @Column(name = “Alter”)
    private int age;

    public Person(String firstName, String lastName, Date birthday)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.birthday = birthday;
        final ZonedDateTime zdt =
ZonedDateTime.ofInstant(birthday.toInstant(),
                                                    ZoneId.systemDefault());
        this.age = (int) (ChronoUnit.YEARS.between(zdt.toLocalDate(),
                                                    LocalDate.now()));
    }

    // …
}
```

Parametrierte Abfragen mit SELECT

Abfragen gewinnen oftmals deutlich an Mehrwert, wenn gewisse Dinge flexibel gestaltet werden können. Nehmen wir z. B. an, wir wollten alle Personen ermitteln, die älter als ein bestimmtes Alter sind, oder diejenigen, die aus einer Altersgruppe stammen. Relativ naiv wäre es, für jeden benötigten Wertebereich wieder eine eigene Query zu schreiben. Selbstverständlich bietet sich hier eine Parametrierung an.

Nachfolgend möchte ich drei Varianten zeigen, wobei eine ungünstig ist und die anderen beiden fast gleichwertig. Von diesen bietet eine jedoch mehr Lesbarkeit und Verständlichkeit. Zudem lässt sich diese im Nachhinein besser erweitern und auch leichter warten. Schauen wir uns diese drei Varianten einmal an.

Variante 1: Stringkonkatenation

Die erste und simpelste, aber auch uneleganteste und fehlerträchtigste Variante besteht darin, die variablen Teile durch Stringkonkatenationen in die JPQL-Abfrage einzufügen, etwa wie folgt:

```
final String jpql1 = “SELECT person FROM Person person” +
                     ” WHERE person.age >= ” + minAge +
                     ” AND person.age < ” + maxAge;
```

Wenn Sie meine Ausführungen aus Abschnitt 3.1.4 noch im Kopf haben oder sich sonst bereits mit SQL-Injection beschäftigt haben, so sehen wir hier starke

Parallelen. Glücklicherweise kann man bei JPQL nicht ganz so viel Schindluder treiben wie bei SQL. In jedem Fall kommt es aber schnell zu Fehlern – vor allem die syntaktische Korrektheit lässt sich bei komplexeren Bedingungen schwieriger einhalten.

Variante 2: Indexbasierte Parameter

Besser als Strings zu konkatenieren ist es, in die JPQL-Query gewisse Platzhalter zu integrieren. Wie bei SQL und JDBC nutzt man in JPQL dazu das `?` als Platzhalter. Später kann für Abfragen der jeweilige Wert über die Position des Platzhalters gesetzt werden. Wir schreiben unsere Abfrage auf den Altersbereich folgendermaßen:

```
final String jpql2 = “SELECT person FROM Person person ” +
                     “WHERE person.age >= ? AND person.age < ?“;
```

Variante 3: Benannte Parameter

Diese Variante 3 der benannten Parameter ist meines Erachtens die eleganteste und am besten verständliche sowie am einfachsten erweiter- und wartbare Variante. Innerhalb der JPQL-Query werden die Parameter durch einen `:` als solche markiert:

```
final String jpql3 = “SELECT person FROM Person person ” +
                     “WHERE person.age >= :minAge AND person.age <
:maxAge“;
```

Diese Variante unterscheidet sich nur geringfügig von Variante 2, jedoch kommunizieren die benannten Parameter die Intention besser und es können auch keine Fehler in der Indexangabe der Position auftreten.

Modifikationen mit UPDATE und DELETE

Analog zu SQL kann man in JPQL auch Änderungen und Löschungen vornehmen. Die Befehle `UPDATE` und `DELETE` sind mit dem in Abschnitt 2.4.3 vermittelten Basiswissen zu SQL leicht nachvollziehbar. Exemplarisch zeige ich je eine `UPDATE`- und `DELETE`–Abfrage mit JPQL:

```
final String updateAgeCommand = “UPDATE Person p SET p.age += 10 ” +
                                “WHERE p.firstname LIKE ‘Mi%’”;

final String deleteMikeCommand = “DELETE FROM Person p ” +
                                 “WHERE p.firstname LIKE ‘Mi%’”;
```

4.3.2 Besondere Arten von Queries

Bei Queries gibt es verschiedene Sonderfälle. Im Folgenden möchte ich kurz auf Named Queries, Queries mit Objekterzeugung und native SQL-Queries in JPQL eingehen.

Named Queries

Sogenannte Named Queries erlauben es, einer Query einen Namen zu geben und die Query später von überall im Programm über ihren Namen anzusprechen. Zur Definition dient die Annotation `@NamedQuery`.

```
@Entity
@Table(name = “PersonenJPQL”)
@NamedQuery(query = “SELECT p FROM Person p WHERE p.age > :age”,
            name = “findPersonOlderThan”)
public class Person
{
    // …
}
```

Queries mit Objekterzeugung

Mitunter möchte man in einer Query nur eine Teilmenge der Attribute, etwa Name und Alter einer Person, auslesen. Das ist wie gesehen möglich, besser noch: Man kann daraus direkt neue spezialisierte Datenbehälterobjekte erzeugen. Vereinfachend nehmen wir hierzu eine Klasse `NameAndAge` an. Dann kann man mithilfe von JPA ganz einfach mit folgender Query eine Menge von `NameAndAge`-Objekten erzeugen:

```
final String createObjQuery = “SELECT NEW
ch04.jpa.jpql_queries.NameAndAge(” +
                                “p.firstName, p.age) FROM Person p”;
```

Wichtig ist hierbei, dass der Klassenname voll qualifiziert angegeben wird, weil es ansonsten zu einer Exception kommt – bei Hibernate als JPA-Provider ist dies folgende: `QuerySyntaxException: Unable to locate class [NameAndAge].`

Die Hilfsklasse `NameAndAge` besitzt zwei ihrem Namen entsprechende Attribute:

```
public class NameAndAge
{
    private final String name;
    private final int age;

    public NameAndAge(final String name, final int age)
```

```
    {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString()
    {
        return “NameAndAge [name=” + name + “, age=” + age + “]”;
    }
}
```

Native SQL-Queries

Sollte sich einmal eine gewünschte Abfrage nicht in JPQL beschreiben lassen oder falls sie dadurch nicht performant wäre, so besteht die Möglichkeit, native Queries direkt in SQL zu formulieren und auszuführen.

Nachfolgendes Listing zeigt die Definition einer Query in SQL. Dabei ist zu beachten, dass man hier auf den Tabellen der Datenbank direkt operiert, d. h., man muss die korrekten Spalten- und Tabellennamen nutzen. In unserem Beispiel sind dies `VORNAME` und `ALTER` aus der für die JPQL-Abfragen spezialisierten Tabelle `PERSONENJPQL`:

```
final String sqlString = “SELECT VORNAME, ALTER FROM PERSONENJPQL”;
```

4.3.3 Abfragen mit JPQL ausführen

Wir haben bislang verschiedene Arten von Queries textuell formuliert. Spannend wird es aber erst, wenn wir aus diesen tatsächlich Abfrageobjekte erzeugen, die dann Daten von der Datenbank abfragen. Dazu dienen die beiden Typen `Query` und `TypedQuery<T>` aus dem Package `javax.persistence`, die über `createQuery()`-Erzeugungsmethoden des `EntityManager`s konstruiert werden.

Ein Beispiel ist im Folgenden für die Abfrage der Menge aller Personen gezeigt:

```
final String findAllPersons = “SELECT p FROM Person p”;
final TypedQuery<Person> query = entityManager.createQuery(findAllPersons,
                                                           Person.class);
final List<Person> persons = query.getResultList();
```

Im Listing sehen wir den Typ `TypedQuery<T>`, dessen Resultate auf einen bei der Erzeugung der Query mitgegebenen Typ abgebildet werden. Dadurch liefert ein Aufruf von `getResultList()` in diesem Beispiel eine `List<Person>`, korrespondierend zur Angabe `Person.class` bei der Konstruktion der Query.

Tipp: Kurzform der Query

Beim Formulieren von Queries wird oftmals der vordere Teil `SELECT p` wiederholt. Zum Test habe ich diesen Teil früher einmal nicht aufgeführt. Und tatsächlich erlaubt Hibernate beide der nachfolgend gezeigten Schreibweisen:

```
final String shortQuery = “FROM Person”;
final String normalQuery = “SELECT p FROM Person p”;
```

Führen Sie doch einmal das nicht als Listing abgebildete Programm PERSONSHORTQUERYEXAMPLE aus, um zu sehen, dass beide Abfragen tatsächlich zum selben SQL führen:

```
Hibernate: select person0_.id as id1_0_, person0_.age as age2_0_,
person0_.
      Geburtstag as Geburtst3_0_, person0_.Vorname as Vorname4_0_,
person0_.
      Name as Name5_0_ from PersonenJPQL person0_
```

Ergebnisse von Queries

Sowohl `Query` als auch `TypedQuery<T>` bieten zum Ermitteln von Ergebnissen und zum Ausführen von Kommandos folgende drei Methoden:[5]

- **`getResultList()`**– Gibt eine Liste von Ergebnissen zurück. Das ist oftmals eine Liste von Entitäten. Es ist aber genauso möglich, andere Datentypen innerhalb der Liste als Rückgabe zu erhalten. Schauen wir dazu exemplarisch auf die Rückgabewerte einiger Queries für das bereits bekannte Objekt- und Datenbankmodell:

  1. `SELECT p FROM Person p` – Das Ergebnis ist eine `List<Person>`.
  2. `SELECT p.firstName FROM Person p` – Liefert eine `List<String>`.
  3. `SELECT p.firstName, p.age FROM Person p` – Man erhält als Rückgabe eine `List<Object[]>`, wobei im `Object[]` hier ein Wert vom Typ `String` und dann einer vom Typ `Integer` gespeichert sind.

  Wie schon erwähnt, sind die Ergebnisse von Abfragen in JPA in der Regel vom `EntityManager` verwaltete Entitäten. Das sind für die erste Abfrage demnach `Person`-Objekte. Für den zweiten und dritten Fall besteht die Rückgabe aus einer Liste von Vornamen und Altersangaben – diese Daten sind (selbstverständlich) nicht verwaltet. Diese Aussagen gelten analog für die nun für `getSingleResult()` gezeigten `SELECT`-Abfragen.

- **`getSingleResult()`** – Ein Aufruf von `getSingleResult()` liefert ein einzelnes Ergebnis zurück. Dies kann eine Entity sein, aber auch Werte aus Spalten oder ein `Object[]`. Auch hier schauen wir wieder auf die Rückgabewerte folgender Queries:

  1. `SELECT p FROM Person p` – Das Ergebnis ist eine `Person`.
  2. `SELECT p.firstName FROM Person p` – Liefert einen `String`.
  3. `SELECT p.firstName, p.age FROM Person p` – Man erhält als Rückgabe ein `Object[]`, wobei im `Object[]` hier ein Wert vom Typ `String` und dann ein `Integer` gespeichert sind.

  Von diesen Abfragen darf jeweils immer nur ein Ergebnis geliefert werden. Falls jedoch kein Ergebnis gefunden wird, wird eine `javax.persistence.NoResult-Exception: No entity found for query` ausgelöst. Falls mehrere Ergebnisse geliefert werden, kommt es zu einer `javax.persistence.NonUnique-ResultException: result returns more than one elements`.

- **`executeUpdate()`** – Ähnlich wie schon bei JDBC kann man durch die Methode `executeUpdate()` Kommandos absetzen und erhält als Rückgabe die Anzahl der dadurch betroffenen Zeilen. Neben JPQL-Kommandos kann man auch jede beliebige native SQL-Anweisung auf diese Weise ausführen.

Erzeugen von Beispieldaten

Für viele der nachfolgenden Beispiele dient folgende Menge von Personen als Datengrundlage:

```
private static void createPersons(final EntityManager entityManager)
{
    final Person micha = new Person(“Micha-JPQL”, “Inden”, new Date(71, 1,
7));
    final Person tim = new Person(“Tim-JPQL”, “Bötz”, new Date(84, 2,
27));
    final Person tom = new Person(“Tom-JPQL”, “Meyer”, new Date(55, 2,
31));

    entityManager.persist(micha);
    entityManager.persist(tim);
    entityManager.persist(tom);
}
```

Gebräuchliche Abfragen ausführen

Die zuvor textuell erstellten Abfragen werden hier in einer Beispielapplikation alle hintereinander ausgeführt. Zudem zeige ich sowohl UPDATE als auch DELETE. Der besseren Übersicht halber sind diese Aktionen jeweils in eigene Methoden `performQueries()` und `performUpdateAndDelete()` ausgelagert:

```
private static void performQueries(final EntityManager entityManager)
{
    // Einfache Abfrage
    final String personsQuery = "SELECT p FROM Person p";
    final TypedQuery<Person> persons =
entityManager.createQuery(personsQuery,
                                                                Person.cl
ass);
    final List<Person> allPersons = persons.getResultList();

    // Zugriff auf einzelnes Attribut
    final String firstNamesQuery = "SELECT p.firstName FROM Person p " +
                                   "ORDER BY p.firstName";
    final TypedQuery<String> firstNames =
                             entityManager.createQuery(firstNamesQuery,
                                                       String.class);
    final List<String> sortedFirstNames = firstNames.getResultList();

    // Berechnungen in Queries
    final String personCountQuery = "SELECT COUNT(p) FROM Person p";
    final Query countQuery = entityManager.createQuery(personCountQuery);
    final Object count = countQuery.getSingleResult();

    final String avgAgeQuery = "SELECT AVG(p.age) FROM Person p";
    final TypedQuery<Double> avgQuery =
                             entityManager.createQuery(avgAgeQuery,
                                                       Double.class);
    final Double avgAge = avgQuery.getSingleResult();

    // Ergebnisse ausgeben
    System.out.println(personsQuery + ":\n" + allPersons);
    System.out.println(firstNamesQuery + ":\n" + sortedFirstNames);
    System.out.println("\nCount: " + count + " / type: " +
count.getClass());
    System.out.println("Avg Age: " + avgAge);
}
```

Das Ändern und Löschen implementieren wir wie folgt:

```
private static void performUpdateAndDelete(final EntityManager
entityManager)
{
    final String updateJPQL = "UPDATE Person p SET p.firstName = 'Mike' "
+
                              "WHERE p.firstName LIKE 'Micha%'";
    final Query updateCommand = entityManager.createQuery(updateJPQL);
    final int updatedRows = updateCommand.executeUpdate();

    final String deleteJPQL = "DELETE FROM Person p WHERE p.firstName LIKE
```

```
'T%'";
    final Query deleteCommand = entityManager.createQuery(deleteJPQL);
    final int deletedRows = deleteCommand.executeUpdate();

    System.out.println("\nUpdated rows: " + updatedRows);
    System.out.println("Deleted rows: " + deletedRows);
}
```

Diese beiden Methoden werden in der Methode `executeStatements()` aufgerufen, die durch unser bereits vorgestelltes Grundgerüst – im Speziellen die Methode `performJPAActions()` – eingebettet wird:

```
private static void executeStatements(final EntityManager entityManager)
{
    createPersons(entityManager);

    performQueries(entityManager);
    performUpdateAndDelete(entityManager);
}
```

Listing 4.3 *Ausführbar als* **'PERSONJPQLQUERIESUPDATEANDDELETEEXAMPLE'**

Führt man das Programm PERSONJPQLQUERIESUPDATEANDDELETEEXAMPLE aus, so kommt es zu folgenden Ausgaben:

```
SELECT p FROM Person p:
[Person [id=1, firstName=Micha-JPQL, lastName=Inden, birthday=Sun Feb 07
      00:00:00 CET 1971, age=45], Person [id=2, firstName=Tim-JPQL,
lastName=Bötz
      , birthday=Tue Mar 27 00:00:00 CEST 1984, age=32], Person [id=3,
firstName=
      Tom-JPQL, lastName=Meyer, birthday=Thu Mar 31 00:00:00 CET 1955,
age=61]]
SELECT p.firstName FROM Person p ORDER BY p.firstName:
[Micha-JPQL, Tim-JPQL, Tom-JPQL]

Count: 3 / type: class java.lang.Long
Avg Age: 46.0

Updated rows: 1
Deleted rows: 2
```

Anhand der Ausgaben sieht man, dass durch die allgemeine Abfrage `SELECT p FROM Person p` alle drei Personen gefunden werden. Danach werden durch die Abfrage `SELECT p.firstName FROM Person p ORDER BY p.firstName` die Vornamen als geordnete Menge ermittelt. Zur Demonstration von Berechnungen zählen wir die Einträge und berechnen das Durchschnittsalter. Als Letztes wird genau der eine Datensatz mit dem Vornamen `Micha` umbenannt. Schließlich werden die beiden Personen, deren Vorname mit dem Buchstaben `T` beginnt, aus der Datenbank gelöscht.

Parametrierte Abfragen ausführen

Kommen wir nun zur Ausführung von parametrierten Abfragen. Nachfolgend zeige ich den indexbasierten Ansatz sowie denjenigen mit benannten Parametern. In beiden Fällen nutzt man eine überladene Variante der Methode `setParameter()`, um die Werte zu setzen:

```
private static void executeStatements(final EntityManager entityManager)
{
    createPersons(entityManager);

    final int minAge = 40;
    final int maxAge = 50;

    final String jpql1 = “SELECT person FROM Person person ” +
                         “WHERE person.age >= ? AND person.age < ?”;
    final TypedQuery<Person> typedQuery1 =
                             entityManager.createQuery(jpql1,
Person.class);
    typedQuery1.setParameter(1, minAge);
    typedQuery1.setParameter(2, maxAge);
    typedQuery1.getResultList().forEach(System.out::println);

    final String jpql2 = “SELECT person FROM Person person ” +
                         “WHERE person.age >= :minAge AND person.age <
:maxAge”;
    final TypedQuery<Person> typedQuery2 =
                             entityManager.createQuery(jpql2,
Person.class);
    typedQuery2.setParameter(“minAge”, minAge);
    typedQuery2.setParameter(“maxAge”, maxAge);
    typedQuery2.getResultList().forEach(System.out::println);
}
```

Listing 4.4 *Ausführbar als* **’PARAMETRIZEDQUERIESEXAMPLE’**

Führt man das Programm PARAMETRIZEDQUERIESEXAMPLE aus, so kommt es in beiden Fällen zu nachfolgender Ausgabe, weil von den drei Personen nur `Micha-JPQL` im Altersbereich 40 – 50 ist, `Tim-JPQL` ist jünger und `Tom-JPQL` älter.

```
Person [id=1, firstName=Micha-JPQL, lastName=Inden, birthday=Sun Feb 07
00:00:00
       CET 1971, age=45]
```

Spezielle Abfragen ausführen

Zum Abschluss führen wir alle besonderen Arten von Queries, also Named Queries, Queries mit Objekterzeugung und in SQL formulierte native Queries, durch Aufruf speziell dafür erstellter Methoden aus:

```
private static void executeStatements(final EntityManager entityManager)
{
    createPersons(entityManager);

    performNamedQuery(entityManager);
    performObjCreateQuery(entityManager);
    performNativeQuery(entityManager);
}
```

Listing 4.5 *Ausführbar als* **'NAMEDCONSTRUCTIONANDNATIVEQUERIESEXAMPLE'**

Dazu haben wir die ab Seite 181 entwickelten, einzelnen Funktionalitäten in kleine, überschaubare Methoden gekapselt:

```
private static void performNamedQuery(final EntityManager entityManager)
{
    final Query namedQuery =
                entityManager.createNamedQuery("findPersonsBornAfter");
    namedQuery.setParameter("birthday", new Date(77, 1, 1));

    System.out.println("Named Query");
    final List<Person> result = namedQuery.getResultList();
    result.forEach(System.out::println);
}

private static void performObjCreateQuery(final EntityManager
entityManager)
{
    final String createObjJPQL = "SELECT NEW
ch04.jpa.jpql_queries.NameAndAge" +
                                 "(p.firstName, p.age) FROM Person p";
    final TypedQuery<NameAndAge> query =
                                 entityManager.createQuery(createObjJPQL,
                                                           NameAndAge.clas
s);

    System.out.println("\nObj Create Query");
    final List<NameAndAge> result = query.getResultList();
    result.forEach(System.out::println);
}

private static void performNativeQuery(final EntityManager entityManager)
{
    final String sqlString = "SELECT VORNAME, NAME FROM PersonenJPQL";
    final Query nativeQuery = entityManager.createNativeQuery(sqlString);

    System.out.println("\nNative Query");
    final List<Object[]> rows = nativeQuery.getResultList();
    rows.forEach(row -> System.out.println(Arrays.toString(row)));
}
```

Beim Start des Programms NAMEDCONSTRUCTIONANDNATIVEQUERIESEXAMPLE wird Folgendes protokolliert:

```
Named Query
Person [id=2, firstName=Tim-JPQL, lastName=Bötz, birthday=Tue Mar 27
00:00:00
     CEST 1984, age=32]

Obj Create Query
NameAndAge [name=Micha-JPQL, age=45]
NameAndAge [name=Tim-JPQL, age=32]
NameAndAge [name=Tom-JPQL, age=61]

Native Query
[Micha-JPQL, 45]
[Tim-JPQL, 32]
[Tom-JPQL, 61]
```

Zunächst wird mit `Tim-JPQL` die Person ausgegeben, die nach dem als Parameter übergebenen Datum 1.2.1977 geboren ist. Danach werden jeweils drei Paare aus Name und Alter für alle Personen erzeugt, einmal als `NameAndAge`-Objekt und einmal durch die native SQL-Query als `Object[]`.

Begrenzungen der Anzahl der Elemente bei Abfragen

Bei umfangreichen Treffermengen ist es aufgrund der Übersichtlichkeit und Handhabbarkeit häufig wünschenswert und aus Performance-Gründen sinnvoll, nur einen Teil der Ergebnismenge zu ermitteln und an Klienten zu übertragen. Man spricht auch von ***Paging*** und meint damit die seitenweise Aufbereitung von Inhalten, z. B. für Listen mit sehr vielen Einträgen. Stellen Sie sich beispielsweise unspezifische Suchanfragen vor, etwa wenn Sie bei Google nach dem Begriff Java EE suchen. Eine solche Suche liefert schnell Millionen von Treffern. Glücklicherweise werden initial nur die ersten 10 davon angezeigt. Ist das gewünschte Ergebnis nicht darunter, besorgt man sich sukzessive die nachfolgenden Einträge, bis man den gewünschten Eintrag findet oder die Suche abbricht. Die Formulierung von Suchanfragen mit Paging ist in SQL vom jeweiligen SQL-Dialekt abhängig. Die Syntax der Abfragen ist zwar meistens ähnlich, aber doch verschieden genug, um Spezialfälle beachten zu müssen. Praktischerweise versteckt JPA diese Details vor uns.

Nehmen wir an, in einer Tabelle `Customer` sei eine umfangreiche Menge an Kunden gespeichert, von denen wir 50 Einträge ab Position 1000 ermitteln wollen:

```
final TypedQuery<Customer> query = entityManager.createQuery(“FROM
Customer”,
                                                          Customer.clas
s);
query.setFirstResult(1000);
```

```
query.setMaxResults(50);

final List<Customer> customers = query.getResultList();
```

4.3.4 Typsichere Abfragen und das Criteria API

Nachfolgend möchte ich Sie auf das Criteria API hinweisen, eine Variante zur Formulierung typsicherer Abfragen. Das Criteria API wird hier lediglich kurz anhand zweier Beispiele vorgestellt, ohne Feinheiten zu behandeln. Es geht nur darum, ein erstes Verständnis und Gefühl für die Abweichungen zur JPQL zu erhalten. Für Details verweise ich auf die weiterführende Literatur, die am Kapitelende aufgelistet ist.

Gegenüberstellung von Queries mit JPQL und Criteria API

Die bislang gezeigten Abfragen mit dem Typ `TypedQuery<T>` liefern zwar typisierte Ergebnisse, bergen aber Fallstricke bei der Formulierung von Queries. Das liegt daran, dass die Queries rein textuell beschrieben werden, wodurch sich leicht Syntaxfehler einschleichen können.[6] Darüber hinaus können Typprobleme nicht sicher vermieden werden: Das Ergebnis einer Abfrage nach Personen kann man im Sourcecode scheinbar problemlos einer Liste von Adressen zuweisen – während des Kompilierens wird dieser Fehler nicht erkannt und äußert sich erst zur Laufzeit durch eine Exception. Um derartige Probleme zu vermeiden, bietet JPA ab Version 2 das Criteria API, mit dem sich typsichere Abfragen formulieren lassen. Schauen wir uns zwei Beispiele an.

- **Berechnung des Durchschnittsalters** – In JPQL formuliert man das wie folgt:

  ```
  final String jpql = “SELECT AVG(person.age) FROM Person person”;
  ```

  Eine Umsetzung mit dem Criteria API sieht folgendermaßen aus:

  ```
  final CriteriaBuilder criteriaBuilder =
  entityManager.getCriteriaBuilder();
  final CriteriaQuery<Double> criteriaQuery =
                              criteriaBuilder.createQuery(Double.class)
  ;
  final Root<Person> person = criteriaQuery.from(Person.class);
  criteriaQuery.select(criteriaBuilder.avg(person.get(“age”)));
  ```

- **Ermittlung all jener Personen, deren Alter im Bereich von 40 bis 50 liegt** – In JPQL formuliert man dies wie folgt:

  ```
  final String jpql = “SELECT person FROM Person person ” +
                      “WHERE person.age >= 40 AND person.age < 50”;
  ```

Eine Umsetzung mit dem Criteria API sieht folgendermaßen aus:

```
final CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
final CriteriaQuery<Double> criteriaQuery =
                            criteriaBuilder.createQuery(Double.class)
;
final Root<Person> person = criteriaQuery.from(Person.class);
final Path<Integer> age = person.get(“age”);
criteriaQuery.where(criteriaBuilder.and(
                    criteriaBuilder.greaterThanOrEqualTo(age, 40),
                    criteriaBuilder.lessThan(age, 50)));
```

Wir wollen diese Queries jeweils als JPQL und mit dem Criteria API ausführen. Praktisch ist, dass sich mit dem Criteria API erstellte Abfragen wie folgt in den Typ `TypedQuery<Person>` überführen lassen:

```
final TypedQuery<Person> query = em.createQuery(criteriaQuery);
```

Beispielapplikation

Im Folgenden greifen wir auf das bereits gewonnene Wissen zu Queries zurück. Zunächst werden die bekannten drei Personen erzeugt und dann die Queries in Form der Methoden `calcAvgAge()` und `personsWithinAgeRange()` ausgeführt:

```
private static void executeStatements(final EntityManager entityManager)
{
    createPersons(entityManager);

    calcAvgAge(entityManager);
    personsWithinAgeRange(entityManager, 40, 50);
}
```

***Listing 4.6** Ausführbar als* **’PERSONJPQLANDCRITERIAAPIEXAMPLE’**

Die Implementierung der beiden Methoden orientiert sich an den zuvor dargestellten Abläufen:

```
private static void calcAvgAge(final EntityManager entityManager)
{
    final String avgAgeJPQL = “SELECT AVG(person.age) FROM Person person”;
    final TypedQuery<Double> query1 =
entityManager.createQuery(avgAgeJPQL,
                                                                Double.cla
ss);

    final CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    final CriteriaQuery<Double> criteriaQuery =
                                criteriaBuilder.createQuery(Double.class);
    final Root<Person> person = criteriaQuery.from(Person.class);
    criteriaQuery.select(criteriaBuilder.avg(person.get(“age”)));
```

```
    final TypedQuery<Double> query2 =
entityManager.createQuery(criteriaQuery);

    final Double avgAge1 = query1.getSingleResult();
    final Double avgAge2 = query2.getSingleResult();
    System.out.println("\nAvg Age 1: " + avgAge1 + " / 2: " + avgAge2);
}

private static void personsWithinAgeRange(final EntityManager
entityManager,
                                          final int minAge, final int
maxAge)
{
    final String selectJPQL = "SELECT person FROM Person person " +
                        "WHERE person.age >= :minAge AND person.age <
:maxAge";
    final TypedQuery<Person> typedQuery1 =
                            entityManager.createQuery(selectJPQL,
Person.class);
    typedQuery1.setParameter("minAge", minAge);
    typedQuery1.setParameter("maxAge", maxAge);

    final CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    final CriteriaQuery<Person> criteriaQuery =
                                criteriaBuilder.createQuery(Person.class);
    final Root<Person> person = criteriaQuery.from(Person.class);
    final Path<Integer> age = person.get("age");
    criteriaQuery.where(criteriaBuilder.and(
                        criteriaBuilder.greaterThanOrEqualTo(age, minAge),
                        criteriaBuilder.lessThan(age, maxAge)));
    final TypedQuery<Person> typedQuery2 =
                             entityManager.createQuery(criteriaQuery);

    final List<Person> results1 = typedQuery1.getResultList();
    final List<Person> results2 = typedQuery2.getResultList();
    System.out.println("Results1: " + results1);
    System.out.println("Results2: " + results2);
}
```

Startet man das Programm PERSONJPQLANDCRITERIAAPIEXAMPLE, so sieht man, dass beide Arten von Abfragen die gleichen Ergebnisse liefern:

```
Avg Age 1: 34.0 / Age 2: 34.0
Results1: [Person [id=1, firstName=Micha-JPQL, lastName=Inden,
birthday=Sun Feb
      07 00:00:00 CET 1971, age=45]]
Results2: [Person [id=1, firstName=Micha-JPQL, lastName=Inden,
birthday=Sun Feb
      07 00:00:00 CET 1971, age=45]]
```

Insgesamt wird deutlich, dass Abfragen mit dem Criteria API zwar typsicher formuliert werden können, dies aber seinen Preis hat: Die Lesbarkeit und

Verständlichkeit leidet doch – mitunter sogar stark. Vergleichen wir die beiden Ansätze kurz miteinander.

Diskussion Criteria API vs. JPQL

Abfragen mit JPQL werden als Strings formuliert, ähnlich wie dies mit SQL und JDBC geschieht. Im Gegensatz dazu bedienen sich Abfragen mithilfe des Criteria API spezieller Java-Hilfsobjekte, die Bestandteile einer Abfrage repräsentieren. Dadurch lassen sich typsichere Abfragen erstellen, die vom Compiler auf Gültigkeit geprüft werden können. Das hilft, Fehler leichter und schneller zu identifizieren, und unterstützt sichere Refactoring-Schritte.

Beim Einsatz des Criteria API sinkt meiner Meinung nach die Lesbarkeit und es ist deutlich mehr Sourcecode zu schreiben, der zudem nicht selbsterklärend ist. Daher bin ich kein allzu großer Fan des Criteria API, obwohl ich Typsicherheit und Prüfungen zur Kompilierzeit eigentlich für eine sinnvolle Sache zur Fehlervermeidung halte. Und noch ein indirekter Kritikpunkt am Criteria API: Für den Einsatz von JPQL sprechen deren gute Lesbarkeit und die Ähnlichkeit zu SQL.

Mit dieser Argumentation kann man folgenden Rat geben: Sind eher simple Abfragen zu erstellen, so liegt der Vorteil bei JPQL. Je komplexer die Abfragen werden und je dynamischer diese zusammengestellt werden sollen, desto mehr läuft man Gefahr, dabei Fehler zu machen. Dann liegt der Vorteil potenziell beim Criteria API.

4.4 DAO-Funktionalität mit JPA

Bereits bei der Besprechung des DAO-Musters in Abschnitt 3.2.2 haben wir erfahren, dass eine Kapselung von Datenbankzugriffen sinnvoll ist. Für JDBC war dies extrem wichtig, um den Rest der Applikation frei von datenbankspezifischem Sourcecode zu halten. Außerdem stellen DAOs für JDBC die ideale Stelle für den Übergang von der Datenbankwelt zur Objektwelt dar, d. h., dort sollte das ORM stattfinden. Bei JPA wird dieses vom Framework erledigt und man muss sich als Entwickler kaum Gedanken über die Realisierung des ORMs sowie eines DAOs machen. Tatsächlich wird man beim Einsatz von JPA sogar nicht unbedingt ein DAO benötigen. Allerdings erlaubt ein DAO eine saubere Trennung von Technik und Fachlichkeit.

4.4.1 CRUD-Funktionalität

Zur Verwaltung der Objekte bietet ein `EntityManager` folgende Methoden, die die CRUD-Funktionalität gewährleisten:

- CREATE – `persist(Object entity)`
- READ – `find(Class<T> entityClass, Object primaryKey)`
- UPDATE – keine direkte Entsprechung im `EntityManager`
- DELETE – `remove(Object entity)`

Erwähnenswert ist, dass für Updates keine explizite Aktion notwendig ist, weil Änderungen im Objektzustand automatisch vom `EntityManager` in der Datenbank nachgeführt werden, solange sich eine Entity in der Obhut eines `EntityManager`s befindet und nicht detached wurde.

Folgendes Listing zeigt, wie man die CRUD-Operationen, also das Erzeugen, die Suche, das Ändern und das Löschen von `Person`-Objekten, in einem DAO realisiert:

```
public final class PersonDAO
{
    private final EntityManager entityManager;

    PersonDAO(final EntityManager entityManager)
    {
        this.entityManager = entityManager;
    }

    // C - CREATE
    public long createPerson(final Person newPerson)
    {
        entityManager.persist(newPerson);
        return newPerson.getId();
    }

    // R - READ
    public Person findPersonById(final long id)
    {
        return entityManager.find(Person.class, id);
    }

    // R - READ
    public List<Person> findAllPersons()
    {
        final TypedQuery<Person> query = entityManager.createQuery("FROM
Person",
                                                                   Person.
class);
        return query.getResultList();
```

```
    }

    // U - UPDATE
    public void updatePersonFrom(final long destId, final Person
otherPerson)
    {
        final Person personInDb = findPersonById(destId);
        if (personInDb != null)
        {
            personInDb.setFirstName(otherPerson.getFirstName());
            personInDb.setLastName(otherPerson.getLastName());
            personInDb.setBirthday(otherPerson.getBirthday());
        }
    }

    // D - DELETE
    public void deletePersonById(final long id)
    {
        final Person personInDb = findPersonById(id);
        if (personInDb != null)
        {
            entityManager.remove(personInDb);
        }
    }
}
```

Bitte bedenken Sie, dass `EntityManager` nicht threadsafe sind, sodass das nun entwickelte DAO nicht aus verschiedenen Threads parallel verwendet werden darf.

An der Realisierung sieht man, dass durch das DAO nahezu nur eine Delegation an Methoden des `EntityManager`s erfolgt. Auf zwei Dinge möchte ich jedoch hinweisen:

1. Die Methode `findAllPersons()` ist eine sinnvolle Erweiterung und ermöglicht durch den Einsatz von `TypedQuery<T>` typsichere Ergebnisse bei Abfragen.
2. Der Abgleich zwischen Objekt- und Datenbankwelt wird in der Regel durch den `EntityManager` automatisch vorgenommen. Die Methode `updatePerson-From()` wird daher nur dann gebraucht, wenn man Daten aus einem anderen Objekt ähnlich wie bei einem Copy-Konstruktor übernehmen möchte. Änderungen am Objekt selbst werden ja ansonsten automatisch mit der Datenbank abgeglichen.

Allerdings ist nicht immer eindeutig, was Update leisten soll. Denkbar und oftmals sehr praktisch ist ein Abgleich mit einem Detached Object. Dieses könnte z. B. durch ein GUI verändert worden sein und diese Änderungen sollen

nun zurück in die Datenbank fließen. Dann bietet sich der Einsatz der Methode `merge()` an. Mögliche Fallstricke und Besonderheiten beleuchtet der nachfolgende Praxishinweis.

Hinweis: Wissenswertes zu `detach()` und `merge()`

In diesem Praxishinweis wollen wir uns mit Detached Objects und der Methode `merge()` beschäftigen. In einer Webapplikation oder einer verteilten Applikation könnten Entities zwischen verschiedenen Bedienschritten zwischengespeichert werden. Dazu könnte man Serialisierung nutzen – ein Webcontainer macht dies automatisch, um Session-Daten zu speichern. Das führt dazu, dass die Entity zu einem Detached Object wird, also nicht mehr vom `EntityManager` verwaltet wird.

Das bietet den Vorteil, dass Änderungen an diesem Objekt erfolgen können, ohne direkt mit der Datenbank abgeglichen zu werden, beispielsweise durch einen Wizard oder eine andere Folge von Eingabemasken. Wenn nun diese modifizierten Daten wieder mit der Datenbank abgeglichen werden sollen, kommt die Methode `merge()` ins Spiel. Diese stellt wieder eine Verbindung eines Objekts zur Datenbank her und integriert die Entity in die Obhut des `EntityManager`s. Dabei werden mögliche Datenänderungen dann in der Datenbank persistiert.

Besonderheiten bei `detach()` und `merge()`

Ein Detached Object bleibt weiterhin eine Entity, kann aber keine Zugriffe auf die Datenbank ausführen, etwa um weitere Informationen, z B. die Adressen zu einer Person, zu ermitteln. Nur diejenigen Informationen, die zum Zeitpunkt der Abkopplung vorlagen, sind im Detached-Zustand zugreifbar. Zwar kann man Attribute ändern, jedoch sollte man nicht die Id ändern, da dann das Mergen nicht mehr korrekt funktioniert. Hat sich zwischenzeitlich nicht nur das Detached Object, sondern auch etwas in dessen Repräsentation in der Datenbank geändert, so kann `merge()` diesen Konflikt nicht auflösen und führt zu einer Exception. Weil ein Merge in der Regel zu Änderungen an der Datenbank führt, muss dieser innerhalb einer Transaktion erfolgen.

Fallstrick bei `merge()` und Detached Objects

Bei Aufrufen von `merge()` muss man bedenken, dass die als Parameter übergebene Entity nicht in den Managed-Zustand versetzt wird, sondern

dass `merge()` eine neue Instanz einer Managed-Entity zurückliefert.[a] Schauen wir uns ein Beispiel an:

```
em.merge(person);
```

Man könnte meinen, dass nach dem Aufruf von `merge()` das Objekt `person` wieder eine Managed-Entity sei. Dem ist aber nicht so.

```
em.merge(person);
otherObj.setPerson(person);
em.persist(otherObj);
```

Die gezeigte Abfolge von Befehlen löst eine Exception aus, weil die Entity `person` detached ist und somit auch nicht in einer anderen Entity indirekt persistiert werden kann. Die Abhilfe ist einfach: Wie schon erwähnt, gibt die `merge()`-Methode eine neue Managed-Entity zurück, die dann wie folgt genutzt werden kann:

```
final Person newPerson = em.merge(person);
otherObj.setPerson(newPerson);
em.persist(otherObj);
```

[a] Das birgt die gleichen Fallstricke wie einige Methoden der Klasse `String`, bei denen auch nicht sofort ersichtlich ist, dass sie neue Instanzen zurückliefern, etwa `concat()`.

Anmerkungen zur Transaktionalität

Ein Detail könnte Ihnen an der Realisierung des DAOs auffallen: Wir sehen keine Transaktionen, obwohl ich doch erwähnt habe, dass Änderungen in einem Transaktionskontext ausgeführt werden müssen. Bei der Implementierung des DAOs wissen wir aber nicht, in welchem Ablaufkontext unser DAO genutzt werden soll. Deshalb sollte die Transaktionssteuerung nicht Bestandteil des DAOs sein.[7] Ansonsten würde jede einzelne CRUD-Methode als eigene Transaktion ausgeführt, was aber dem »Alles oder Nichts«-Prinzip von Transaktionen für eine Folge von Befehlen widerspricht.

4.4.2 Einsatz des DAO

Schreiben wir ein Programm, um das DAO als Abstraktion für Zugriffe auf die Tabelle `PersonenJPA_DAO` im Einsatz zu sehen. Zunächst werden zwei `Person`-Objekte erzeugt und persistiert. Abschließend löschen wir eines der beiden Objekte und führen eine Namensänderung bei dem anderen durch. Aus unseren bisherigen Beispielen wissen wir, dass die Methode `executeStatements()`

innerhalb einer Transaktion läuft, und somit ist für das dort erzeugte DAO ein Transaktionskontext gegeben.

```
private static void executeStatements(final EntityManager entityManager)
{
    // DAO erzeugen
    final PersonDAO dao = new PersonDAO(entityManager);

    // Einfügeoperationen ausführen
    final Person michael = new Person(“Micha-DAO”, “Inden”, new Date(71,
1, 7));
    final Person werner = new Person(“Werner-DAO”, “Inden”, new Date(40,
0, 31));
    final Integer michaelId = dao.createPerson(michael);
    final Integer wernerId = dao.createPerson(werner);

    // Änderungen ausführen und das Resultat prüfen
    dao.deletePersonById(michaelId);
    werner.setFirstName(“Dr. h.c. Werner”);

    final List<Person> persons = dao.findAllPersons();
    persons.forEach(System.out::println);
}
```

Listing 4.7 *Ausführbar als* **’PERSONDAOEXAMPLE’**

Blick in die Datenbank

Das Programm PERSONDAOEXAMPLE führt die schon bei der Besprechung des `EntityManagerS` genutzten Datenbankaktionen mithilfe des gerade implementierten DAOs aus. Dadurch wird in der Datenbank das in Abbildung 4-3 gezeigte Modell erzeugt. Darüber hinaus sollte es einen Eintrag in einer Tabelle `PersonenJPA_DAO` geben. Um dies zu überprüfen, können wir das in Abschnitt 2.3 vorgestellte Datenbankadministrationstool von HSQLDB nutzen.

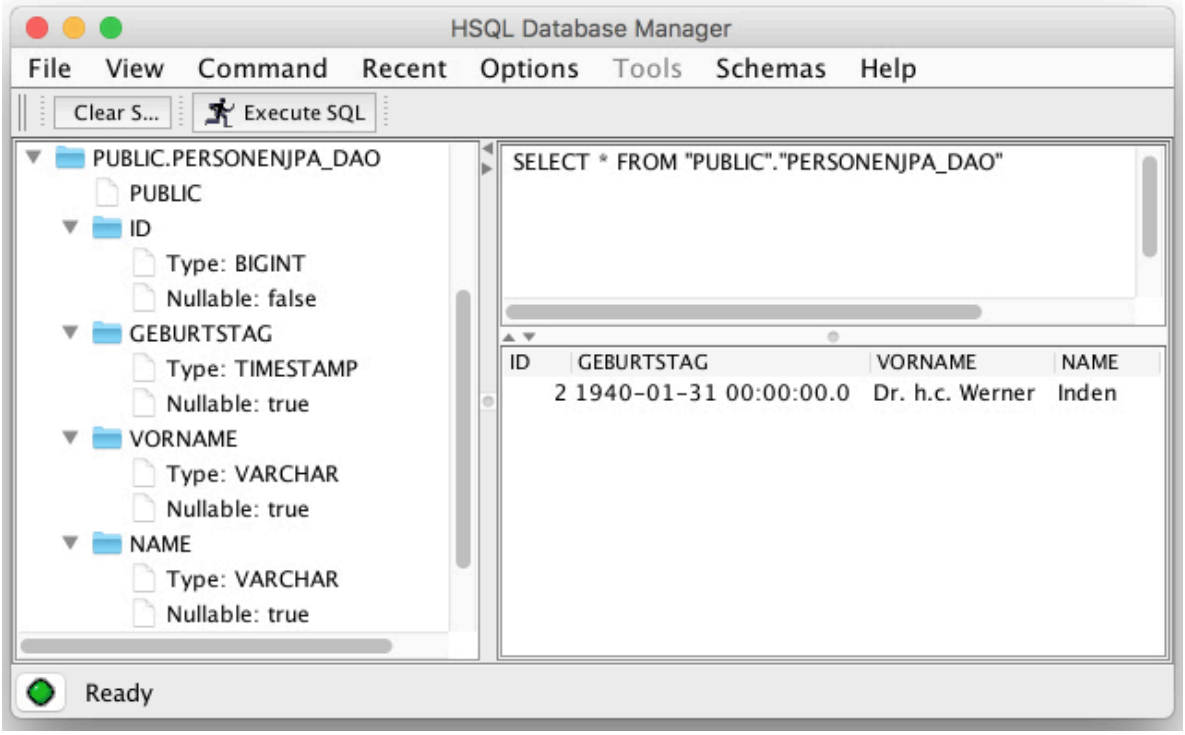


Abbildung 4-3 *Datenbankmodell für das DAO-Beispiel*

4.5 Fortgeschritteneres ORM mit JPA

Bis hierher wurden einige grundlegende Funktionalitäten und Eigenschaften des ORMs betrachtet. Die Vorteile einer Persistenzlösung mit JPA erschließen sich in ihrer Gänze jedoch erst, wenn man für die Abbildung von Assoziationen und Vererbung von den Automatiken in JPA profitiert – anstatt komplexe Anfragen und Anweisungen selbst realisieren zu müssen, wie dies bei JDBC der Fall ist.

Die folgenden Abschnitte gehen auf die Abbildung von Assoziationen und Vererbung sowie auch auf die Verarbeitung von nicht direkt durch JPA unterstützte Typen mithilfe von Konvertern am Beispiel der mit JDK 8 eingeführten Klasse `LocalDate` ein.

Die Grundlage bildet das in Abschnitt 2.2 zur Besprechung des ORMs und des Impedance Mismatch vorgestellte Klassenmodell mit Assoziationen und Vererbung. Dieses wird hier zum besseren Verständnis nochmals abgebildet.

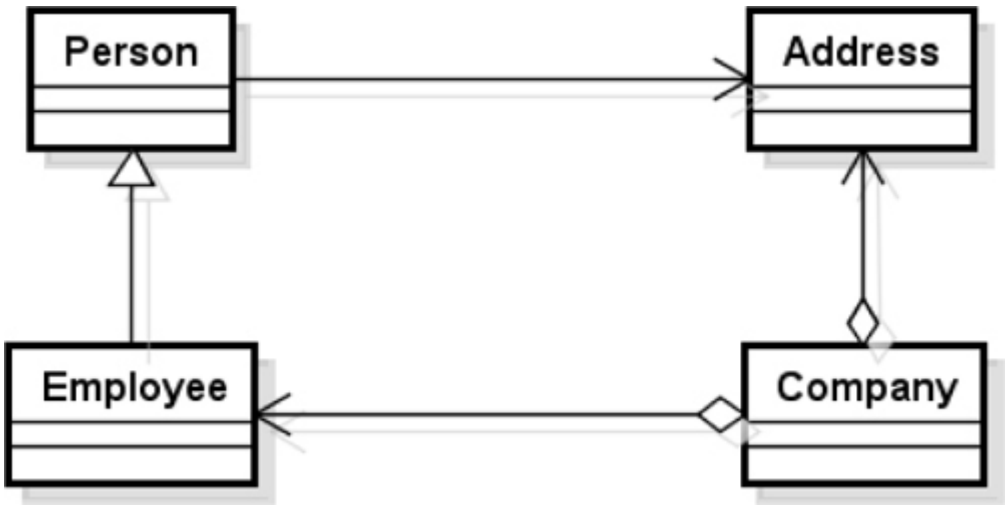


Abbildung 4-4 *Beziehungen im Objektmodell*

In den folgenden Beispielen wird jeweils nur ein für das zu betrachtende Thema relevanter Teilbereich daraus genutzt, für Assoziationen etwa die Verbindung zwischen den Klassen `Person` und `Address`.

Wie schon in den vorangegangenen Beispielen nutzen wir eigenständige Persistence Units und Datenbanktabellen. Einen Überblick gibt Tabelle 4-1.

Tabelle 4-1 *Postfixe der ORM-Beispiele*

Beispiel	Postfix für Persistence Unit / Tabellen
Assoziationen	Associations, z. B. `PersonenJPA_Associations`
Vererbung	Inheritance, z. B. `PersonenJPA_Inheritance`
Datumsverarbeitung	`PersonenJPA_DateAndTime`

Abweichend von den deutschen Tabellennamen verwende ich als Postfix die gebräuchlichen englischen Begriffe Associations und Inheritance für Assoziationen und Vererbung.

4.5.1 Abbildung von Assoziationen

Exemplarisch für verschiedene Typen von Assoziationen wird hier eine gerichtete 1:n-Abbildung von Personen auf Adressen verwendet. In der Objektwelt besitzen Personen also Adressen, aber Adressen »kennen« die Personen nicht.

Anpassungen in den Entitätsklassen

Die Klasse `Person` speichert im Attribut `addresses` eine Liste von Adressen. Auch werden in der Klasse `Person` einige Annotations genutzt. Die Annotation `@OneToMany` verwendet man für eine 1:n-Assoziation. In diesem Beispiel lernen wir noch den `CascadeType` kennen. Dieser besagt, in welcher Form die auf der Entity ausgeführten Aktionen auch auf den referenzierten Entities, hier den Adressen, ausgeführt werden. Ist der `CascadeType.PERSIST` gewählt, so werden beim Persistieren einer `Person` automatisch auch alle mit ihr verknüpften `Address`-Objekte persistiert. Ohne diese Angabe wäre zwar die `Person` in der Datenbank, nicht jedoch ihre `Address`-Objekte. Weil dies inkonsistent wäre, wird eine Exception ausgelöst – bei Hibernate als JPA-Provider ist das folgende: `org.hibernate.TransientObjectException: object references an unsaved transient instance.`

Mit diesem Vorwissen implementieren wir die Klasse `Person` nun wie folgt:

```
@Entity
@Table(name = “PersonenJPA_Associations”)
public class Person implements Serializable
{
    @Id
    @GeneratedValue
    private long id;

    @Column(name = “Vorname”)
    private String  firstName;

    @Column(name = “Name”)
    private String  lastName;

    @Column(name = “Geburtstag”)
    private Date    birthday;

    @OneToMany(cascade=CascadeType.PERSIST)
    @JoinColumn(name=“PersonenID”)
    private List<Address> addresses = new ArrayList<>();

    // …
}
```

Im Listing sind die für die Assoziation relevanten Zeilen fett markiert. An diesen sieht man, dass Assoziationen sehr einfach mithilfe von Annotations beschrieben werden. Leider lässt sich im Objektmodell nicht ganz verstecken, dass die Realisierung über Fremdschlüssel im Datenbankmodell erfolgt (`@JoinColumn(name=“PersonenID”)`).

Die Klasse `Address` ist ebenfalls als POJO implementiert und wird hier daher nicht mehr ausführlich beschrieben und auch nur ausschnittsweise abgebildet. Wissenswert ist aber, dass bei einer gerichteten Abhängigkeit (wie hier) in der referenzierten Klasse nicht einmal etwas konfiguriert werden muss:[8]

```
@Entity
@Table(name = “AdressenJPA_Associations”)
public class Address
{
    @Id
    @GeneratedValue
    private long id;

    @Column(name = “STRASSE”)
    private String street;

    @Column(name = “HAUSNR”)
    private String no;

    @Column(name = “PLZ”)
    private Integer zipcode;

    @Column(name = “STADT”)
    private String city;

    // …
}
```

Beispielprogramm

Schreiben wir ein kleines Testprogramm, das zwei `Person`-Objekte sowie drei `Address`-Objekte anlegt und diese einander wie folgt zuweist:

```
private static void executeStatements(final EntityManager entityManager)
{
    // Person-Objekte und Address-Objekte erzeugen
    final Person person1 = new Person(“Mike”, “Inden”, new Date(71, 1,
7));
    final Person person2 = new Person(“Tim”, “Bötz”, new Date(71, 2, 27));
    final Address address1 = new Address(“Street 1”, “Nr 1”, 11111, “City
1”);
    final Address address2 = new Address(“Street 2”, “Nr 2”, 22222, “City
2”);
    final Address address3 = new Address(“Street 3”, “Nr 3”, 22222, “City
3”);

    // Objekte miteinander verbinden => Assoziation
    person1.setAddresses(Arrays.asList(address1, address2));
    person2.setAddresses(Arrays.asList(address3));

    // Objekte transitiv in Datenbank speichern
```

```
    entityManager.persist(person1);
    entityManager.persist(person2);

    // Personen aus Datenbank auslesen
    final List<Person> persons =
                        entityManager.createQuery("FROM
Person").getResultList();
    persons.forEach(System.out::println);
}
```

Listing 4.8 *Ausführbar als* **'ASSOCIATIONSWITHJPAEXAMPLE'**

Konfigurationsänderungen

Bevor wir das Programm ASSOCIATIONSWITHJPAEXAMPLE starten, modifizieren wir die Konfigurationsdatei `persistence.xml` leicht. Wir fügen dort eine weitere Persistence Unit ein, die auf die zu persistierenden Klassen dieses Beispiels verweist:

```
<persistence-unit name="java-profi-PU-Associations"
                        transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <class>ch04.jpa.orm.associations.Person</class>
    <class>ch04.jpa.orm.associations.Address</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>

    <properties>
        …
    </properties>
</persistence-unit>
```

Blick in die Datenbank

Abschließend wollen wir nachvollziehen, ob die Daten wie erwartet in der Datenbank gespeichert wurden. Die Struktur der Datenbanktabellen ist ähnlich wie zuvor für das DAO und wird nicht nochmals gezeigt. Stattdessen schauen wir nur auf den Inhalt und sehen zwei Einträge in der Tabelle `PersonenJPA_Associations` und drei Einträge in der Tabelle `AdressenJPA_Associations`, die auf die Personen verweisen. Damit wird deutlich, dass die Referenzbeziehung der Java-Objekte auf eine Fremdschlüsselbeziehung der Datenbankeinträge abgebildet wird.

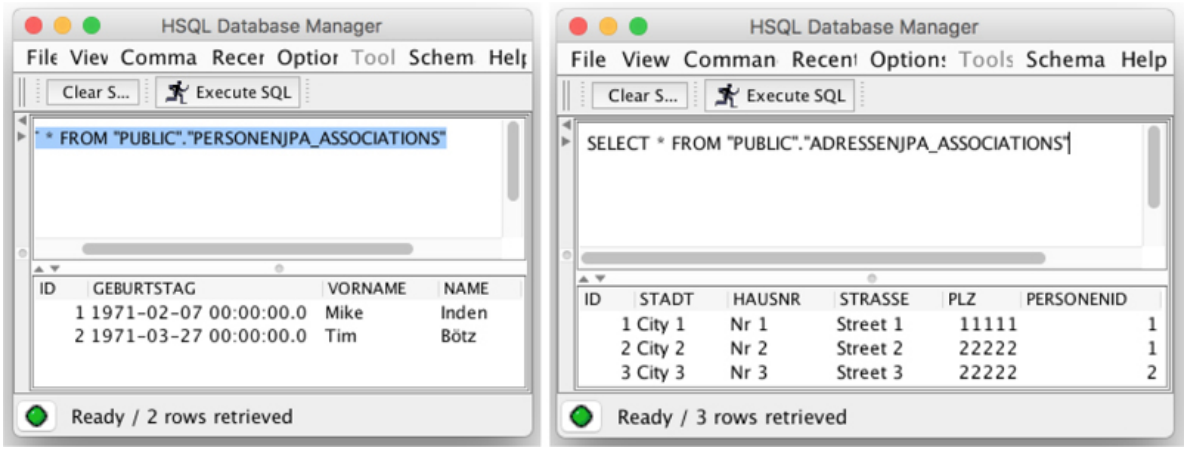


Abbildung 4-5 *Datenbankmodell für Assoziationen*

Hinweis: Modellierung von Postleitzahlen

Der Typ `Integer` für die PLZ ist für einige ostdeutsche Orte nicht valide, da diese mit einer 0 starten und die Zahl dann potenziell als Oktalzahl ausgewertet würde. Der Typ `String` oder ein eigener Domänentyp `PLZ` wären als Abhilfe denkbar.

4.5.2 Abbildung von Vererbungshierarchien

Zur Abbildung der Vererbungshierarchie eines Objektmodells auf die Tabellen eines RDBMS sind verschiedene Strategien möglich und in JPA umgesetzt. Die Thematik wurde bereits in Abschnitt 2.2.3 vorgestellt. Dort habe ich erwähnt, dass die Abbildungsverfahren spezifische Vor- und Nachteile besitzen. Daher muss eine für den jeweiligen Anwendungsfall geeignete Variante gewählt werden. In JPA sind in der `enum`-Aufzählung `javax.persistence.InheritanceType` die nachfolgend aufgeführten Konstanten definiert: `SINGLE_TABLE`, `JOINED` und `TABLE_PER_CLASS`. Zum besseren Verständnis visualisiere ich diese in jeweils eigenen Abbildungen innerhalb der nachfolgenden Aufzählung, wobei die mit Kästchen gefüllten Rechtecke den Datenbanktabellen entsprechen sollen.

- `SINGLE_TABLE` – Eine Tabelle für alle Klassen einer Vererbungshierarchie

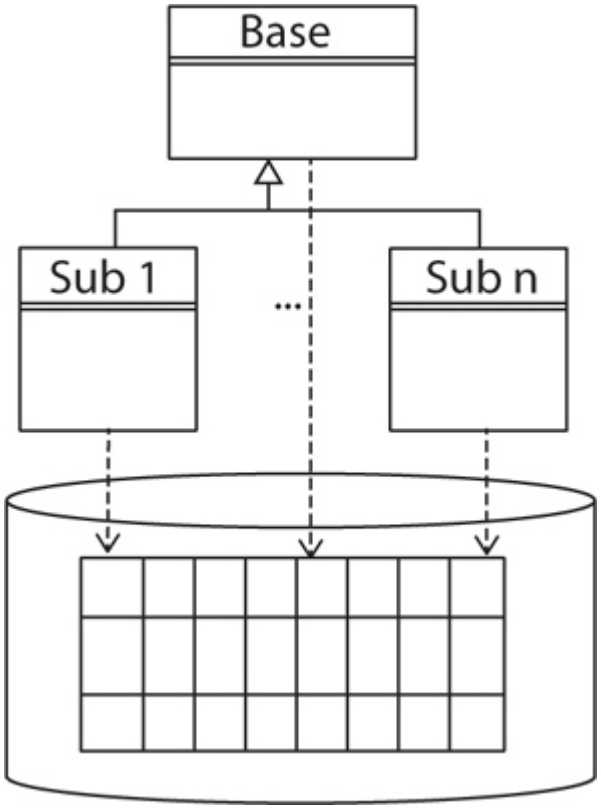


Abbildung 4-6 `SINGLE_TABLE`-*Strategie der Abbildung von Vererbung*

- `JOINED` – Jeweils eine eigene Tabelle für jede Klasse (Basis- und Subklassen)

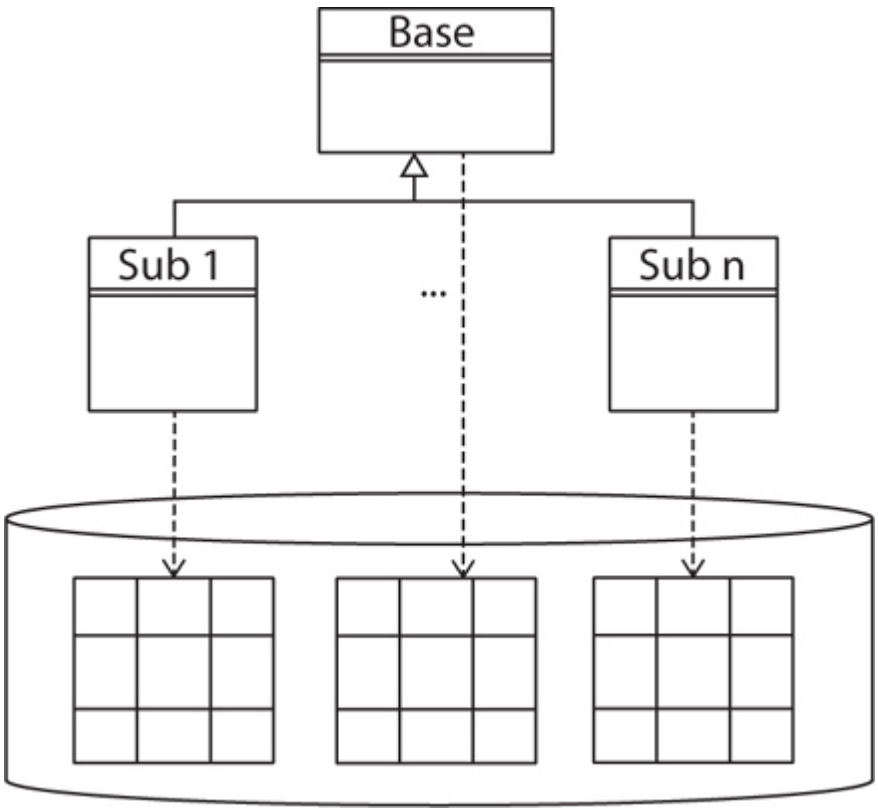


Abbildung 4-7 *JOINED-Strategie der Abbildung von Vererbung*

- TABLE_PER_CLASS – Eine Tabelle für jede konkrete Klasse

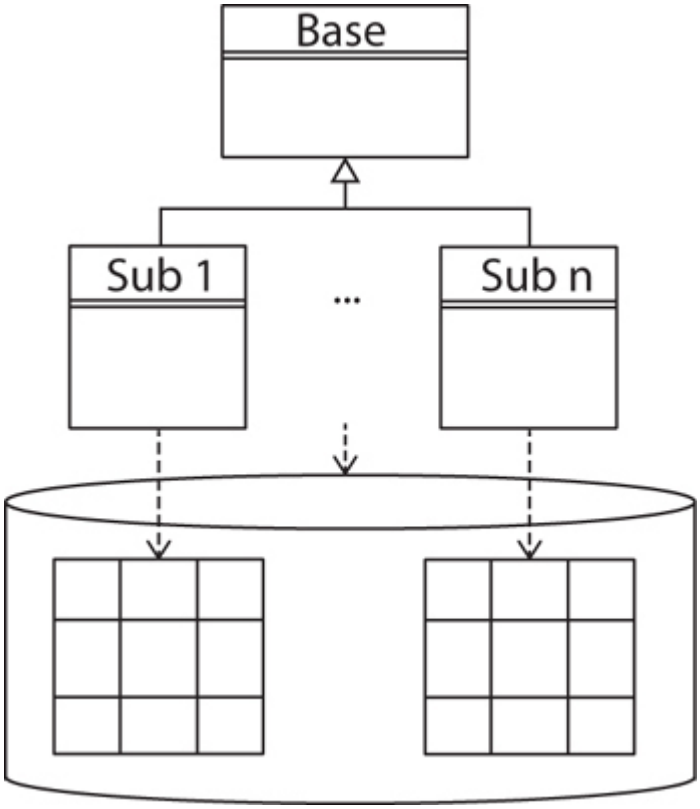


Abbildung 4-8 *TABLE_PER_CLASS-Strategie der Abbildung von Vererbung*

Anpassungen in den Klassen

Kommen wir zu dem Klassenmodell des Beispiels zurück. Dort sind Mitarbeiter einer Firma durch die Klasse `Employee`, die die Basisklasse `Person` besitzt, realisiert. Das Datenbankmodell nutzt für diese Vererbung die Strategie `JOINED` und enthält daher die zwei Tabellen `MitarbeiterJPA_Inheritance` und `PersonenJPA_Inheritance`. Zunächst schauen wir uns an, wie wir mit JPA die Klasse `Employee` implementieren. Interessanterweise beschreiben wir dort nur die spezifischen Attribute. Es wird jedoch keine Primärschlüssel-ID als Attribut benötigt, sondern aus der Basisklasse geerbt.

```
@Entity
@Table(name = “MitarbeiterJPA_Inheritance”)
public class Employee extends Person
{
    @Column(name = “MitarbeiterNr”)
    private Integer employeeNumber;

    @Column(name = “Abteilung”)
    private String workgroup;
```

```
    // JPA fordert Defaultkonstruktor
    private Employee() {}

    public Employee(final String firstName, final String lastName,
                    final Date birthday, final Integer employeeNumber,
                    final String workgroup)
    {
        super(firstName, lastName, birthday);
        this.employeeNumber = employeeNumber;
        this.workgroup = workgroup;
    }

    // …
}
```

Zur Modellierung der Vererbung mit JPA muss in der Basisklasse `Person` der gewünschte Vererbungstyp `InheritanceType.JOINED` angegeben werden – die restliche Implementierung bleibt bis auf die geänderte Sichtbarkeit des Defaultkonstruktors (bzw. genauer: No-Arg-Konstruktors) gleich: Dieser muss `protected`[9] sein, damit er aus abgeleiteten Klassen zugreifbar ist.

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name = “PersonenJPA_Inheritance”)
public class Person implements Serializable
{
    // …

    protected Person()
    {}

    // …
}
```

Fallstrick: Tiefe Vererbungshierarchien und JPA

Zwar versteckt JPA einige Details der Datenbankanbindung, jedoch wissen wir aus der allgemeinen Diskussion zur Abbildung von Vererbung in Datenbanken, dass je nach gewählter Strategie der eine oder andere Nachteil in Kauf genommen werden muss. Insbesondere wenn man recht tiefe Vererbungshierarchien im Entitätenmodell nutzt, kann dies ziemliche Performance-Probleme verursachen.

Konfigurationsänderungen

Wie schon für Assoziationen müssen wir auch für Vererbung die Konfigurationsdatei `persistence.xml` ein wenig modifizieren. Wiederum fügen

wir eine eigenständige Persistence Unit ein, diesmal mit dem Namen `java-profi-PU-Inheritence`. Dort führen wir die zu persistierenden Klassen dieses Beispiels folgendermaßen auf:

```
<persistence-unit name="java-profi-PU-Inheritence"
                transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <class>ch04.jpa.orm.inheritance.Person</class>
    <class>ch04.jpa.orm.inheritance.Employee</class>
    <class>ch04.jpa.orm.inheritance.Address</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>

    <properties>
        …
    </properties>
</persistence-unit>
```

Beispielprogramm

Nach der gezeigten Modifikation der Konfiguration machen wir uns daran, ein Testprogramm zu schreiben, das ein `Employee`-Objekt anlegt und diesem zwei `Address`-Objekte zuweist. Hier geht es darum, Objekte aus einer Vererbungsbeziehung zu persistieren, zudem wird durch die Verwaltung von Adressen auch Basisklassenfunktionalität angesprochen.

```
private static void executeStatements(final EntityManager entityManager)
{
    // Subklasse Employee erzeugen => das führt automatisch zu
    // Einträgen in den Tabellen Personen und Mitarbeiter
    final Employee newEmployee = new Employee("Mr", "Java", new Date(7, 7,
7),
                                              7777, "Entwicklung");

    // Address-Objekte erzeugen
    final Address address1 = new Address("Street 1", "Nr 1", 1111, "City
1");
    final Address address2 = new Address("Street 2", "Nr 2", 2222, "City
2");

    // Objekte miteinander verbinden => Assoziation
    newEmployee.setAddresses(Arrays.asList(address1, address2));

    // Mitarbeiter samt Adressen speichern
    entityManager.persist(newEmployee);

    // Employee-Objekt samt Adressen ausgeben
    final List<Person> employees =
                entityManager.createQuery("FROM Employee").getResultList();
```

```
    employees.forEach(System.out::println);
}
```

Listing 4.9 *Ausführbar als* **'INHERITANCEWITHJPAEXAMPLE'**

Anhand der Beispiele sehen wir, dass sich mit JPA Vererbungsbeziehungen ebenso wie Assoziationen einfach auf Datenbanktabellen abbilden lassen, ohne dass man dafür SQL-Anweisungen schreiben muss.

Wenn wir das Programm INHERITANCEWITHJPAEXAMPLE ausführen, dann wird zunächst das erzeugte `Employee`-Objekt in der Datenbank persistiert. In einem zweiten Schritt werden auch die Adressen in der Datenbank gespeichert, wie es folgende Konsolenausgaben belegen:

```
Employee [employeeNumber=7777, workgroup=Entwicklung,
super.toString()=Person [
      id=1, firstName=Mr, lastName=Java, birthday=Wed Aug 07 00:00:00 CET
1907, addresses=[Address [id=1,
      street=Street 1, no=Nr 1, zipcode=1111, city=City
       1], Address [id=2, street=Street 2, no=Nr 2, zipcode=2222,
city=City 2]]]]
```

Blick in die Datenbank

Wir wollen nun noch nachvollziehen, wie die Daten in der Datenbank gespeichert wurden. Für die Referenzierung von Adressen haben wir das bereits für das vorherige Beispiel nachvollzogen. Hier schauen wir nur noch auf die Details der Vererbungshierarchie. Demnach müsste es einen Eintrag in der `PersonenJPA_Inheritance`-Tabelle und eine Ergänzung dazu in der Tabelle `MitarbeiterJPA_Inheritance` geben.

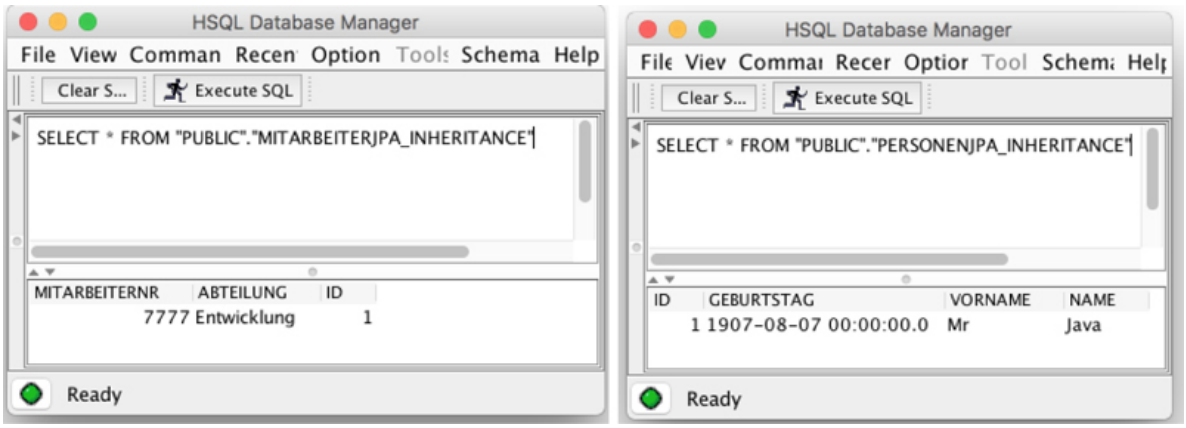


Abbildung 4-9 *Datenbankmodell für Vererbung*

4.5.3 Verarbeitung der Typen aus JSR-310: Date and Time

Das mit Java 8 neu eingeführte Date and Time API stellt einen Meilenstein für die Datumsarithmetik dar und erleichtert diese fundamental. Leider bietet JPA selbst in der aktuellen Version 2.1 keine direkte Unterstützung für die Persistierung der Typen des Date and Time API. Glücklicherweise gibt es aber seit JPA 2.1 sogenannte `javax.persistence.AttributeConverter`, mit deren Hilfe sich auch von JPA nicht direkt unterstützte Klassen persistieren lassen.

Definition von Konverterklassen

Für die Klasse `java.time.LocalDate` könnte man wie nachfolgend gezeigt einen `AttributeConverter` schreiben, der auf der Klasse `Date` beruht:

```
@Converter(autoApply = true)
public class LocalDateConverter implements
                                AttributeConverter<LocalDate, Date>
{
    @Override
    public Date convertToDatabaseColumn(final LocalDate date)
    {
       if (date != null)
       {
           final Instant instant = date.atStartOfDay().
                                     atZone(ZoneId.systemDefault()).toIns
tant();
           return Date.from(instant);
       }
       return null;
    }

    @Override
    public LocalDate convertToEntityAttribute(final Date value)
    {
       if (value != null)
       {
           final Instant instant = Instant.ofEpochMilli(value.getTime());
           return LocalDateTime.ofInstant(instant,
                                ZoneId.systemDefault()).toLocalDate();
        }
        return null;
    }
}
```

Die gezeigte Realisierung ist zwar relativ kurz, jedoch erfordert sie schon einiges an Spezialwissen bezüglich der neuen Klassen und darüber, mit welchen Zeitzonen-Offsets man arbeitet. Auch der Umweg über die Klassen `Instant` und `LocalDateTime` ist nicht wirklich selbsterklärend. Wie kann man es besser machen?

Tatsächlich ist es in diesem Fall sehr einfach durch eine textuelle Darstellung möglich. Die neuen Zeittypen besitzen standardisierte Stringrepräsentationen und können aus solchen problemlos rekonstruiert werden. Das nutzen wir bei der Implementierung der verbesserten Version unseres Konverters basierend auf `String` wie folgt:

```
@Converter(autoApply = true)
public class LocalDateConverter implements
                                AttributeConverter<LocalDate, String>
{
    @Override
    public String convertToDatabaseColumn(final LocalDate date)
    {
        if (date == null)
        {
            return null;
        }
        return date.toString();
    }

    @Override
    public LocalDate convertToEntityAttribute(final String value)
    {
        if (value == null)
        {
            return null;
        }
        return LocalDate.parse(value);
    }
}
```

Ähnliche Konverter können für die Typen `java.sql.Date`, `java.sql.Time` und `java.sql.Timestamp` definiert werden. Dazu kann man auf die drei Methoden `Date.valueOf(LocalDate)`, `Time.valueOf(LocalTime)` und schließlich noch `Timestamp.valueOf(LocalDateTime)` zurückgreifen.

Änderungen in den Entitäten

Interessanterweise können unsere Entitäten unverändert bleiben, allerdings nur dann, wenn wir Konverter für spezielle Typen schreiben, wie hier für `LocalDate`, und zudem im Konverter `autoApply = true` setzen. Dadurch klinkt sich der JPA-Provider automatisch in die Verarbeitung ein, wenn er einen Typ findet, für den ein Konverter wie oben definiert ist.[10]

Manchmal möchte man Konverter auch explizit einsetzen, etwa wenn verschiedene Darstellungen eines Zahlenwerts oder einer Farbe erfolgen sollen.

Für diese Fälle lässt sich der für ein Attribut zu verwendende Konverter explizit wie folgt spezifizieren:

```
@Column
@Convert(converter = LocalDateConverter.class)
private LocalDate birthday;

@Column
@Convert(converter = EyeColorConverter.class)
private Color eyeColor;
```

Beispielapplikation

Mit diesem Wissen ändern wir unsere Datenklasse `Person` so, dass sie nun als Attribut des Geburtstags ein `LocalDate` speichert. In einer eigenständigen Persistence Unit verweisen wir auf diese Klasse und implementieren dann die aus den vorangegangenen Beispielen hinlänglich bekannte Methode `executeStatements()` wie folgt:

```
private static void executeStatements(final EntityManager entityManager)
{
    final Person michael = new Person(“Micha-Date”, “Inden”,
                                      LocalDate.of(1971, 2, 7));
    final Person werner = new Person(“Werner-Date”, “Inden”,
                                     LocalDate.of(1940, 1, 31));

    entityManager.persist(michael);
    entityManager.persist(werner);
}
```

Listing 4.10 *Ausführbar als* **’PERSONDATEANDTIMEEXAMPLE’**

Blick in die Datenbank

Startet man das Programm PERSONDATEANDTIMEEXAMPLE, so werden zwei Personen in der Datenbank gespeichert (vgl. Abbildung 4-10).

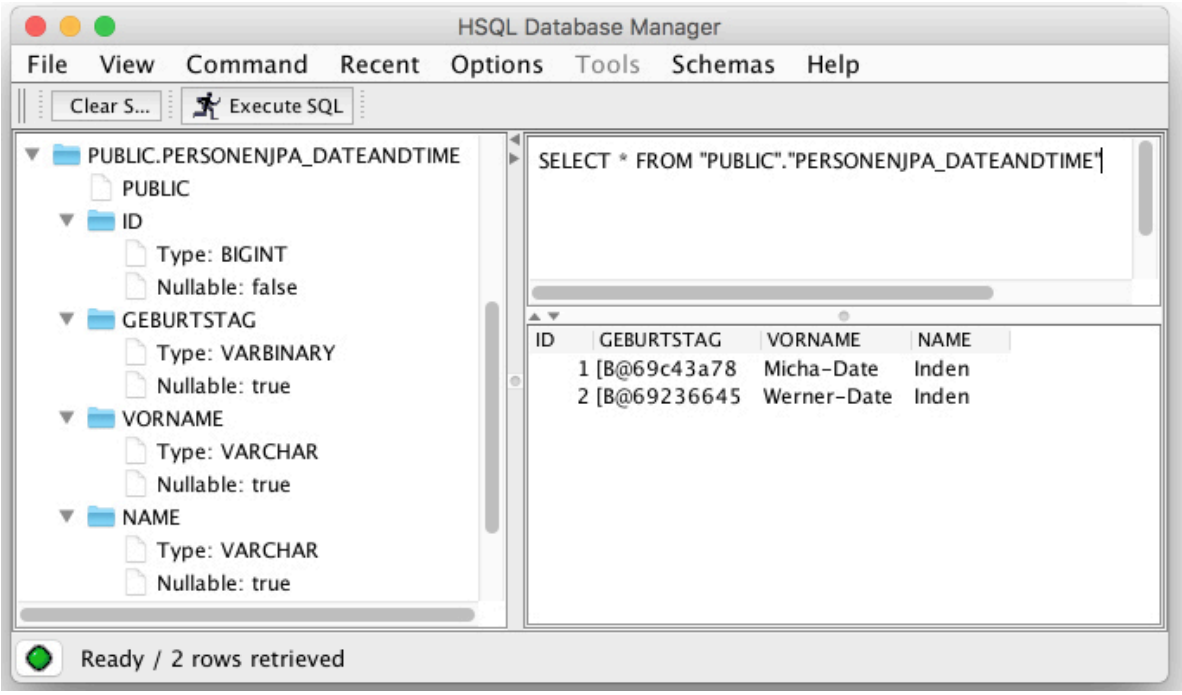


Abbildung 4-10 *Datenbankmodell mit* `LocalDateConverter`

Werfen wir einen Blick in die Datenbank: Dabei könnte der Typ und die Ausgabe der Geburtstagsspalte etwa überraschen. Die Umwandlung ins Datenbankmodell variiert zwischen verschiedenen JPA-Providern – Hibernate wählt hier ein nicht gut menschenlesbares `VARBINARY`, EclipseLink verwendet `VARCHAR` und eine textuelle Darstellung des Datums.

4.5.4 Bean Validation im Einsatz

Bislang haben wir eine Sache noch außen vor gelassen, nämlich die Gültigkeitsprüfung der Daten. Diese umfasst zum einen, dass bestimmte konkrete Bedingungen wie `@Null`, `@NotNull`, `@AssertTrue`, `@AssertFalse` gelten oder nur Werte eines gewünschten Wertebereichs (`@Max`, `@Min`) erlaubt sind, und zum anderen, dass gewisse Längenbeschränkungen eingehalten werden (`@Size`).[11] Letzteres gilt z. B. für textuelle Attribute. In der Java-Welt besitzen Strings keine fixe Maximallänge, jedoch werden die korrespondierenden Spalten im Datenbankmodell mit einer fixen Größe angelegt.

Wie können wir gewisse Prüfungen ausführen? Dazu gibt es die Bean Validation, deren Referenzimplementierung der Hibernate Validator ist, den man durch folgende Angaben in der Datei `build.gradle` einbindet:

```
compile 'org.hibernate:hibernate-validator:5.2.4.Final'
compile 'javax.el:javax.el-api:2.2.4'
```

Validierung mit Bean Validation

Oftmals gibt es Randbedingungen für Werte, die eingehalten werden müssen, so soll beispielsweise ein Name nicht kürzer als 5 und nicht länger als 20 Zeichen sein. Zudem darf dieser Wert niemals `null` werden.

Praktischerweise erlaubt es die Bean Validation, gewisse Regeln in unseren Domänenklassen per Annotations anzugeben. Machen wir es etwas konkreter und schauen uns ein Beispiel einer Entitätsklasse `Customer` und einige Prüfungen an:

```
…
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@Entity
@NamedQuery(name = “findAllCustomers”, query = “SELECT c FROM Customer c”)
public class Customer
{
    @Id
    @GeneratedValue
    private Long id;

    @NotNull
    @Size(min = 5, max = 20)
    private String name;

    public Customer()
    {}

    public Customer(final String name)
    {
        this.name = name;
    }

    // …
}
```

Nachdem wir nun wissen, wie wir Bedingungen formulieren können, müssen wir uns noch kurz Gedanken machen, wie diese geprüft und sichergestellt werden.

Ausführung der Validierung

Tatsächlich geschieht dies immer dann automatisch, wenn die Methode `persist()` aufgerufen wird. So kann noch vor der Speicherung in der Datenbank dafür gesorgt werden, dass die Daten konsistent und valide sind.

Meinung: Prüfung in Domänenklassen

Ob eine Prüfung in Domänenklassen erfolgen sollte, darüber gibt es unterschiedliche Ansichten. Ich finde es aber hilfreich, wenn die Domänenklassen ihre Constraints kennen und prüfen können. Neben den eher technischen durch das Datenbankmodell vorgegebenen Beschränkungen wie einer Maximallänge eines Namens gibt es diverse andere Bedingungen, die sich aus der Fachlichkeit ergeben, etwa die Wertebereiche für Postleitzahlen.

Validation am Beispiel eines Unit Test

Wir wollen nun die Einhaltung der Längenbeschränkung für das Namensattribut der obigen Klasse `Customer` prüfen. Um auch den Aspekt des Testens zu betrachten, wählen wir einen Unit Test mit folgenden drei Methoden, die eine gültige Länge sowie eine Unter- und Überschreitung der Grenzen prüfen:

```
@Test
public void validCustomer()
{
    final Customer customer = new Customer(“Michael”);
    em.persist(customer);

    final TypedQuery<Customer> query =
em.createNamedQuery(“findAllCustomers”,
                                                        Customer.class)
;
    assertEquals(1, query.getResultList().size());
}

@Test
public void nameTooShort()
{
    try
    {
        em.persist(new Customer(“One”));

        fail(“Expected ConstraintViolationException, but not thrown.”);
    }
    catch (final ConstraintViolationException ex)
    {
        final List<ConstraintViolation<?>> violations =
getViolationsFrom(ex);
        assertEquals(1, violations.size());

        final ConstraintViolation<?> firstViolation = violations.get(0);
        assertEquals(“name”, firstViolation.getPropertyPath().toString());
    }
}

@Test
```

```
public void nameTooLong()
{
    try
    {
        em.persist(new Customer(“ABCDEFGHIJKLMNOPQRSTUVWXYZ”));

        fail(“Expected ConstraintViolationException, but not thrown.”);
    }
    catch (final ConstraintViolationException ex)
    {
        final List<ConstraintViolation<?>> violations =
getViolationsFrom(ex);
        assertEquals(1, violations.size());

        final ConstraintViolation<?> firstViolation = violations.get(0);
        assertEquals(“name”, firstViolation.getPropertyPath().toString());
    }
}

private List<ConstraintViolation<?>> getViolationsFrom (
                                    final ConstraintViolationException
ex)
{
    return new ArrayList<>(ex.getConstraintViolations());;
}
```

Aus der Lektüre der vorherigen Abschnitte wissen wir, dass schreibende Aktionen eines `EntityManager`s im Kontext von Transaktionen stattfinden müssen. Wir nutzen einen kleinen Trick. JUnit erlaubt es, gewisse Aktionen vor dem Start eines Tests (`@BeforeClass`) und nach dessen Ende (`@AfterClass`) auszuführen. Damit erzeugen wir einmalig eine `EntityManagerFactory` und geben diese schließlich wieder frei:

```
public class CustomerTest
{
    private static EntityManagerFactory emf;

    private EntityManager em;

    @BeforeClass
    public static void createEntityManagerFactory()
    {
       emf = Persistence.createEntityManagerFactory(“java-profi-PU-
VALIDATION”);
    }

    @AfterClass
    public static void closeEntityManagerFactory()
    {
       emf.close();
    }
```

```
    // …
}
```

Was nun noch fehlt, ist die Transaktionsverwaltung für jede Testmethode. Diese realisieren wir mithilfe der Annotations `@Before` und `@After`. In den derart annotierten Methoden wird ein `EntityManager` pro Ausführung einer mit `@Test` annotierten Testmethode initialisiert und eine neue Transaktion gestartet. Nach getaner Arbeit wird die Transaktion abgebrochen und Ressourcen freigegeben. Diesen Rahmenablauf implementieren wir folgendermaßen:

```
public class CustomerTest
{
    // …

    @Before
    public void beginTransaction()
    {
        em = emf.createEntityManager();
        em.getTransaction().begin();
    }

    @After
    public void rollbackTransaction()
    {
        em.getTransaction().rollback();
        em.close();
    }

    // Testmethoden
}
```

Diese Unit Tests lassen sich ausführen und protokollieren das gewünschte Verhalten, in unserem Beispiel zweimal das Auftreten einer `ConstraintViolationException` für das Attribut `name`. Abbildung 4-11 zeigt eine Testausführung.[12]

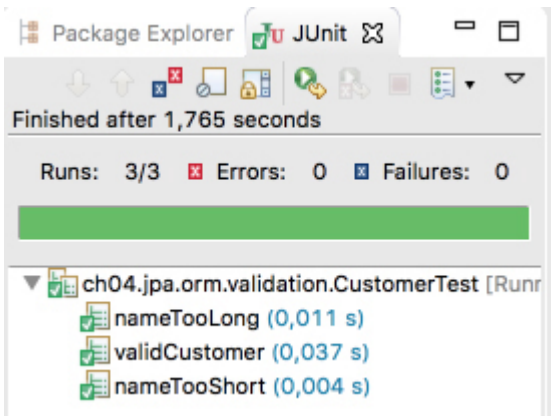


Abbildung 4-11 *Testausführung der Bean Validation in Eclipse*

In diesem Unit Test haben wir verschiedene Aufrufe im Kontext von Transaktionen gesehen. Nach dem Fazit zur Bean Validation möchte ich auf das Thema Transaktionen nochmals etwas genauer eingehen.

Fazit

Ich habe einen kurzen Schnelleinstieg in Bean Validation gegeben. Diese bietet auf einfache und deklarative Weise die Möglichkeit, gewisse Randbedingungen zu spezifizieren und sicherstellen zu lassen. Für eine ausführliche Besprechung verweise ich auf die weiterführende Literatur, die Sie am Kapitelende finden.

4.6 Transaktionen und Locking

In diesem Unterkapitel vertiefen wir unser Wissen zu Transaktionen und Locking, im Besonderen im Kontext von JPA.

4.6.1 Isolationslevel und Effekte

Bei der Beschreibung von Transaktionen habe ich auf die ACID-Eigenschaften hingewiesen. Diese sorgen dafür, dass Transaktionen unabhängig voneinander ausgeführt werden, erfordern aber auch eine Serialisierung, d. h. eine Hintereinanderausführung. Dies ist aber aus Performance-Gründen oftmals nicht erwünscht, sodass Datenbanken in der Regel einige Varianten der Isolation kennen. Damit lässt sich die Parallelausführung zwar verbessern, verursacht jedoch potenzielle Fehler. Diese sind so allgemeingültig, dass sie spezielle Namen besitzen. Nachfolgend nenne ich diese und beschreibe, wie sie zustande kommen:

- **Dirty Read** – Damit ist gemeint, dass von anderen Transaktionen geschriebene Daten bereits sichtbar sind und gelesen werden können, obwohl die Transaktionen noch gar nicht committet wurden. Potenziell könnten die Daten ungültig sein und später durch ein Rollback zurückgesetzt werden.
- **Non-repeatable Read** – Ein etwas weniger schlimmer Fall wird durch den Non-repeatable Read beschrieben – man spricht auch von Stale Data, also veralteten Daten. Der Name rührt daher, dass während einer Transaktion beim mehrfachen Lesen von Daten eventuell unterschiedliche Werte

zurückgeliefert werden, wenn zwischenzeitlich andere Transaktionen Daten schreiben.

- **Phantom Read** – Die schwächste Form von Inkonsistenzen ist unter dem Namen Phantom Read bekannt. Hierbei kommt es beim Auslesen von Daten zu zwei unterschiedlichen Zeitpunkten zu einer abweichenden Ergebnismenge, weil durch zwischenzeitlich beendete Transaktionen Datensätze hinzugefügt und/oder gelöscht wurden.

Je nach gewähltem Isolationslevel kann nur eine Teilmenge der zuvor aufgelisteten Fehlersituationen auftreten. Dabei sind folgende Isolationslevel definiert, die jedoch nicht von allen Datenbanken gleichermaßen unterstützt werden:

- **Read Uncommitted** – Bietet die geringste Isolation, aber höchste Performance. Es können allerdings alle drei der Fehler auftreten, also Dirty Reads, Non-repeatable Reads und Phantom Reads.
- **Read Committed** – Es werden nur committete Daten gelesen, wodurch keine Dirty Reads mehr vorkommen können. Jedoch sind immer noch Non-repeatable Reads und Phantom Reads möglich.
- **Repeatable Read** – Dem Namen zufolge werden hier verlässliche Lesezugriffe garantiert, wodurch Dirty Reads und Non-repeatable Reads ausgeschlossen sind. Trotzdem kann es zu Phantom Reads kommen.
- **Serializable** – Dieses Level bietet die höchste Isolation, aber auch die geringste Performance. Keiner der problematischen Reads kann auftreten.

4.6.2 Problemkontext

Wenn eine Datenbank von mehreren Benutzern oder Programmen verwendet wird, so ist es potenziell möglich, dass Datensätze konkurrierend verändert werden. In der Regel möchte man derartige Situationen vermeiden, insbesondere möchte man verhindern, dass ein Benutzer seine Änderungen verliert, weil ein anderer diese unbemerkt überschreibt. Statt bei menschlichen Benutzern sind solche Szenarien natürlich auch für unterschiedliche Serverprozesse denkbar.

Bekanntermaßen dienen Transaktionen dazu, eine Folge von Aktionen als eine Gesamtheit abzuarbeiten. Nur wenn alle Einzelaktionen erfolgreich abgeschlossen werden können, dürfen diese auch bestehen bleiben. Bei einem

Fehler müssen die Änderungen vollständig wieder rückgängig gemacht werden. Wie kann man dies erreichen?

Im allereinfachsten Falle könnte man dies dadurch lösen, dass man das Isolationslevel `Serializable` wählt. Das führt jedoch dazu, dass alle Tabellen, die in einer Transaktion angesprochen werden, während der Zeitdauer dieser Transaktion nicht durch andere Transaktionen abgefragt oder modifiziert werden können. Das wiederum bedingt, dass andere Transaktionen in ihrer Abarbeitung aufgehalten werden. Müssten die Transaktionen aber nur auf verschiedenen Daten arbeiten, so wäre diese Abschottung gar nicht notwendig. Deshalb existieren die gerade vorgestellten Isolationslevel.

Allerdings könnte es bei gelockerter Isolation folgenden Ablauf geben, der trotz seiner nur drei Schritte doch zwei potenzielle Probleme enthält:

1. Transaktion A lädt Daten.
2. Transaktion B lädt und verändert die Daten und führt einen Commit aus.
3. Basierend auf den im ersten Schritt gelesenen (und mittlerweile veralteten) Daten setzt Transaktion A die Arbeit fort und schreibt Ergebnisse in die Datenbank.

Die nicht mehr strikte Isolation erlaubt eine Verzahnung der Transaktionen und in diesem Fall würden die Berechnungsergebnisse von Transaktion B überschrieben werden, und schlimmer noch, Transaktion A würde nach dem Ende von Transaktion B mit veralteten Daten arbeiten. Beim »Verschlucken« von Änderungen anderer Transaktionen spricht man auch von ***Lost Update***. Das Weiterarbeiten auf veralteten Daten kann man dem Non-repeatable Read zuordnen. Beide Probleme sind ohne geeignete Gegenmaßnahmen nicht zu erkennen. Wie könnten diese aussehen? Die Antwort lautet: durch eine Versionierung und den Einsatz von Optimistic Locking. Das schauen wir uns nun genauer an.

4.6.3 Optimistic Locking

Als Abhilfemaßnahme gegen unerkannte Modifikationen durch andere Transaktionen dient das Optimistic Locking. Wie erkennt man nun aber Modifikationen? Recht unpraktisch wäre es, die Daten an sich vergleichen zu müssen. Es gibt aber eine praktikablere Lösung: Jedes Objekt erhält eine Versionsnummer, die bei Modifikationen hochgezählt wird. Auf diese Weise

können beim Optimistic Locking konkurrierende Schreibzugriffe durch Abweichungen in der Versionsnummer erkannt werden. Dazu wird vor den Aktionen auf einem Objekt die gerade aktuelle Versionsnummer ausgelesen. Beim späteren Schreibvorgang wird geprüft, ob der derzeit in der Datenbank gespeicherte Datensatz immer noch diese Versionsnummer besitzt. Stimmt die Version noch überein, hat zwischenzeitlich keine Änderung stattgefunden. Das Schreiben ist problemlos möglich. Wird eine andere Versionsnummer vorgefunden, so muss die Transaktion abgebrochen werden, weil ein konkurrierender Schreibzugriff vorliegt. Praktischerweise erfolgen diese Prüfungen automatisch, sofern man Optimistic Locking aktiviert hat (JPA) oder aber die richtigen SQL-Anweisungen schreibt, etwa wie folgt:

```
UPDATE Personen SET Name = 'Changed', VERSION = 7
                  WHERE ID = 4711 AND VERSION = 6;
```

Die für die Versionsprüfung verwendete Spalte `VERSION` wird bei JPA automatisch in das Datenbankmodell aufgenommen – bei SQL muss dies manuell und als expliziter Schritt geschehen.

Das im Listing gezeigte spezielle `UPDATE`-Kommando führt die Änderungen atomar aus, und zwar nur dann, wenn auch die Versionsnummer der erwarteten Versionsnummer entspricht. Das ist aber nur der eine Aspekt. Wie erkennt man aber, dass es einen Konflikt gegeben hat? Bei der Besprechung von JDBC, genauer dem Ausführen von SQL-Kommandos in Abschnitt 3.1.1, habe ich erwähnt, dass SQL-Kommandos die Anzahl der durch sie modifizierten Datensätze zurückliefern. Für obiges Kommando ist das beim ersten Update also der Wert 1 und bei folgenden Updates der Wert 0. Letzteres signalisiert also auf Ebene von SQL, dass es einen Konflikt gegeben hat. In JPA führt dies dann zu einer `javax.persistence.OptimisticLockException`.

Hinweis: Pessimistic Locking

Ergänzend zum Optimistic Locking gibt es noch das Pessimistic Locking, das jedoch erst seit JPA 2 unterstützt wird. Hierbei werden die Datensätze exklusiv gesperrt. Dazu nutzt man ähnlich wie bei Java einen speziellen Lock. Andere Transaktionen können auf den Datensatz so lange nicht mehr zugreifen, bis der Lock wieder freigegeben wird. Wie immer bei Locking kann es auch zu Deadlocks kommen, wobei sich Transaktionen gegenseitig blockieren.

Optimistic Locking in JPA

In JPA ist es auf einfache Weise möglich, Optimistic Locking zu unterstützen. Dazu muss in der Entitätsklasse ein spezielles Attribut hinzugefügt werden, das eine Versionsnummer speichert. Der JPA-Provider verwaltet dieses, wenn das Attribut mit `@Version` annotiert ist. Idealerweise verwendet man einen numerischen Typ, etwa ein `int`- oder `long`-Attribut.[13]

```
@Entity
public class Customer
{
    @Id
    @GeneratedValue
    private Long id;

    @Version
    private int version;

    @NotNull
    @Size(min = 5, max = 20)
    private String name;

    public Customer()
    {
    }

    public Customer(final String name)
    {
        this.name = name;
    }

    // …
}
```

Wird ein Verstoß gegen die Eindeutigkeit festgestellt, wurde also von einer anderen Transaktion eine Änderung vorgenommen, bevor die eigene Transaktion ihre Daten speichern kann, dann wird eine `OptimisticLockException` ausgelöst.

Fazit und Vorteile des Optimistic Locking

Ich konnte aus Platzgründen das Thema Transaktionen im Kontext von JPA nur einführend darstellen. Detaillierte Informationen finden Sie aber in der am Kapitelende genannten weiterführenden Literatur.

Fassen wir die Eigenschaften von Optimistic Locking kurz zusammen: Damit ist es möglich, für Konsistenz zu sorgen. Der gleiche Schreibzugriff kann niemals mehrfach erfolgen und es wird sichergestellt, dass immer aktuelle Daten sichtbar sind. Daneben lässt sich Optimistic Locking wie folgt charakterisieren:

- Es erfordert nur einen minimalen Mehraufwand.
- Es ist fast immer sehr schnell, weil kein aufwendiges Locking erfolgen muss.
- Durch die fehlenden Sperren können auch länger laufende Transaktionen oftmals problemlos ausgeführt werden.
- Es lässt sich Konsistenz erzielen, da die Schreibzugriffe durch die Versionsnummer immer geordnet ablaufen. Bei parallelen Modifikationen schlägt die zweite immer fehl.
- Konkurrierende Schreibzugriffe lösen eine Exception aus. Zudem lassen sich Repeatable Reads garantieren.

Halten wir Folgendes abschließend fest: Optimistic Locking ist vor allem dann die richtige Wahl, wenn es deutlich mehr Lese- als Schreibzugriff gibt und nur wenige Kollisionen erwartet werden.

4.7 Caching in JPA

Gerade bei Zugriffen auf Datenbanken kann Caching zu einer signifikanten Verbesserung der Performance führen. Im Bereich der Persistenz sind unter anderem Datenbankverbindungen mögliche Kandidaten, die man cachen kann. Seit JPA 2 unterstützt der Standard durch die Annotation `@Cacheable` auch das Caching für Entities:

```
@Entity
@Cacheable
public class Employee
{
  …
}
```

Im Anschluss stelle ich vor, was diese Annotation bewirkt. Zuvor gehe ich aber zum besseren Verständnis der Abläufe kurz auf verschiedene Formen des Cachings ein.

JPA-Caching im Überblick

JPA unterstützt zwei verschiedene Arten von Caches: einen First Level Cache und einen Second Level Cache. Letzterer muss explizit per Konfiguration aktiviert werden, ersterer ist standardmäßig aktiv. Wo liegen darüber hinaus die Unterschiede?

Um ein besseres Verständnis zu gewinnen, schauen wir zunächst, was passiert, wenn Objekte bzw. Entitäten durch einen `EntityManager` geladen werden. In diesem Fall werden diese automatisch in einem `EntityManager`-bezogenen Cache zwischengespeichert. Das ist der ***First Level Cache***. Immer wenn in der Folgezeit auf das gleiche Objekt per Schlüssel zugegriffen wird, erfolgt keine Datenbankabfrage mehr, sondern es wird das Objekt aus dem Cache zurückgeliefert.

Nun gibt es aber folgende Beschränkung: Dieser First Level Cache wirkt immer für eine Instanz eines `EntityManager`s. Wenn wir uns eine typische Webapplikation anschauen, entsteht dadurch jedoch folgendes Problem: Für jeden neuen Request existiert ein eigener `EntityManager` und somit werden die Daten meistens nicht besonders lange vorgehalten.[14] Zudem profitieren Abfragen in anderen Requests nicht von den bereits ermittelten Daten. Dadurch kann es durchaus sein, dass eine Liste von Benutzern oder Bestellungen mehrfach innerhalb kürzester Zeit geladen wird.

Hier kommt nun ein sogenannter ***Second Level Cache*** ins Spiel, der zwischen `EntityManager`-Instanzen geteilt wird. Demzufolge profitiert man von zuvor ausgeführten Datenbankabfragen für alle anderen Abfragen gleicher Art. Ein Second Level Cache als Zwischenspeicher für Entitäten kann somit die Performance erheblich verbessern, indem teure Zugriffe auf die Datenbank vermieden werden. Das geschieht transparent für die Applikation, was bedeutet, dass diese nichts von dem Cache weiß, also sowohl Lese- als auch Schreibzugriffe über ganz normale Methoden des `EntityManager`s abgewickelt werden.

Cache-Modi zur Einflussnahme

Um das Caching beeinflussen zu können, lassen sich verschiedene Cache-Modi konfigurieren. Folgende sind durch JPA definiert:

- `ALL` – Alle Entities werden im Second Level Cache gehalten.
- `NONE` – Es werden keine Entities für die Persistence Unit gecacht.
- `ENABLE_SELECTIVE` – Es werden nur solche Entities gecacht, die mit der Annotation `@Cacheable` bzw. `@Cacheable(true)` markiert sind.
- `DISABLE_SELECTIVE` – Es werden alle Entities gecacht, bis auf diejenigen, die explizit mit der Annotation `@Cacheable(false)` davon ausgeschlossen wurden.

- `UNSPECIFIED` – Das Verhalten ist nicht spezifiziert und es wird der Default vom verwendeten JPA-Provider genutzt.

Wenn man sich nicht ganz sicher ist, welchen Modus man wählen sollte, so stellt `ENABLE_SELECTIVE` fast immer eine recht vernünftige Wahl dar, weil hierbei all diejenigen Entities gecacht werden, für die man dies explizit vorgegeben hat.

Die zweitbeste Wahl ist wohl `DISABLE_SELECTIVE`, weil hier einige Entitäten explizit vom Caching ausgeschlossen werden.

Dagegen bietet `NONE` keinen Vorteil und `ALL` ist so unspezifisch, dass es durch das Caching von vermutlich nicht cache-würdigen Entitäten zu Speicherplatzverschwendung kommt und sich dieses oftmals sogar kontraproduktiv auswirkt. Potenziell steht dadurch der eigentlichen Applikation nicht mehr genügend Speicher zur Verfügung.

Vor- und Nachteile beim Caching

Der Einsatz von Caching kann zu enormen Verbesserungen in der Performance führen, weil Daten nicht mehr von der Datenbank, sondern aus dem im Hauptspeicher beheimateten Cache gelesen werden. Allerdings sollte man auch an einige Nachteile denken: Ein Cache belegt Hauptspeicher – mitunter einen nicht unerheblichen Teil. Zudem birgt die Zwischenspeicherung von Daten im Hautspeicher die Gefahr für Inkonsistenzen mit den Werten aus der Datenbank, etwa, wenn es zwischenzeitlich zu Änderungen in der Datenbank gekommen ist, die noch nicht mit dem Cache abgeglichen wurden.

Schlussgedanken zum Caching

Aus Platzgründen konnte ich für das Thema Caching lediglich einen kurzen Überblick geben. Weiterführende Information finden Sie unter `https://en.wikibooks.org/wiki/Java_Persistence/Caching`. Verschiedene Caching-Strategien sowie ihre Auswirkungen werden anschaulich auf der Seite `http://www.thejavageek.com/2014/09/25/jpa-caching-example/` dargestellt.

4.8 Fazit

Dieses Kapitel hat einen einführenden Überblick über das JPA gegeben. Damit sollten Ihnen sowohl der Einstieg als auch weitere Schritte leicht fallen. Dabei ist es hilfreich, dass JPA weniger Komplexität als andere Lösungen erfordert, um

ein RDBMS anzubinden: In Enterprise-Applikationen kann man auf Entity Beans verzichten und in Java-SE-Anwendungen werden keine aufwendigen Realisierungen der Datenzugriffe per JDBC mehr benötigt. Stattdessen kann die Datenzugriffsschicht in weiten Teilen unverändert in Java-EE- und Java-SE-Anwendungen benutzt werden. Unterschiede ergeben sich jedoch aufgrund der Art der Transaktionssteuerung und des Zugriffs auf den `EntityManager`. Für die eigentlichen Entitätsklassen muss das allerdings nicht berücksichtigt werden.

4.9 Weiterführende Literatur

Zu den Themen Datenbanken und JPA ist einiges an Literatur erschienen. Folgende Aufzählung nennt einige empfehlenswerte Bücher:

- »**Java Persistence with Hibernate**« von Christian Bauer und Gavin King [2]

  Dieses exzellente Buch bietet eine umfassende und fundierte Einführung in das Persistenz-Framework Hibernate.

- »**Pro JPA 2**« von Mike Keith und Merrick Schincariol [11]

  Dieses Buch behandelt JPA in der aktuellen Version 2.1. Es fängt bei Grundlagen an und baut diese Stück für Stück aus. Dabei behandelt es eine Vielzahl an JPAThemen.

- »**Java Persistence API 2**« von Bernd Müller und Harald Wehr [14]

  Dieses Buch bietet eine verständliche, gut lesbare und fundierte Einführung in die Persistenz mit JPA 2 – ohne jedoch genauer auf die aktuelle Version 2.1 einzugehen.

Index

1:n-Beziehung, 91
1NF, 84
Abbildungstabelle, 92
Abfrage, 178
    ausführen, 186
    eingebettete Dokumente, 251
    parametrierte, 179, 186
    typsichere, 188
Abfragen
    mit dem Client-API, 279
Absatz, 364
AbstractTableModel, 135, 139
ACID-Eigenschaften, 85, 211, 235
Administrations-GUI, 333, 336
Aggregation, 178
    Abbildung von, 91
Aggregatsfunktion, 111
Ajax, 328, 378
Alias, 110, 125
ALTER COLUMN, 105
ALTER TABLE, 105
Anforderung
    neue, 177
    sich ändernde, 177
Anfrage, 355

Anfragelast
    hohe, 222
Anwendungszustand, 75
Apache OpenJPA, 162
Application, 20
Applikation
    verteilte, 193
ARM, 120, 122
Arrange Act Assert, 304
Arrays, 372
Assoziation, 196
    Abbildung von, 91, 197
Asynchronous JavaScript and XML, 328, 378
Atomcity, 85
AtomicLong, 322
Attribut, 2, 5, 23
AttributeConverter, 204
Auditing, 267, 295, 341
Ausfallsicherheit, 114
Auszeichnungssprache, 1, 363
    Motivation von, 363
Authentifizierung, 86, 122, 299
Auto-Commit-Modus, 148
    Auswirkungen, 148
    Schwächen, 148
Automatic Resource Management, 120, 122
Autorisierung, 86, 122
B*-Baum, 81

Base64, 299
Basic Auth, 299
BasicDBObject, 244
Baum
    binärer, 81
    geordneter, 81
    Sortierkriterium, 81
Baumoperation, 21
Bean Validation, 207, 211
Betriebssystem, 3
BETWEEN, 111
Big Data, 220
Big Endian, 3
Body, 61, 357
BSON, 223
Build
    inkrementeller, 351
Builder, 274
Byte-Reihenfolge, 3
Cache-Modi, 217
Caching
    Nachteile, 217
    Vorteile, 217
CAP-Theorem, 238
CascadeType, 197
Cascading Stylesheet, 336, 368
CDATA, 11
Chunk, 240

Client, 270, 276, 355
Client-Server-Kommunikation, 270, 272, 355
ClientBuilder, 276
Collection-Literal, 354
column, 79
Comma Separated Values, 3
Commit, 85, 148
Common Object Request Broker Architecture, 282
composite primary key, 82
Connection, 118, 120, 124, 127, 134
    getMetaData(), 134
Connection Pooling, 127
Connection-Pool, 127
Consistency, 85
Constraints, 95
ConstraintViolationException, 211
Content Negotiation, 271
ContentHandler, 15–17, 19, 62
Convention over Configuration, 166, 261
CORBA, 282
CREATE INDEX, 106
CREATE TABLE, 101
Criteria API, 188, 189, 191
CRUD, 97, 158, 172, 191, 227, 261, 272
CRUD-Aktionen, 258, 261–263
CRUD-Applikation, 260, 283, 299
CRUD-Funktionalität, 191, 285, 288
CRUD-Operationen, 257, 261, 288

CSS, 43, 336, 368
CSV, 3, 44, 307, 312
    XSLT-Konvertierung in, 44
CSV-Datei, 307
CSV-Import, 308
CURL, 330
Cursor, 128, 130
Cursor-API, 29
DAO, 151, 156, 157, 191, 192, 194, 195, 257, 259
    Realisierung eines, 158
DAO-Funktionalität, 191
Data Access Object, 151, 156, 157, 191
Data Analytics, 220
Data Definition Language, 100, 101
Data Manipulation Language, 100, 111
Data Query Language, 100, 107
Data Transfer Object, 174, 233
DatabaseMetaData, 134
Daten
    direkt als JSON verarbeiten, 245
Datenabfrage, 107
Datenbank, 75
    Migration, 224
    NoSQL-, 219
    relationale, 79, 117
    Schritte zur Abfrage, 120
    Tabelle, 79
Datenbankadministrationstool, 195

Datenbankadministrator, 86
Datenbankanbindung, 117
Datenbankmanagementsystem, 75
Datenbankmodell, 161, 175
    Aufbau, 83
    Erzeugung, 176
Datenbankressource
    freigeben, 119
Datenbanksystem, 75
Datenbanktreiber, 117, 118
    einbinden, 121
    laden, 121
Datenbankverbindung
    aufbauen, 121
    schließen, 127
Datenbankzugriff
    per JPA, 168
Datenbankzugriffstool, 175
Datendurchsatz, 114
Datenkonsistenz, 85
Datenmanipulation, 111
Datenmenge
    wachsende, 220
Datenmigration, 177
Datenmodell, 177, 219
Datenpersistenz, 75
Datensatz, 79
    Eindeutigkeit, 80

Identifizierung, 80
Identität, 80
Suchzugriff, 81
Datensatzzeiger, 130, 136
Datenverlust
Vorsorge gegen, 114
DateTimeFormatter, 309
DB, 75, 242
DBCollection, 242
DBCursor, 244
DBMS, 75
DBObject, 244
DDL, 100, 101
DDL-Generierung, 176
DefaultHandler, 16, 17
DELETE FROM, 113
Dependency Injection, 261
Deployment, 269
Desktop-Anwendung
klassische, 328
Detached Object, 174, 193
Dirty Read, 150, 212
Diskriminatorspalte, 93, 95
DML, 100, 111
Document, 23, 243, 252, 254
Document Object Model, 1, 14
Document Store, 222, 241
Grundlagen, 222

MongoDB, 222
Document Type Definition, 1, 10
DocumentBuilder, 21
DocumentBuilderFactory, 21
Dokumente
eingebettete, 251
DOM, 1, 14
Baumoperation, 21
Parsing mit, 21
speichern, 25
Speicherung in einer Datei, 26
verarbeiten, 25
DOM-Baum, 328
Domänenmodell, 224
DOMSource, 26
DQL, 100, 107
DriverManager, 121
DTD, 1, 10, 53
externe, 11
Verweis auf, 11
DTO, 174, 233
Durability, 85
Eager Fetch, 156
Eager Loading, 156, 163
EclipseLink, 162
Eindeutigkeit, 101
Element, 2, 5, 23, 25
End-Tag, 2

Entität, 79, 164
    Lebenszyklus von, 173
Entity, 163
    detached, 173
    managed, 173
    new, 173
    removed, 173
Entity Bean, 161
Entity Lifecycle, 173
Entity-Relationship-Diagramm, 101
EntityManager, 169, 170
    Methoden des, 171
EntityManagerFactory, 169
Envelope, 61
ER-Diagramm, 101
Event-Iterator-API, 33
Eventual Consistency, 86, 114
eXtensible Markup Language, 1
eXtensible Stylesheet Language, 1
Fehlertoleranz, 114, 282
Filter
    Request, 295
    Response, 295
FIND, 228
FindIterable, 243
First Level Cache, 216
foreign key, 89
foreign key constraint, 105

Format
    proprietäres, 1
    textbasiertes, 1
Formular, 328, 367
    Interaktivität, 366
Fremdschlüssel, 89
    Modellierung, 90
Fremdschlüsselbeziehung, 105
    prüfen, 113
Full Table Scan, 81, 233
Funktionen, 372
Gesprächigkeit, 67
Gleichheit, 80
Gradle
    Abhängigkeiten definieren, 352
    Builds mit, 347
    eigene Tasks definieren, 353
    Einführung, 345
    externe Abhängigkeit spezifizieren, 351
    IDE-Projekte erzeugen, 354
    Installation, 347
    JAR erstellen, 353
    Projektlayout, 345
    Projektstruktur, 345
    Sourcen kompilieren, 350
    Unit Tests ausführen, 351
    verfügbare Tasks, 349
Hash Based Sharding, 240

Header, 357
Hibernate, 161, 162
Hierarchieebenen, 2
HSQLDB, 98, 175
    Mac OS, 99
HTML
    Absatz, 364
    Basiswissen, 363
    Formular, 328, 367
    Interaktivität, 366
    Liste, 364
    Tabelle, 364
    XSLT-Konvertierung in, 42
HTTP, 355, 356
    Basiswissen, 357
    Body, 357
    Header, 357
    Kommunikation, 356
HTTP Basic Auth, 299
HTTP-Befehl, 357
HTTP-Header, 271
HTTP-Kommando, 273, 285
HTTP-Statuscode, 291, 358
HttpURLConnection, 361, 362
Hypertext Transfer Protocol, 355
Idempotenz, 271
Identität, 80
Identitätsattribut, 165

IFUR, 227
IFUR-Aktionen, 243
Impedance Mismatch, 87, 196, 223, 252
Index, 81, 233
    Auswirkungen eines, 106
    erstellen, 106
Index erstellen, 106
InheritanceType, 200
Inkonsistenz
    Gefahr von, 78
Inkonsistenzen, 77
INSERT, 228
INSERT INTO, 111
Installation
    Gradle, 347
Integrität, 114
    referenzielle, 94
Interzeptor, 298
Isolation, 85
Isolationslevel, 222, 235
Jackson, 67, 71
JAR
    erstellen, 353
Java API for XML Processing, 14
Java Architecture for XML Binding, 1, 49
Java Database Connectivity, 117
Java first, 50, 54
Java Persistence API, 117, 161

Java Persistence Query Language, 164
JavaBean, 46
    Mechanismus, 56
JavaBean-Mechanismus, 56
JavaScript, 67
    Arrays, 372
    Engine, 67
    Funktionen, 372
    Grundlagen zur Sprache, 371
    implizite Objekte, 375
    Modifikation von HTML, 373
    REST-Services ansprechen, 378
    Variablen, 371
    Wissenswertes zu, 371
JavaScript Object Notation, 1, 67, 223, 377
JAX-RS, 269, 272
    Annotations, 272, 274
JAX-RS-Annotations, 274
JAXB, 1, 49
    Annotations, 50
    Lesen von Objekten, 59
    On-the-Fly-Modifikation, 65
    Rekonstruktion von Objekten, 59
    Schreiben von Objekten, 61
    Schwächen, 58
    Stärken, 58
    Verarbeitung von Collections, 56
JAXB_FRAGMENT, 65

JAXBContext, 54
JAXP, 14
JDBC, 117, 161
    Transaktion, 148
JDOM, 23
Jersey, 269, 272
JettisonMappedXmlDriver, 69
Join, 109, 110
Joined, 94, 95
JPA, 117, 161
    Assoziation, 196
        Abbildung von, 197
    Bestandteile von, 163
    Caching, 216
    DAO, 192
    DAO-Funktionalität, 191
    Datenbankzugriff, 168
    Grundlagen zum, 162
    Optimistic Locking, 214
    Transaktion, 170
    Transaktionskontext, 170
    Vererbung, 196
        Abbildung von, 200
JPQL, 164, 178, 191
    Abfragen mit, 182
    im Überblick, 178
    Modifikationen mit, 180
    Queries in, 178

SQL-Queries
native, 182
Syntax von, 178
vordefinierte Funktionen, 178
JSON, 1, 67, 223, 252, 269, 336, 377
Jackson, 71
Verarbeitung von, 377
XStream, 69
JTable, 139
Kommentar, 2, 7, 105
Kommunikation, 269
Ablauf der, 359
Client-Server-, 355
Request-Response-, 355
synchrone, 282, 357
verteilte, 281
Kommunikationspartner, 282
Komparator, 312
Konfigurationsdatei
persistence.xml, 167
Konfigurationsverwaltung, 122
Konsistenz, 114
Gewährleistung von, 224
Sicherstellung von, 84
Konsistenzbedingung
prüfen, 235
Konverterklasse, 204
Kreuzprodukt, 109

Längenbeschränkung
    Probleme, 325
    Verbesserung der, 325
Lambda, 308
Lastverteilung, 114
Lazy Fetch, 156
Lazy Loading, 156, 163
Lebenszyklus
    von Entitäten, 173
Liste, 364
Little Endian, 3
Locking, 211
Logging, 267, 295
Lost Update, 213
m:n-Beziehung, 92
Mac OS, 99
Markup, 1
Markup Language, 1
Marshaller, 54
Marshaller.JAXB_FRAGMENT, 65
Marshalling, 49, 54
Maschine
    virtuelle, 269
Master, 114
MathML, 4
MediaType, 273, 275
Metadaten
    Abfrage von, 134

Connection
getMetaData(), 134
Ermittlung von, 134
objektrelationale, 163, 165
ResultSet
getMetaData(), 134
Methodenaufruf
entfernter, 269, 282
Microservice, 268
Middleware, 282
Migration, 224
Migrationshilfe, 244
Modifikation von HTML, 373
Mongo-Client, 225–227
MongoClient, 242
MongoCollection, 242
MongoDatabase, 242
MongoDB, 219, 327, 336, 339
Abfrage
eingebettete Dokumente, 251
aus Java ansprechen, 241
Ausfallsicherheit, 236, 238
Document Store, 222
Dokumente
eingebettete, 251
Einführung, 225
Enum verarbeiten, 255
Fehlertoleranz, 231

FIND, 228
Index, 233
INSERT, 228
komplexere Abfragen, 232
Merkwürdigkeit, 231
pretty(), 228
Protokollierung in, 336
Protokollierung von Aktionen, 336
REMOVE, 231
Replica Set, 238
Sharding, 239
Skalierbarkeit, 236
Skalierung, 239
Suchbedingung, 229
Transaktionen, 235
UPDATE, 230
Verarbeitung
komplexerer Daten, 247
von Enums, 248
MongoRepository, 258, 261
MonogDB, 340
n-Tupel, 78, 79
Objekt als, 78
Nachrichtenaustausch, 269
Named Queries, 181
NamedNodeMap, 23, 24
Namensraum, 9
natürliche Ordnung, 312

Node, 23, 24
NodeList, 23, 24
Noise, 73
Non-repeatable Read, 212
Normalform, 83, 224
    erste, 84
Normalisierung, 83, 224
NoSQL
    Column Stores, 221
    Document Stores, 220
    Dokumente, 219
    Graph DBs, 221
    Graphen, 219
    Key-Value-Stores, 220
    Namensgebung, 219
    Schlüssel-Wert-Paare, 219
NoSQL-Datenbank, 219
Object Relational Impedance Mismatch, 87
Object-Relational Mapping, 117, 161
Object/Document Mapping, 223, 252, 255
    JSON-basiertes, 252
ObjectMapper, 71
Objekt
    als n-Tupel, 78
    als Tabelleneintrag, 79
    Gleichheit, 80
    Identität, 80
Objektgraph, 77

Zyklus im, 78
objektrelationale Metadaten, 163
objektrelationales Mapping, 86
Objektzustand, 75
ODBC, 131
ODM, 223, 252, 255
für komplexere Java-Objekte, 254
Spring Data MongoDB, 255
On-the-Fly-Modifikation, 65
Optimistic Locking, 213
OptimisticLockException, 215
OR-Mapping
Grundlagen zum, 162
Wissenswertes zum, 162
Ordnung
natürliche, 312
ORM, 86, 117, 161, 196
Abbildung von Aggregationen, 91
Abbildung von Assoziationen, 91
Abbildung von Referenzen, 88
Abbildung von Vererbung, 94
am Beispiel, 87
Grundlagen zum, 162
Joined, 95
mit JDBC, 151
Single Table, 94
Table per Class, 96
Wissenswertes zum, 162

overly verbose, 4
Paging, 188, 294
Parameter
    benannte, 180
    indexbasierter, 180
Parser, 5, 14
Parsing, 6
PCDATA, 11
Performance-Probleme, 202
Persistence Context, 171
Persistence Unit, 168, 169
Persistenz, 75
    Grundlagen zur, 76
    konfigurative, 177
Persistenz-Framework, 161, 163
Persistenz-Komponente, 313
Persistierung
    von Objekten, 78
Pessimistic Locking, 214
Phantom Read, 212
Plain Old Java Object, 68, 163
POJO, 68, 163
Postman, 281, 330
Präsenz, 282
Prüfung
    inhaltliche, 9
Predicate, 295
PreparedStatement, 145

pretty(), 228
Primärschlüssel, 82, 101, 165
    künstlicher, 82
    natürlicher, 82
    zusammengesetzter, 82
primary key, 82
Processing Instruction, 6
Programmiersprache, 269
Projektion, 233
Projektlayout, 345, 347
Projektstruktur, 345
Prolog, 6
Protokoll, 356
Protokollierung, 267
    von Methodenaufrufen, 296
Pull, 29
Push, 29
Queries
    besondere Arten von, 181
Queries mit Objekterzeugung, 181
Query, 182, 183
    Ergebnisse von, 183
Query By Example, 229
Range Based Sharding, 240
RDBMS, 81, 117, 161, 219
    ein Blick zurück, 221
    Impedance Mismatch, 223
Read Committed, 212

Read Uncommitted, 212
Reader, 29
Rechteprüfung
    rollenbasierte, 302
record, 79
Redundanz, 77, 83
    Vermeidung von, 83, 224
Redundanzfreiheit, 78, 83
Referenz
    Abbildung von, 88
    Speicherung von, 77
Referenzielle Integrität, 113
Reflection, 56
Remote Method Invocation, 269, 282
Remote Procedure Call, 282
REMOVE, 231
Repeatable Read, 212
Replica Set, 236
Replikation, 114
Repository, 257, 259, 261
Request, 270, 355
Request-Filter, 295
Request-Response-Kommunikation, 355
Response, 270, 276, 355
Response-Filter, 295
Ressource, 270
    adressierbare, 269
    Adressierung von, 322

bereitgestellte, 273
Pfad der, 270
REST, 269
Abfragen
mit dem Client-API, 279
CRUD, 272
CRUD-Funktionalität, 283
Datenformat, 283
Design von Interfaces, 290
Eingabetypen, 274
Error Handling bei, 290
MIME-Types, 283
Netzwerktransparenz, 272
Rückgabewerte, 274
vielfältige Formate, 272
REST-Client-API, 276
REST-Service, 269, 270, 272
Erstellung eines, 323
Zugriff auf, 276
Zugriff mit CURL, 276
restfuse, 302, 304
ResultSet, 118, 120, 125, 127, 129–134, 136
Änderungen an, 131
Behandlung von NULL-Werten, 133
Besonderheiten von, 128
getMetaData(), 134
Konfiguration, 129
Positionierung

absolute und relative, 130
ResultSetMetaData, 134, 138
RMI, 269, 282
Rollback, 85, 148
row, 79
RPC, 282
SAX, 1, 14
SAXParser, 15, 17
SAXParserFactory, 17
Scalable Vector Graphics, 4
scale out, 236
scale up, 236
schemafrei, 222
Schemafreiheit, 223, 224
Diskussion zu, 223
Schlüssel, 82
Second Level Cache, 216
Security, 299
SELECT FROM, 107
Serialisierung, 211, 311, 317
Serializable, 212
Server, 270, 355
Sharding, 236
Sharding-Strategien, 240
Short-Circuit-Prinzip, 36
Simple API for XML, 1, 14
Simple Object Access Protocol, 269
Single Table, 94

Singleton, 323
Slave, 114
SOAP, 269, 282
    Body, 61
    Envelope, 61
SOAP-Nachricht, 61
Sonderzeichen, 369
Sortierfunktionalität, 312
Sortierrichtung, 233
Sortierung, 108, 312
Speicherplatzverschwendung, 77
Speicherung
    redundate, 77
Spezialzeichen, 7
Spring Data MongoDB, 254, 256, 257, 260, 261, 267
    CREATE, 261
    DELETE, 262
    komplexere Abfragen mit, 263
    ODM mit, 255
    READ, 261
    UPDATE, 262
    Vorteile von, 259
SpringApplication, 261
SQL, 76, 100
    Aggregatsfunktion, 111
    ALTER COLUMN, 105
    ALTER TABLE, 105
    BETWEEN, 111

Constraints, 101
CREATE INDEX, 106
CREATE TABLE, 101, 104
Datensatz ändern, 112
Datensatz hinzufügen, 111
Datensatz löschen, 113
Datentypen, 102
DDL, 101
DELETE FROM, 113
Dialekt, 102, 105
DML, 111
DQL, 107
FOREIGN KEY, 105
Fremdschlüssel definieren, 105
Grundlagen, 100
INSERT INTO, 111
Kommentar, 105
Konsistenzbedingungen, 101
SELECT FROM, 107
Sortierung, 108
Spalten ändern, 105
Spalten auslesen, 107
Tabellen ändern, 105
Tabellen erzeugen, 104
UPDATE, 112
Vergleichsoperation, 108
Wertebereich, 111
WHERE, 108

SQL-Anweisung
    ausführen, 124
    erzeugen, 124
    schließen, 127
SQL-Dialekt, 163, 177, 188
SQL-Injection, 143
SQL-Queries
    native, 182
SQLFeatureNotSupportedException, 130, 137
Start-Tag, 2
Statement, 118, 120, 124, 127
    Nachteile von, 144
    Schwachstellen von, 142
StAX, 1, 14, 29
    Cursor-API, 29
    Event-Iterator-API, 29, 33
    Lesen von Objekten, 59
    On-the-Fly-Modifikation, 65
    Rekonstruktion von Objekten, 59
    Schreiben von Objekten, 61
Strategy, 316
StreamResult, 26
Stringkonkatenation, 180
Structured Query Language, 76, 100
Strukturbeschreibung, 10
Strukturierung, 5
Subelement, 5
Suchbaum

binärer, 81
Suchzugriff, 81
SVG, 4
Syntaxfehler, 188
System
verteiltes, 237, 238, 269
Systemressource
belegen, 121
freigeben, 119, 121
Tabelle, 79, 364
Tabellengenerierung
dynamische, 330
Table per Class, 94, 96
Tag, 2, 5
öffnendes, 2, 5
benutzerdefiniert, 9
End-, 2
Kurzschreibweise, 2
schließendes, 2, 5
Start-, 2
TCP/IP, 356
Testbarkeit, 158
Top-N-Funktionalität, 324
Transaktion, 85, 170, 194, 210, 211, 222
benutzerdefiniert, 149
Datenkonsistenz, 85
in JDBC, 148
Isolationsgrade, 150

Isolationslevel, 222
Transaktionalität, 194
Transaktionskontext, 170, 194, 195, 222
Transaktionsverwaltung, 210
Transformer, 26
TransformerFactory, 26
Treffermenge
umfangreiche, 188
Trennung von Zuständigkeiten, 177
TypedQuery, 182, 183, 188, 189, 193
Uniform Resource Identifier, 270
Unmarshaller, 54
Unmarshalling, 54
unrepeatable reads, 150
UPDATE, 112, 230
URI, 270
URL, 362
URLConnection, 361, 362
valide, 3
Validierung, 10
Variablen, 371
Verarbeitung
komplexerer Daten, 247
von Enums, 248
Verbindungseinstellung
konfigurieren, 122
Vererbung, 196
Abbildung von, 94, 200

Vererbungshierarchie, 200
    tiefe, 202
Vergleichsoperation, 108
Versand
    von Nachrichten, 282
Versionierung, 177
verteilte Applikation, 193
W3C, 1, 8, 23
W3C-Spezifikation, 8
Web-GUI, 328, 336
Webapplikation, 193, 269, 356
WebEngine, 19
Weboberfläche, 328
Webseite
    zur Administration, 328
Webserver, 356
Webservice, 269, 356
WebTarget, 276
WebView, 19
Wertebereich
    vordefinierter, 111
WHERE, 108
wohlgeformt, 3, 8
World Wide Web Consortium, 1, 23
Wurzelelement, 2
xjc, 50
    DTD, 53
XML, 1, 67, 269

Abfragemöglichkeiten, 4
Attribut, 5
Austauschformat, 1
Beschreibung von Daten, 6
Element, 5
Extraktion von Daten, 5
Gesprächigkeit, 67
Hierarchieebenen, 2
Kommentar, 7
maschinelle Verarbeitung, 4
Namensraum, 9
Noise, 73
Processing Instruction, 6
Prolog, 6
Pull, 29
Push, 29
Robustheit, 4
Spezialzeichen, 7
Suchmöglichkeiten, 4
Tag, 2, 5
Transformierbarkeit, 4
Validierung, 10
XStream, 69
XML first, 50, 54
XML Schema Definition, 1, 10
XML-RPC, 282
XmlAdapter, 284
XMLDecoder, 46

XMLEncoder, 46, 48
XMLEvent, 65
XMLEventReader, 33, 65
XMLEventWriter, 34, 65
XMLHttpRequest, 378
XMLStreamConstants, 30
XMLStreamException, 49
XMLStreamReader, 29
XMLStreamWriter, 32, 48, 61
XPath, 37
    in Java, 38
    Syntax von, 37
XPath-Abfragen, 37
XPathConstants, 39
XPathExpression, 38
XPathFactory, 38
XSD, 1, 10
    Verweis auf, 14
XSD-Schema
    elementare Typen, 12
    komplexe Typen, 13
    Validierung mit, 12
XSL, 42
XSLT, 1, 4, 42
XSLT-Basiskommandos, 42
XStream, 67, 69
Zeichensatz-Encoding, 6
Zeichensatzcodierungen, 3

Zustandslosigkeit, 322

Zyklus, 78

Rezensieren & gewinnen!

Besprechen Sie dieses Buch und helfen Sie uns und unseren Autoren, noch besser zu werden.

Als Dankeschön verlosen wir jeden Monat unter allen neuen Einreichungen fünf dpunkt.bücher. Mit etwas Glück sind dann auch Sie mit Ihrem Wunschtitel dabei.

Wir freuen uns über eine aussagekräftige Rezension, aus der hervorgeht, was Sie an diesem Buch gut finden, aber auch was sich verbessern lässt. Dabei ist es egal, ob Sie den Titel auf Amazon, in Ihrem Blog oder bei YouTube besprechen.

Schicken Sie uns einfach den Link zu Ihrer Besprechung und vergessen Sie nicht, Ihren Wunschtitel anzugeben: www.dpunkt.de/besprechung oder besprechung@dpunkt.de

dpunkt.verlag GmbH · Wieblinger Weg 17 · 69123 Heidelberg fon: 06221/148322 · fax: 06221/148399